Analyzing Large-Scale Object-Oriented Software to Find and Remove Runtime Bloat

Dissertation

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Guoqing Xu
Graduate Program in Computer Science and Engineering

The Ohio State University

2011

Dissertation Committee:

Atanas Rountev, Advisor

Feng Qin

Michael D. Bond

ABSTRACT

Over the past decade, the pervasive use of object-oriented languages and the increasing complexity of problems solved by computer software have led to the proliferation of large-scale framework-intensive applications. These applications are typically built by combining standard packages (e.g., J2EE application server frameworks), third-party layers for domain-specific functionality, and in-house solutions. While employing libraries and frameworks eases the development effort, it can be expensive to invoke the general APIs provided by these libraries, especially when they are used for simple tasks. As a result, many applications suffer from excessive memory footprint caused by chronic runtime bloat that significantly impacts scalability and performance. In addition, programmers are taught to pay more attention to abstractions and models, and to leave the performance optimization to the runtime system. While a few redundant objects, calls, and field copies may seem insignificant, the problem quickly gets exacerbated due to nesting and layering. At some point, these inefficiencies accumulate, and compilers (e.g., the JIT in a JVM) can no longer eliminate them, since layers of abstractions grow to be deep, dynamic, and well beyond the capabilities of compiler analysis.

In this dissertation, the term *bloat* is used to refer to the general phenomenon of using excessive work and memory to achieve seemingly simple tasks. Bloat exists commonly in large-scale object-oriented programs and it is a major challenge that stands in the way of bridging the productivity-performance gap between managed and unmanaged languages. The overarching goal of our work is to find large bloat-related optimization opportunities, with a small amount of developer time. As a fundamental methodology to achieve this goal, we advocate *tool-assisted manual optimization*, in order to combine developer insight with the automated tool support. In particular, we have designed, implemented, and evaluated code checking techniques accomplished by bytecode-level static analysis, and runtime optimization techniques achieved by VM-level dynamic analysis, for detecting and removing runtime bloat in large framework-intensive Java applications. These techniques can be used to (1) help a programmer *detect* the root cause if a problem is seen; (2) *remove* frequently-occurring bloat patterns; and (3) *prevent* bloat from occurring in an early stage of development.

Incorporating these techniques in an existing Java compiler or virtual machine will lead to increased developer productivity by helping programmers quickly locate bloat-related performance bottlenecks, as well as improved scalability of large programs by optimizing away bloat at run time. In addition, it is possible to generalize these approaches to handle similar problems in programs written in other object-oriented languages (e.g., C#), so that non-Java communities would also benefit from them. While these techniques are designed specifically to solve the bloat problem, many of them are applications of more general (theoretical) frameworks developed in the dissertation that can be instantiated to solve other problems.

Bloat contains wasteful operations that, while not strictly necessary for the forward progress, are executed nevertheless. We propose novel dynamic analysis techniques to detect such wasteful operations and to produce information that is necessary for the programmer to pinpoint the performance bottlenecks. One such analysis, as the first contribution of this dissertation, is *copy profiling*. This analysis is designed based on an important observation that the wasteful operations often consist of *data copy activities* that move data among heap locations without any useful computation. By profiling copies, this analysis looks for program regions containing large volumes of copies and data structures whose construction involves data copied frequently from other data structures.

Different from this "from-symptom-to-cause" approach that finds bloat from the symptoms through which it manifests, the second dynamic analysis this dissertation advocates attempts to capture directly the center of bloat, which is the set of operations that, while expensive to execute, produce values of little benefit. Detecting the operations that have high cost-benefit rates is, thus, an important step towards tracking down the causes of performance problems. In order to compute cost and benefit efficiently, we propose a novel technique, called *abstract dynamic thin slicing*, that performs dynamic thin slicing over bounded abstract domains. We demonstrate, using real-world examples, that this technique can also be adopted to solve a range of *backward data flow problems* efficiently. With the help of a variety of data aggregation approaches, these analyses can help a programmer quickly find potential performance problems.

The accumulation of bloat effects may result in memory leaks, which occur when object references that are no longer needed are unnecessarily maintained. Existing dynamic analyses for leak detection track fine-grained information about individual objects, producing results that are hard to interpret and lack precision. The third contribution of the dissertation is a novel container-based heap-tracking technique, based on the observation that many memory leaks in Java programs occur due to

containers that keep references to unused data entries. By profiling containers and understanding their semantics, it is much easier to track down the causes of memory leak problems, compared to existing leak detection approaches based on the tracking of arbitrary objects.

Although container profiling is effective in detecting container-induced memory leaks, it has difficulties dealing with leaks that are caused by general unnecessary references instead of containers. In order to help programmers identify the root causes of these general leaks, we propose a specification-based dynamic technique called *LeakChaser*, as the fourth contribution of this dissertation. LeakChaser brings high-level application semantics into low-level leak detection by allowing programmers to specify and infer object liveness properties. This new technique exploits object lifetime relationships and uses varying levels of abstraction to help both experts and novices quickly explore the leaky behavior to pinpoint the leak cause.

Using these four dynamic analyses, we have found many interesting bloat patterns that can be regularly observed in the execution of large Java programs. A further step to avoid bloat is to develop static analyses that can find and remove such patterns during application development, so that small performance issues can be *prevented* before they accumulate and become significant.

One interesting pattern is the inefficient use of Java containers. The fifth contribution of this dissertation is a static analysis that identifies inefficiencies in the use of containers, regardless of inputs and runs. Specifically, this static analysis detects underutilized and overpopulated containers by employing a context-free-language

(CFL)-reachability formulation of container operations, and by exploiting container-specific properties. The analysis is client-driven and demand-driven. It always generates highly-precise reports, but trades soundness for scalability. We show that this analysis exhibits small false positive rates, and large optimization opportunities can be found by inspecting the generated reports.

Another bloat pattern that we have regularly observed is constructing and initializing data structures that are invariant across loop iterations. As the sixth contribution of this dissertation, we develop a static analysis that uses a type and effect system to help programmers find such loop-invariant data structures. Instead of transforming a program to hoist these data structures automatically (which must be over-conservative and thus becomes ineffective in practice), we advocate a semi-automated approach that uses the static analysis to compute *hoistability measurements*. These measurements indicate how likely it is that these data structures can be hoisted, and are presented to the user for manual inspection. Our experimental results indicate that the proposed technique can be useful both in the development phase (for finding small performance issues before they accumulate) and in performance tuning (for identifying significant performance bottlenecks).

In conclusion, despite the existence of a number of profiling tools, this dissertation attempts to establish more systematic ways of identifying run-time inefficiencies. The dynamic analyses presented in the dissertation are implemented in J9 (a commercial JVM developed by IBM) and Jikes RVM (an open-source JVM written in Java). The static analyses are implemented in Soot, a popular Java program analysis framework. All of the proposed analyses have been shown to scale to real-world Java applications. Our experimental results strongly suggest that these techniques can be used in practice to find and remove bloat in large-scale applications, leading to performance gains. We hope that with the help of the analyses we have developed, performance tuning could be made much easier and will no longer be a daunting task that requires special skills and experience. Developers should be able to easily understand performance and perform optimizations, when they are assisted by good tools and do not need to focus on every low-level detail of the execution behavior and the analysis process. The productivity-performance gap between managed languages and unmanaged languages could be further reduced by using these techniques and tools so that performance would no longer be an issue that stands in the way of using object-oriented languages to implement performance-critical systems. Furthermore, we hope that the examples and patterns discovered by this dissertation can be used to raise the awareness of bloat in real-world software development—developers should understand the performance impact of their decisions and should try to avoid these bloat patterns in order to have high-performance implementations.

To my grandfather Jiming Xu and my wife Jingyuan Yang

ACKNOWLEDGMENTS

I would like to start by giving my deepest thanks to my Ph.D. advisor Nasko Rountev, who has spent a tremendous amount of time and energy forming me into both a good researcher and a nice person. His unwavering patience and optimism have been a constant source of encouragement that helped me overcome numerous challenges and difficulties, and keep pushing my projects forward towards seemingly impossible goals. During the past six years, Nasko offered me the largest possible freedom that one can get in graduate school. I was able to do almost whatever I would like to do—pursuing my own research interests, working with other groups, doing internships, attending classes that have nothing to do with my projects, etc. His dedicated mentorship made my six-year graduate school experience a fulfilling and satisfying one.

The work described in this dissertation has benefited from collaboration with a number of researchers from IBM Research. Nick Mitchell and Gary Sevitsky are the ones who coined the term “runtime bloat” after many years’ study of performance issues in real-world large-scale applications. They are my main motivation to explore the bloat problem. Nick also did a case study and some empirical comparisons for the copy profiling project discussed in Chapter 3. Matthew Arnold mentored me during my first internship with IBM and provided me with tremendous support for understanding and modifying the J9 virtual machine. Manu Sridharan invested a great amount of time into the alias analysis project. Michael Hind wrote a reference letter to support my job application. Erik Altman and Edith Schonberg, my group managers, made a lot of effort to keep me on the project after my summer internships ended, allowing me to have access to IBM internal resources when I was in school. All in all, this dissertation would not have been possible without the two summer internships at IBM, not to mention that much of my education and research at Ohio State was supported through a Ph.D. fellowship and other funds from IBM Research.

Many people at Ohio State made contributions to the content of this dissertation. I need to thank Feng Qin and Michael Bond for serving on my dissertation committee. Feng is always willing to help when I need to deal with large-scale systems code. Through our regular meetings and discussions, I learned many interesting ways that systems researchers use to solve programming problems. Mike has been a constant support for both modifying Jikes RVM and answering my job interview-related questions. I especially want to thank Dacong Yan, a graduate student in my research group, who implemented the frequency profiler for the loop-invariant data structure hoisting project and helped me with a lot of things when I was working from home in California during my last year.

I thank God for showing himself to me and guiding me throughout my life. This work wouldn't have been possible without the amazing support, encouragement, and love from my dear wife Jingyuan, my parents, and my grand-parents. Jingyuan is absolutely my biggest support during my six years' Ph.D. studies: she took care of most of the family duties so that I can have large chunks of time to work on research projects. My grandfather Jiming Xu was my first teacher in my life who spent his last

twenty years teaching me love and care, and how to be a better person. I dedicate this dissertation to them.

The material presented in this dissertation is based upon work supported by the National Science Foundation under CAREER grant CCF-0546040 and grant CCF-1017204, by an IBM Ph.D. Fellowship Award, and by an IBM Software Quality Innovation Faculty Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VITA

September 2010 – August 2011	IBM Ph.D. Fellow at The Ohio State University
June 2010	M.S. Computer Science, The Ohio State University
September 2006 – August 2010	Graduate Research Associate, The Ohio State University
September 2005 – August 2006	University Graduate Fellow, The Ohio State University
March 2005	M.S. Computer Science, East China Normal University
June 2002	B.S. Computer Science, East China Normal University

PUBLICATIONS

Research Publications

Guoqing Xu, Dacong Yan, and Atanas Rountev. Finding Loop-Invariant Data Structures. In *Technical Report OSU-CISRC-8/11-TR24, CSE/OSU*, August 2011.

Dacong Yan, Guoqing Xu, and Atanas Rountev. Context-Sensitive Demand-Driven Analysis for Java. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'11)*, pages 155-165, July 2011.

Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*, pages 270-282, June 2011.

Raffi Khatchadourian, Phil Greenwood, Awais Rashid, and Guoqing Xu. Pointcut Rejuvenation: Recovering Pointcut Expressions in Evolving Aspect-Oriented Software. In *IEEE Transactions on Software Engineering (TSE)*, to appear, 2011.

Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Software Bloat Analysis: Finding, Removing, and Preventing Performance Problems in Modern Large-Scale Object-Oriented Applications. In *ACM SIGSOFT FSE/SDP Workshop on the Future of Software Engineering Research (FoSER'10)*, pages 421-425, November 2010.

Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding Low-Utility Data Structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*, pages 174-186, June 2010.

Guoqing Xu and Atanas Rountev. Detecting Inefficiently-Used Containers to Avoid Bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*, pages 160-173, June 2010.

Raffi Khatchadourian, Phil Greenwood, Awais Rashid, and Guoqing Xu. Pointcut Rejuvenation: Recovering Pointcut Expressions in Evolving Aspect-Oriented Software. In *IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*, short paper, pages 575-579, November 2009.

Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *European Conference on Object-Oriented Programming (ECOOP'09)*, pages 98-122, July 2009.

Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Go with the Flow: Profiling Copies to Find Runtime Bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, pages 419-430, June 2009.

Guoqing Xu and Atanas Rountev. Merging Equivalent Contexts for Scalable Heap-Cloning-Based Context-Sensitive Points-to Analysis. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08)*, pages 225-236, July 2008.

Guoqing Xu and Atanas Rountev. Precise Memory Leak Detection for Java Software Using Container Profiling. In *ACM SIGSOFT/IEEE International Conference* *on Software Engineering (ICSE'08)*, pages 151-160, May 2008. **ACM SIGSOFT Distinguished Paper Award**.

Guoqing Xu and Atanas Rountev. AJANA: A General Framework for Source-Code-Level Interprocedural Dataflow Analysis of AspectJ Software. In *International Conference on Aspect-Oriented Software Development (AOSD'08)*, pages 36-47, March 2008.

Atanas Rountev, Mariana Sharp, and Guoqing Xu. IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries. In *International Conference on Compiler Construction (CC'08)*, pages 53-68, April 2008.

Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. Efficient Checkpointing of Java Software Using Context-Sensitive Capture and Replay. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'07)*, pages 85-94, September 2007.

Guoqing Xu and Atanas Rountev. Regression Test Selection for AspectJ Software. In *ACM SIGSOFT/IEEE International Conference on Software Engineering (ICSE'07)*, pages 65-74, May 2007.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Programming Languages and Compilers	Prof. Atanas Rountev
Software Systems	Prof. Feng Qin
Distributed Computing	Prof. Gagan Agrawal

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: Introduction

A major and sustained technology trend in the past few decades is the proliferation of software designed to solve problems with increasing levels of complexity. These computer programs range from stand-alone desktop applications developed to meet our individual needs and interests, to large-scale long-running servers that can process requests for many millions of concurrent users. Thanks to the advancement of object-oriented programming languages and the standardization of software management processes, the difficulty of developing and maintaining modern programs keep decreasing. Much of these achievements is due to the community-wide recognition of *abstraction* and *reuse*: software should be designed in a modular way so that specifications and implementations are well separated, functional components communicate with each other only through interfaces, and component interfaces are declared as general as possible in order to provide services in a variety of different contexts. The concretization of these ideas has led to an explosion of patterns [42, 53, 73, 78], models [25, 151], and development methodologies [2, 52], which have been considered as the most important software development guidelines that every programmer should know.

However, while software reuse makes development tasks easier, it often comes with certain kinds of excess, leading to performance degradation. Implementation details are hidden from the external users, who have to (and are encouraged to) rely on general-purpose APIs to fulfill their specific requests. When their needs are much narrower than the service that these APIs can provide, wasteful operations start emerging. For example, a study from [92] shows that the conversion of a single date field from a SOAP data source to a Java object can require more than 200 method calls and the generation of 70 objects. Programmers are not aware of the potential performance problems, because they believe the runtime system (e.g., the just-in-time (JIT) compiler) can eliminate these inefficiencies behind the scenes. In addition, the pervasive use of managed languages such as Java and C# makes it possible for programmers to take their attention away from issues of memory usage, because whenever an object becomes unreachable, it is reclaimed by the garbage collector. Programmers are thus encouraged to create objects, taking it for granted that creating and reclaiming objects is free. In a large program that is typically built on top of many layers of frameworks and libraries, a small set of inefficiencies can multiply and quickly get magnified to cause significant slowdown of the system. When the call stack grows to be deep, the usefulness of the compiler analyses in a runtime system becomes limited and the optimizer can no longer remove these inefficiencies.

In this dissertation, the term *bloat* is used to refer to the general phenomenon of using excessive work and memory to achieve seemingly simple tasks. Bloat commonly exists in large-scale object-oriented applications, and impacts significantly their performance and scalability. A program that suffers severe bloat such as a memory leak can crash due to OutOfMemory errors. In most cases, excessive memory consumption and significant slowdown may be seen in programs that contain bloated operations. It is important to find and remove these wasteful operations in large-scale and long-running applications. This need is motivated not only by the desire to reduce the running time and memory usage of these applications. Software maintainers often interpret the growing amount of inefficiency as a result of lack of support from the hardware side, and intend to perform expensive system upgrades (e.g., memory increase, replacement of CPUs with more cores, use of higher-speed network devices, etc.) to help an application reach the expected performance line, which could have been achieved by (more) efficient use of memory and (more) careful choice of operations. For example, as revealed by a recent study [6] conducted at IBM Research, fixing memory and execution inefficiencies (without any system upgrade) has increased the throughput of a large document management server more than twice as much as the alternative that tackles the problem by improving both the quality and the quantity of hardware. Hence, the elimination of bloat can lead to reduced requirements for the execution environment, resulting in significant savings in power and cost. Removing bloat is especially relevant as multicore systems get popular: the excess that exists in memory consumption and execution becomes increasingly painful because memory bandwidth per core goes down, and we cannot rely on speed increases to ameliorate ever-increasing levels of inefficiency.

1.1 Challenges

The JIT's difficulty Bloat cannot be easily removed. Modern JITs have sophisticated optimizers that offer important performance improvements, but they are often unable to remove the penalty of bloat. One problem is that the code in large applications is relatively free of hot spots. Table 1.1 shows a breakdown of the top ten methods from a commercial document management server. This application executes over 60,000 methods, with no single method contributing more than 3.19%

method	*CPU time*
`HashMap.get`	3.19%
`Id.isId`	2.84%
`String.regionMatches`	2.12%
`CharToByteUTF8.convert`	2.04%
`String.hashCode`	1.77%
`String.charAt`	1.70%
`SimpleCharStream.<init>`	1.65%
`ThreadLocalMap.get`	1.32%
`String.toUpperCase`	1.30%

Table 1.1: In a commercial document management server, no single frequently-executed method can be optimized for easy performance gains.

to total application time, and only 14 methods contributing more than 1%. JITs are faced with a number of important methods, and have to rely heavily on the object inliner to combine together code into larger, hopefully optimizable, regions. Forming perfect code regions, and then optimizing them, is an immensely challenging problem [124]. Method inlining is determined based on control flow profiling, and it is *not* necessary for the frequently-executed regions to contain large optimization opportunities, which are, in many cases, related to data creation and propagation (e.g., non-escaping objects). In addition, optimizations that can be easily performed by a programmer (e.g., moving an invocation of a side-effect-free method out of a loop) can require orders of magnitude greater JIT effort to achieve the same effect. That call may ultimately perform thousands of method invocations with call stacks hundreds deep, and allocate many objects. Automatically performing such a transformation requires a number of powerful analyses to work together. If language features that restrict optimization (e.g., reflection and precise exceptions) are taken into account, there is little hope that a performance bottleneck can be identified and removed by a fully automated solution. As an example, a study described later found that to perform the seemingly simple task of inserting a single small document in the database, the document server invokes 25,000 methods and creates 3000 temporary objects, even after JIT optimizations. However, with less than one person-week of manual tuning, a performance expert was able to reduce the object creation rate by 66%. Such results indicate that vast improvements are possible when tuning is made easier with more powerful tool support.

The human experts' difficulty While with manual tuning a skillful performance expert may find larger optimization opportunities, it is not easy as well for her to perform the tuning task, primarily due to the lack of good tools. Despite the existence of many profilers [51, 80, 109], the information that they report is far from being sufficient for a programmer to locate the performance bottleneck. For example, in most cases, JProfiler can report the numbers of instances of different types during the execution. However, the fact that type java.util.HashMap$Entry has the highest number of instances tells the programmer nothing about the HashMaps that hold these entries and the calling contexts under which the elements are added to the HashMaps. In addition to the limitation with the existing tool technology, tuning requires deep understanding of implementation logic and a great deal of programming experience. Thus, there are often only a handful of programmers who are capable of manually performing this task, which can involve many (iterative) cycles of program inspection, cost measurement and analysis, configuration change, and source code modification, and can be extremely time-consuming.

1.2 Compiler-Assisted Manual Tuning: Bloat Detection via Program Analysis

Program analysis targeting performance problems The goal of this dissertation is to develop compiler and tool support that can help a programmer detect and remove bloat throughout the entire software development process. Our objective is not to replace JIT optimizations, but to complement them, thus making it easier to perform tuning that targets runtime bloat. Through a combination of metrics and analyses focused on bloat, we hope to quickly guide developers to the problematic areas of the application, allowing them to transform the program to eliminate the problem. The techniques we have designed range from static analyses that find wasteful operations in the early stage of development, to dynamic analyses that detect bloat based on various kinds of heuristics and symptoms being observed during the execution. The static analyses can be easily incorporated in existing coding assistance tools to give programmers warnings whenever inefficiency is seen, and to transform the program to avoid the potential bloat. The dynamic analyses can be either implemented in offline diagnosis tools, or embedded in Java Virtual Machines (JVM) to offer both coarse-grained and fine-grained problem diagnosis with varying time (i.e., 50% through 70×) and space (i.e., up to 2×) overheads.

1.2.1 Dynamic Analyses That Detect Bloat by Making Sense of Heaps and Executions

One of the most difficult tasks during performance tuning is making sense of the heap, with its large number of objects, and of the execution that may have millions of method invocations and last for hours and days. We have developed four dynamic analysis techniques that exploit activity profiling and garbage collection runs to expose bloat symptoms. The essence of these techniques is to highlight performance-impacting activities in programs. Understanding these activities makes it possible for programmers to make more sense of execution behavior and heap usage, so that they can quickly recognize the problems and perform appropriate design and/or implementation modifications to overcome them.

Existing compilers and tools focus on control flow (e.g., method execution times and frequencies) when making optimization decisions. However, in large-scale systems, data-based activities are sometimes stronger signs of excess than control-based ones. Examples of such symptoms include excessive data copying, operations of high cost and little benefit, problematic container behaviors, and stale objects (that are reachable in the object graph but not used for a long time). The first part of this dissertation presents four dynamic analyses that identify inefficiencies by focusing on such interesting data-based activities.

Profiling copies to make sense of data flow Bloat often occurs due to unnecessary work along data flows. Such work may consist of *data copy activities* that move data among heap locations without any useful computation. By optimizing copies, one is likely to remove the objects that carry copied values, and the method calls that allocate and populate these objects. We design a dynamic analysis (described in Chapter 3) [156] that looks for program regions containing large volumes of copies, and data structures whose construction involves data copied frequently from other data structures. By detecting frequently-occurring copy chains and computing copy volumes for data structures, the analysis can quickly guide developers to the problematic areas of the application.

Abstract dynamic slicing to make sense of cost and benefit Object-oriented programming encourages the creation of interfaces with *general* functionalities. While these interfaces can be used in different contexts, it is often the case that the service they provide is more general than what a specific client needs, leading to wasteful operations and runtime bloat. A common effect of these operations is that they produce values that could have been obtained in much easier ways. It is possible for a programmer to quickly detect performance bottlenecks, if she can understand the cost of producing a certain value and the benefit of consuming it. We propose a run-time cost-benefit analysis (described in Chapter 4) [155] that can help a programmer understand accurately the performance of her program and find the reasons for performance problems seen during program execution. The cost and benefit computation is based on a novel technique called *abstract dynamic thin slicing*, that applies dynamic slicing over bounded abstract domains, leading to a tunable analysis framework with adjustable abstractions and memory usage. This technique can serve as the foundation for efficiently solving a set of *backward dynamic flow* problems that exhibit bounded-domain properties.

Profiling container operations to make sense of (unnecessary) reference holding An important source of bloat is the inefficient use of containers. For example, failure to remove objects from containers can lead to a memory leak, resulting in performance degradation and a program crash. We develop a dynamic analysis (described in Chapter 5) [158] to detect memory leaks by profiling container operations, based on the observation that many memory leaks in Java programs occur due to containers that keep references to unused data entries. Compared to existing dynamic leak detection techniques that are based on heap differencing and fine-grained object tracking, container profiling has fewer false positives and is capable of quickly directing the programmer to the problematic container that is exactly the cause of the leak.

Checking and inferring object lifetime invariants to make sense of object lifetime relationships While the container profiling technique can precisely detect container-related memory leaks, many real-world leak problems are not caused by containers, but by simple unnecessary references that programmers forget to invalidate. To help programmers narrow down the causes of these leaks, we design a run-time analysis that exploits invariants on object lifetimes.

For many objects, implicit invariants may exist among their lifetimes, such as "object a references object b and when a dies, b will also die". These invariants hold for most of the execution and if, occasionally, they are violated (e.g., a is garbage collected but b is not, because there exists an unnecessary reference from another object to b), a memory leak may occur. We develop a multi-level dynamic technique called LeakChaser (described in Chapter 6) [157] that exploits such (lifetime) relationships to detect Java memory leaks. At the lowest level of this approach is a small set of basic assertions (such as *diesBefore*(a, b)) that can be employed by users to specify these relationships. These assertions are checked regularly at garbage collection runs. We then introduce high-level application semantics into the analysis by considering transactions, which are periodically-occurring events during the execution. Objects that are supposed to be specific to a particular transaction (i.e., they should be transaction-local), but instead are observed by the analysis to be shared among transactions, are flagged as potentially leaking. At the highest-level of the approach is an inference algorithm. The user only needs to tell the tool an object which defines the duration of the transaction. The algorithm automatically infers the corresponding transaction-local and shared objects based on their lifetimes, in order to detect memory leaks. This approach provides users at different levels of skills and code familiarity with sufficient flexibility to specify memory leak properties, and generates reports containing adequate semantic information to facilitate understanding.

Summary Our dynamic analysis work addresses the challenge of automatically detecting bloated computations that fall out of the purview of conventional JIT optimization strategies. In general, existing bloat detection work can be classified into two major categories: manual tuning methods (i.e., mostly based on measurements of bloat) [49, 50, 96, 97], and fully automated performance optimization techniques such as the entire field of JIT technology [8] and the research from [124]. The work described in this dissertation sits in between: we provide sophisticated analyses to support manual tuning, guiding programmers to the program regions where bloat is most likely to exist, and then allowing human experts to perform the code modification and refactoring. By doing so, we hope to help the programmers quickly get through the hardest part of the tuning process—finding the likely bloated regions—and yet use their (human) insight to perform application-specific optimizations.

1.2.2 Static Analyses That Remove and Prevent Bloat

Using the dynamic analyses, we have identified a number of common bloat patterns that are regularly seen during the executions of large Java applications. We have developed static analysis tools that can identify these patterns early during development, in order to allow a programmer to inspect the code to eliminate them before program execution.

Detecting inefficiently-used containers to avoid bloat We found that a significant source of runtime bloat is the inefficient use of containers. To find and prevent container inefficiencies early in development, we develop a static analysis (described in Chapter 7) [159] that can identify two specific types of container inefficiencies, namely, underutilized container and overpopulated container. This work proposes the *first static analysis* to identify bloat: this analysis automatically extracts container semantics and does not require any user annotations. We focus on two abstract container operations ADD and GET, and the analysis detects heap stores and loads that concretize them based on the context-free-language (CFL) reachability formulation of points-to analysis. The second step of the analysis is to approximate the frequencies of the identified stores/loads to/from heap locations based on the nesting relationships among the loops where they are located. If the number of ADD operations is very small, the container is underutilized. If the number of ADD operations is significantly larger than that number of GET operations, the containers is overpopulated because many elements are not retrieved at all. Reports generated from this analysis have been shown to be more precise than those produced by the corresponding dynamic analysis.

Hoisting invariant loop data structures to remove bloat Another important bloat pattern that we have regularly observed is the construction and initialization of data structures that are invariant across loop iterations. We develop a static analysis (described in Chapter 8) [161] that can help programmers find such data structures. The analysis focuses on logical data structures that may be hoisted. Specifically, we design a novel type and effect system that identifies, for each data structure created in the loop, its hoistable part. The larger this part is, the more likely this data structure can be hoisted. In order to quantify this likelihood, our analysis computes *hoistability measurements*. These measurements are eventually presented to the user for further manual inspection. Our experimental results indicate that the proposed technique can be useful both in the development phase for finding small performance problems, and in the tuning phase for identifying significant performance bottlenecks.

Summary The static analysis work presented in this dissertation is the first attempt to detect performance problems using static compiler-based approaches. All existing techniques targeting bloat are dynamic approaches that try to locate performance problems by identifying suspicious run-time bloat-indicating behaviors. Our insight is that some of these behaviors are the dynamic reflection of certain static properties inherent in the source code of the program, which can be easily seen at compile time. By finding these properties, a static analysis can also identify potential performance issues, despite the lack of run-time information such as method execution frequencies and object allocation counts.

1.3 Impact

Our techniques are based on novel insights into programming languages and on empirical studies of large-scale and long-running applications, and thus, can be more precise in detecting bloat and more effective at removing inefficiencies than prior approaches. The detection and diagnosis techniques rely on observations of different bloat effects, and novel analysis algorithms to perform both light-weight and heavy-weight profiling efficiently. At the center of these bloat effects are suspicious data-based activities. They are better indicators of bloat than what has been traditionally thought of as performance-related properties, such as execution time and numbers of objects created. In addition, checking and inferring object liveness properties can bridge the gap between performance problems and traditional specification-based research. This may open up new research directions where the existing body of software verification and validation techniques can be employed to find memory bloat effects, which are non-functional and essentially caused by inefficient use of memory.

The static analysis work presented in the dissertation is the first attempt to understand performance issues related to runtime bloat in object-oriented languages with the help of static approaches. The success of these analyses has demonstrated that it is possible to identify certain kinds of bloat at compile time, thus providing new insights for future development of static bloat detectors and compiler optimizations.

The work in this dissertation has immediate practical impact because the static analyses can be incorporated in existing coding-assistance tools (e.g., Eclipse) to find inefficiencies during software development, while the dynamic analyses can be embedded as a JVM service that can be enabled when problem diagnosis is needed.

The broader impact of this work is that it sheds new light on performance optimization of large applications, and demonstrates the viability of quickly finding and removing performance bottlenecks with the support of systematic techniques that can make more sense of the execution, the heap and resource usage, and the behavior of containers. This attention to typical patterns of inefficiencies (e.g., copies and loop-invariant data structures) can also help with benchmark design and future improvements in code optimization technology. Benchmark designers can use metrics that target certain bloat patterns to ensure that the benchmarks exhibit the categories of bloat that occur in real-world applications. If benchmark suites contain such bloat measurements, researchers could use these metrics to evaluate any proposed bloat detection/optimization techniques. In addition, future compiler design can take into account these frequently-observed bloat patterns, and can include optimization techniques that work specifically for each such pattern.

CHAPTER 2: Analysis Frameworks

This chapter presents infrastructures that we have implemented for our dynamic and static analyses. An important component required by many dynamic analysis techniques described in this dissertation is a framework that supports whole-program information flow tracking. This chapter first introduces one general information flow tracking framework we have developed. It is used in both the copy profiler (Chapter 3) [156] and the cost-benefit analysis (Chapter 4) [159]. Second, we present the platforms and environments used to implement the dynamic [155–158] and static analyses [159, 161]. Finally, the chapter presents the benchmarks and platforms used in the experimental studies.

2.1 Dynamic Analysis Implementation

We implemented the copy profiling (discussed in Chapter 3) and the cost-benefit analysis (discussed in Chapter 4) in J9 (build 2.4, J2RE 1.5.0 for Linux x86-32), a commercial Java Virtual Machine developed by IBM [144]. This JVM is highly versatile and is used as the basis for many of IBM's product offerings from embedded devices to enterprise solutions. By piggy-backing the analyses on J9's JIT compiler, we are able to apply and evaluate the developed techniques on large and long-running Java programs such as database servers, JSP/servlet containers, and application servers. This makes it possible to find problems in these real-world applications, which are widely used and have significant impact on the software industry. While implemented in J9, these analyses are general enough to be migrated to other JVMs and runtime systems such as Microsoft's common language runtime (CLR) [143]. The analyses were built as JVM services. Enabling/disabling them can be controlled by specific command line options (e.g., -Xjit:...) used to invoke the JVM. We ran the JVM in the JIT-only mode to avoid the effort to modify the interpreter, whose source code is not available to us.

The LeakChaser memory leak detector (described in Chapter 6) was implemented in Jikes RVM 3.1.0, an open source Java-in-Java Virtual Machine. We piggy-pack the analysis on its garbage collector (GC), and check the specified liveness properties at the end of each GC run. Similarly to other dynamic analyses presented in this dissertation, the LeakChaser tracking and analysis were implemented as a JVM service that can be switched on/off by using different Jikes RVM command line options.

The implementation of the container-based memory leak detector (Chapter 5) exploits the common JVM Tool Interface (JVMTI) [145], which is supported by all commercial JVMs. The reason why we did not choose to implement this technique in a JVM is that it does not rely on any VM-internal information and the event callbacks provided by JVMTI are sufficient to perform the needed profiling.

2.2 Dynamic Information Flow Infrastructure

As a key component of our dynamic analysis framework (used in [155, 156] and implemented in J9), this infrastructure tracks the flow of data and performs the predefined (side-effect-free) operations as instructions are executed. Each piece of data used during the execution is associated with a piece of *tracking data*, which is updated when its corresponding execution data is updated. Tracking data is contained

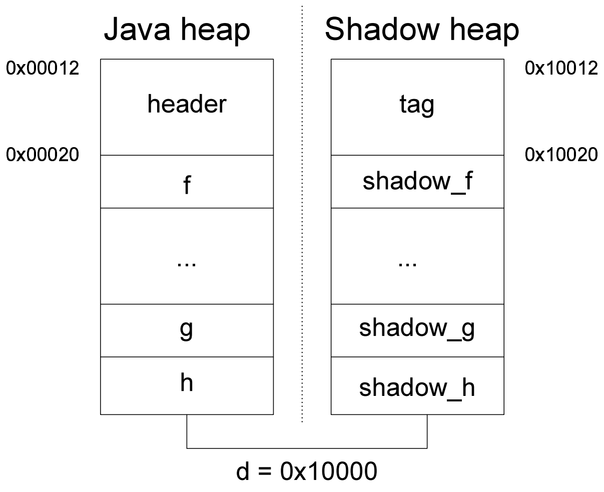


Figure 2.1: Representation of an object in the Java heap and its shadow memory. `f`, `g`, and `h` represent the fields in this object. `d` represents a constant offset.

in a shadow memory, which is separate from the memory space created for the execution. Our information flow infrastructure supports shadowing of all memory in the application, including local variables, static fields, arrays, and instance fields.

Shadow variable A local variable is shadowed simply by introducing an extra variable of the same type on the stack.

Shadow heap Shadowing of static fields, arrays and instance fields is supported by the use of a *shadow heap* [102]. The shadow heap is a contiguous region of memory equal in size to the Java heap. To allow quick access to shadow information, there is a constant distance between the shadow heap and the Java heap. Thus, the scratch space for every byte of data in the Java heap can be referenced by adding this constant offset to its address. The size of the tracking data equals the size of its corresponding data in the Java heap. While the creation of the shadow heap introduces 2× space overhead, it has not limited us from collecting data from large-scale and long-running applications such as the aforementioned document management server (built on top of the WebSphere application server and DB2 database server). In fact, with 1GB Java heap and 1GB shadow heap, we were able to successfully run all programs we encountered, including large production web server applications.

Tagging objects In addition to the tracking data, the infrastructures supports object tagging. For example, we often need to tag each run-time object with its allocation site in order to relate the run-time behavior of the object to the source code that creates it. In other cases, calling contexts are also associated with objects to enable context-sensitive diagnosis [20,21,155,156]. One way of implementing object tagging is to store the associated information into the object headers. For example, for some JVMs such as Jikes RVM, each object header has a few free bytes that are not used by the VM, and these free bytes can be employed to store the tag. However, in J9, most bytes in an object header are used by the garbage collector, and modifying them can crash the JVM at run time. Our infrastructure (implemented in J9) stores the tag into the shadow heap. The shadow space corresponding to an object header contains the tag for the object. Because an object header in J9 takes two words (i.e., 8 bytes), the infrastructure allows us to tag objects with up to 8 bytes data; this space is much larger than the free bytes in the object header. A representation of the Java heap and shadow heap is shown in Figure 2.1, where the constant offset between the two heaps is 0x10000.

Tracking stack The infrastructure also supports the passing of tracking data interprocedurally through parameters and return values. This is achieved by the use of a *tracking stack*, which is similar to the Java call stack. Tracking data for the actual parameters is pushed onto the stack at a call site, and is popped at the entry of the callee that the call site invokes. Similarly, tracking data for the return variable is pushed onto the stack at the return site, and is popped immediately after the call returns. The tracking of exceptional data flow is not supported in our framework, because it usually does not carry important data across method invocations.

Flow transfer function and framework This infrastructure is designed in a way so that it can be easily parameterized and reused for a wide variety of dynamic analyses that require information flow tracking, such as taint analysis [38,59,103,108], null origin tracking [24], web security analysis [60], and information flow strength measurement [88]. Such reuse requires the definition of flow transfer function for each instruction, which can be given by implementing the callback interfaces defined by the infrastructure. There are two types of callbacks: compile-time inlining callback and run-time invocation callback. Functions implementing compile-time inlining callback interfaces are actually never invoked. Instead, code in them is "inlined" into the executing program when the JIT compiles the program. This type of interface is suitable for simple instrumentation that needs to perform only Java operations and does not rely on internal VM support. On the other hand, functions implementing run-time invocation callback interfaces are invoked at run time. Our framework only inserts calls to these functions at appropriate positions during compilation. This type of interface is suitable for heavy-weight instrumentation that usually maintains data structures external to the Java program, performs non-Java operations such as storing pointer values, and requires special JVM supports such as sampling and object graph traversal during garbage collection. Of course, larger run-time overhead can be incurred by this type of instrumentation.

The user also needs to provide a "report" function, and a parameter that tells the infrastructure when the report should be generated (e.g., the JVM shuts down, a garbage collection run occurs, or a sampling period completes). The framework automatically invokes the function when the specified event is triggered, to analyze the profiled data and generate a report.

2.3 Static Analysis Implementations

Our static analysis and transformation techniques are implemented in Soot [132, 147] (version 2.3.0), a popular program analysis framework for Java. The analyses make use of the Spark component [83] in Soot to obtain context-insensitive points-to information. The work on inefficiently-used container detection [159] also uses the Sridhran-Bodik framework [134] implemented in Soot to solve the context-free-language (CFL)-reachability formulation of container behaviors. The detailed description of CFL-reachability and the formulation can be found in Chapter 7.

2.4 Benchmarks and Execution Platforms

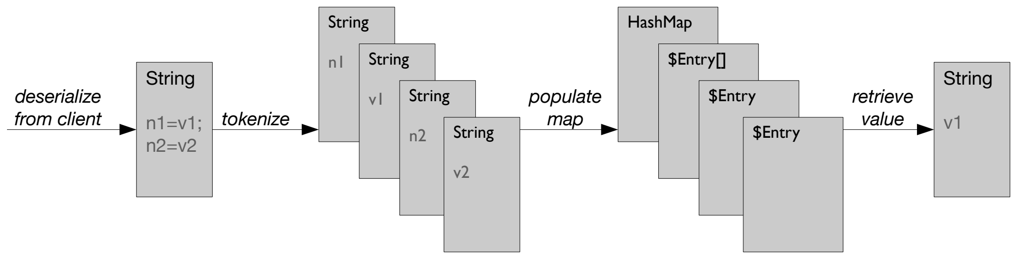
A common set of benchmarks that we use to evaluate both our static and dynamic analyses is DaCapo [15,17,40] (version 2006-10-MR1), which contains 11 open source, real-world applications with non-trivial memory loads. In addition to this set of programs, the work of copy-profiling [156] is evaluated using the IBM document management server and the SPECJbb 2000 benchmark. For the cost-benefit analysis [155], we also considered the additional 7 programs included in a new version of DaCapo as of October 2009, which are typically widely-used server applications running on top of layers of frameworks and libraries. The benchmark set for static analyses additionally includes programs from SPECjvm98 [139] and Ashes [10], and some other well-known Java applications such as `muffin` (a WWW filtering system) and `polyglot` (an extensible compiler framework).

Experiments for copy profiling [156] and cost-benefit analysis [155] were undertaken on a machine with a 1.99GHZ Dual Core AMD Opteron processor, running Linux 2.6.9. The programs were run with JIT optimizations turned off to collect data from the unmodified source programs. The work on LeakChaser [157], container profiling [158], and all the static analyses [159,161] were evaluated on a machine with an Intel Xeon 2.80GHz processor, running Linux 2.6.9.

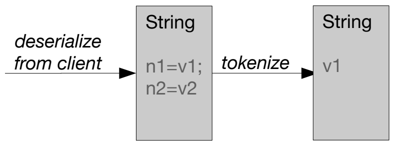
CHAPTER 3: Making Sense of Copy Chains

One important symptom of runtime bloat is the existence of large volumes of copies in bloated regions: data is transferred from one object to another, with no computation done on it. Long copy chains can often be seen, simply to form objects of certain types required by APIs of one framework from their original representations used in another framework. The aforementioned example (in Chapter 1) of the SOAP protocol clearly illustrates this point: the original objects are wrapped in a way so that they can be passed to the SOAP layer for network transmission, and the SOAP representations are unwrapped on the other side of the network to obtain the data. This entire process does not contain any computation on the data that is transmitted. In this example, while data copying is a necessary step for the network transmission, the observation of large volumes of copies may quickly remind the user to consider the overall design and architecture of the application. Is it really worth the use of SOAP? Is there any cheaper way of transmitting data if the format of the data is simple enough?

If the SOAP example exhibits design issues, the following example that can be observed in the document management server reveals bloat caused by a programmer's mistake. The server extracts name-value pairs from a cookie that the client transmits in a serialized, string form. The methods that use these name-value pairs expect Java objects, not strings. They invoke a library method to decode the cookie string into a Java `HashMap`, yet another transient form of this very simple data. In the common



(a) Original version.



(b) Specialized version.

Figure 3.1: The steps a commercial document management server uses to decode a cookie; the original version tokenizes and returns the entire map, even if the caller needs only one name-value pair.

case, the caller extracts one or two elements from the 8-element map, and never uses that map again. Figure 3.1 illustrates the steps necessary to decode a cookie in this application.[1] Decoding a single cookie, an operation that occurs repeatedly, costs 1000 method invocations and 35 temporary objects, after JIT optimizations. A hand-optimized specialization for the common case that only requires one name-value pair invokes 4 invocations and constructs 2 temporary objects.

The inefficiencies at the heart of the SOAP example and the cookie decoding example are common to many bloated implementations. In these implementations, there is often a chain of information flow that carries values from one storage location to another, often via temporary objects [97]; e.g., as visualized in Figure 3.1. Bloat of this form manifests itself in a number of ways: temporary structures to carry values,

[1]We thank Nick Mitchell for providing this illustration.

and a large number of method invocations that allocate and initialize these structures, and copy data between them.

In our experience, it is this data copying activity that is an excellent indicator of bloat. When copy activities are reduced through code transformations, this often reduces the need for creating and deallocating the corresponding objects, and for invoking methods on these objects. For example, the optimized cookie decoding in Figure 3.1 eliminates the cost of constructing the `HashMap` and the related key-value pairs.

Section 3.1 provides an example of the amount of copy activities in a server application, and shows how they are not handled well by the JIT in a state-of-the-art JVM. This example illustrates the need to track copy operations and to provide a programmer with useful summarization of chains of such operations. For example, consider the cookie decoding in Figure 3.1(b). It is neither the result of tuning the `HashMap` `put` or `get` methods, nor of tuning the `HashMap` data structure. To specialize this scenario requires understanding the *chains* of copies: which storage locations carry values, and which methods contain the copies. During program execution, there will be billions of copy chains. In Section 3.2, we introduce an abstraction, the *copy graph*, that concisely summarizes chains of copies.

3.1 Profiling Copy Activity

A *copy* operation is a pair of a heap load and a heap store instructions that transfers a value, unmodified, from one heap location to another. A *copy profile* counts the number of copies; a copy operation is associated with the method that performed the write to the heap. Although the profiling tracks the propagation

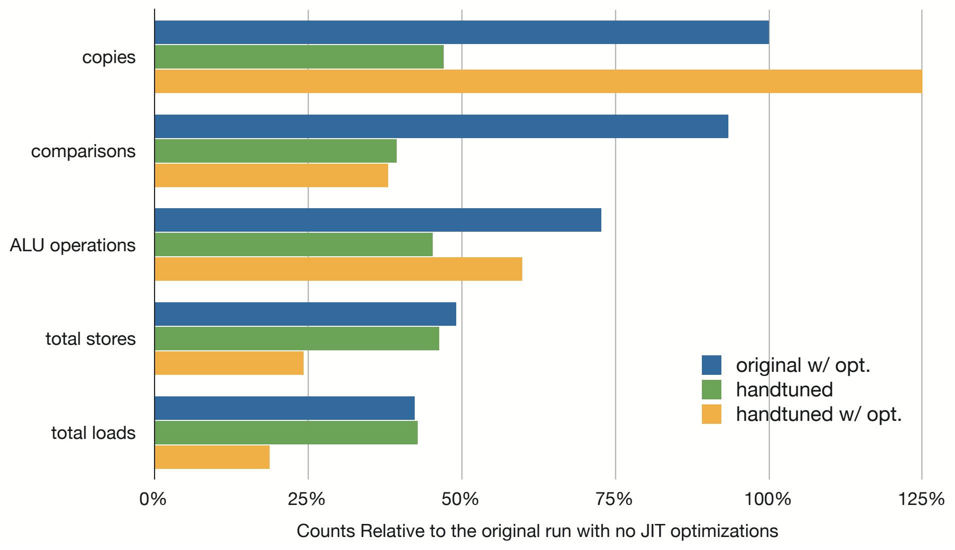


Figure 3.2: A breakdown of activity in a document processing server application. The baseline, at 100%, is the original code run with JIT optimizations disabled. This baseline is compared to the original code with JIT optimizations enabled, and to an implementation with a dozen hand-tunings.

through stack locations (in order to determine whether a store is the second half of a copy), the profiling reports do not include that level of detail. Since stack variables will likely be assigned to registers, chains of copies between stack locations will usually involve only register transfer operations. They are also more likely to be optimized by conventional dataflow analysis.

Figure 3.2 shows a comparison of four scenarios of the document management server, executing in the IBM J9 production JVM. The baseline, at 100%, represents the behavior of the original code, with JIT optimizations disabled, during a 10 minute load run. This baseline is compared to the original code with JIT optimizations enabled, and to a version of the code that had been hand-tuned (both with and without JIT optimizations). The figure also shows the number of comparison operations, the number of ALU operations, and the total number of loads and stores. While the JIT successfully reduces the number of ALU operations and loads/stores, it does not affect significantly the number of copies and comparisons.

3.2 Profiling Copy Chains and Copy Graph

Individual copies are usually part of longer copy chains. Optimizing for bloat requires understanding the chains as a whole, as they may span large code regions that need to be examined and transformed. We now show how to form an abstraction, the copy graph, that can be used to identify chains of copies.

3.2.1 Copy Chain

Definition 3.2.1 *(*Copy chain*). A* copy chain *is a sequence of copies that carry a value through two or more heap storage locations. Each copy chain node is a heap location. Each edge represents a sequence of copies that transfers a value from one heap location to another, abstracting away the intermediate copies via stack locations, parameter passing, and value returns.*

The heap locations of interest are fields of objects and elements of arrays. A copy chain ends if the value it carries is the operand of a computation, which produces a new value, or is an argument to a native method. It is important to note that, in a copy chain, each maximal-length subsequence of stack copies is abstracted by a single edge directly connecting two heap locations.

The code in Figure 3.3 is used for illustration throughout this chapter. The example is based on a common usage scenario of Java collections. A simple implementation of a data structure `List` is used by a client `ListClient`. `ListClient` declares two

```
class List{
   Object[] elems; int count;
   List(){ elems = new Object[1000]; }
   List(List l){ this(); // call default constructor
      for(Iterator it = l.iterator(); it.hasNext();)
         { add(it.next()); } }
   void add(Object m){
      Object[] t = this.elems;
      t[count++] = m;
   }
   Object get(int ind){
      Object[] t = this.elems;
      Object p = t[ind]; return p;
   }
   Iterator iterator(){
      return new ListIterator(this);
   }
}
class ListIterator{
   int pos = 0; List list;
   ListIterator(List l){
      this.list = l;
   }
   boolean hasNext(){ return pos < list.count - 1;}
   Object next(){ return list.get(pos ++);}
}
class ListClient{
   List myList;
   ListClient(List l){ myList = l; }
   ListClient slowClone(){
      List j = new List(myList);
      return new ListClient(j);
   }
   ListClient fastClone(){
      return new ListClient(myList);
   }
}
static void main(String[] args){
   List data1 = new List();
   for(int i = 0; i < 1000; i++)data1.add(new Integer(i));
   List data2 = new List();
   for(int i = 0; i<5; i++){data2.add(new String(args[i]));
      System.out.println(data2.get(i));}
   ListClient c1 = new ListClient(data1);
   ListClient c2 = new ListClient(data2);
   ListClient new_c1 = c1.slowClone();
   ListClient new_c2 = c2.fastClone();
}
```

Figure 3.3: Copy profiling running example.

clone methods `fastClone` and `slowClone`, which return a new `ListClient` object by reusing the old backing list and by copying list elements, respectively. The entry

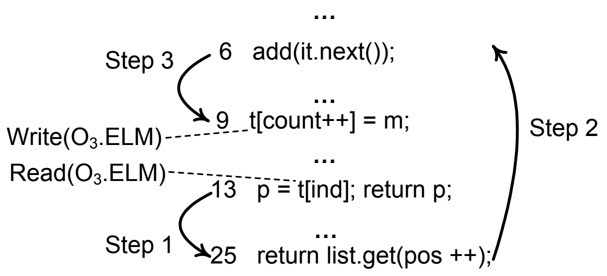


Figure 3.4: A copy chain due to `ListClient.slowClone`. Line numbers 6, 9, 13, and 25 correspond to the code in Figure 3.3.

method `main` creates two lists `data1` and `data2` and initializes them with 1000 Integer and 5 String objects (lines 40 and 42). The two lists are then passed into two `ListClient` objects and eventually two new `ListClient` objects are created by calling `slowClone` and `fastClone`. For simplicity, the approach is described at the level of Java source code, although our implementation works with a lower-level virtual machine intermediate representation (IR).

Figure 3.4 depicts the steps in the creation of a single-edge copy chain. This chain results from the invocation of `slowClone` (line 46) which copies Integer object references from the array referenced by field *elems* of one `List` to the array referenced by field *elems* of another `List`. The source array and the target array will be denoted by O_3 since they are created at line 3 in the code. (For now, the reader can ignore the naming scheme; it will be discussed shortly.) The copy chain in Figure 3.4 is $O_3.ELM \rightarrow O_3.ELM$, where ELM represents any array element.

To represent the source and the sink of the data propagated along a copy chain, we can augment the chain with two nodes: a *producer* node added at the beginning, and a *consumer* node added at the end. The producer node can be a constant value, a `new X` expression, or a computation operation representing the creation of a new value. The consumer node has only one instance (denoted by C) in the copy graph, showing that the data goes to a computation operation or to a native method. These two types of nodes are not heap locations, and are added solely for the purpose of subsequent client analyses. Note that not every chain has these two special nodes. For the producer node, we are interested only in reference-typed values because they are important for further analysis and program understanding. Thus, chains that propagate values of primitive types do not have producer nodes. Not every piece of data goes to a consumer and therefore not every chain has a consumer node. The absence of a consumer is a strong symptom of bloat and can be used to identify performance problems. An example of a full augmented copy chain starting from producer O_{42} (i.e., `new String`) is $O_{42} \rightarrow O_3.ELM \rightarrow C$. This chain ends in consumer node C because the data goes into method `println` which eventually calls native method `write`.

3.2.2 Copy Graph

Profiling copy chains can be extremely space expensive, because it requires maintaining a distinct node for each heap location on each copy chain, regardless of whether chains have shared heap locations. In addition, for each heap location, it is necessary to maintain the history information regarding all chains that go through this location, which may incur significant running time overhead. To make the analysis scale to large applications, we apply a series of abstractions on copy chains. These abstractions are also essential for producing summarized reports that do not overwhelm the tool user with millions of chains. The first abstraction is to merge all copy chains in a *copy graph*, so that nodes shared among chains do not need to be maintained separately. In addition, the copy graph construction algorithm can be designed to profile only graph edges (i.e., one-hop heap copy), which is much more efficient than profiling of entire chains.

Definition 3.2.2 *(*Copy graph*). A copy graph $G = (\mathcal{N}, \mathcal{E})$ has node set $\mathcal{N} \subseteq \mathcal{AL} \cup \mathcal{IF} \cup \mathcal{SF} \cup \{C\}$. Here $\mathcal{AL}$ is the domain of allocation sites O_i which serve as producer nodes and do not have any incoming edges. $\mathcal{IF}$ is the domain of instance field nodes $O_i.f$. $\mathcal{SF}$ is the domain of static field nodes. C is the consumer node; it has only incoming edges. The edge set is $\mathcal{E} \subseteq \mathcal{N} \times Integer \times Integer \times \mathcal{N}$. Each edge is annotated with two integer values: the frequency of the heap copy and the number of copied bytes (i.e., 1, 2, 4, or 8).*

There could be many different ways to map the run-time execution to these abstractions. The rest of this section describes the mapping used in our current work; future work could explore other choices with varying cost, precision, and usefulness for tool users.

Object naming scheme Following an abstraction technique widely adopted in static analysis, an allocation site is used to represent the set of run-time instances that it creates. Similarly, all heap locations that an instance field dereference expression $a.f$ represents are projected to a set of nodes $\{O_i.f\}$ such that the objects that a points to are projected to set $\{O_i\}$. Applying this abstraction reduces the number of allocation site nodes $\mathcal{AL}$ and instance field nodes $\mathcal{IF}$. Each element of an array a is represented by a special field node $O_a.ELM$, where O_a denotes the allocation site of a and ELM represents the field name. Individual array elements are not

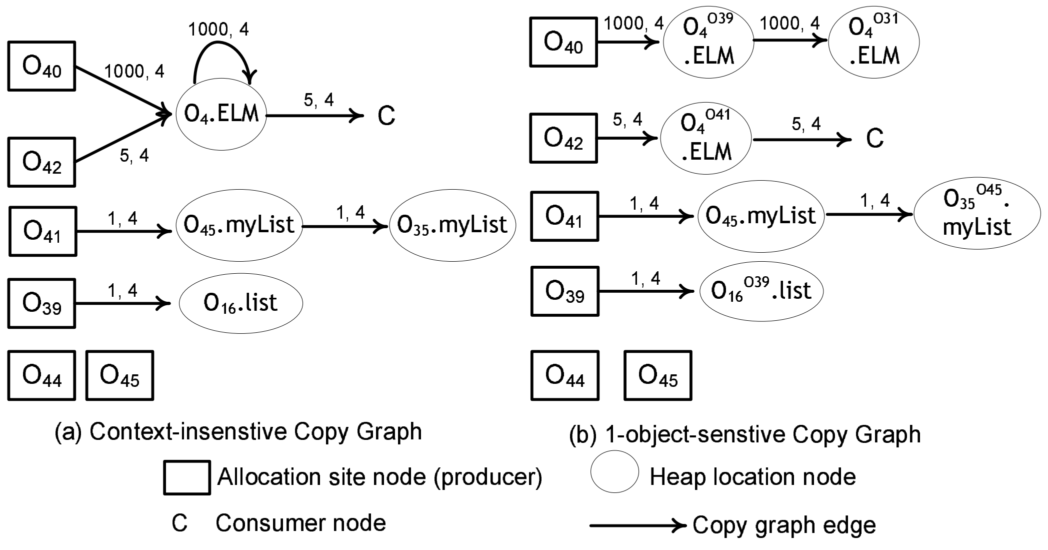


Figure 3.5: Partial copy graph with context-insensitive and context-sensitive object naming.

distinguished: considering each element separately may introduce infeasible time and space overhead.

For illustration, consider the partial copy graph in Figure 3.5(a). The figure shows only paths starting from nodes in method `main` in the running example. An allocation site is named O_i, where i is the number of the code line containing the site. Each copy graph edge is annotated with two numbers: its frequency and the number of bytes it copies. For example, edge $O_{40} \xrightarrow{1000,4} O_3.ELM$ copies the `Integer` objects created at line 40 into the array referenced by `data1`'s `elems` field. This edge consists of a sequence of copies via parameter passing (line 40 and line 9). This sequence of copies occurs 1000 times, and each time 4 bytes of data are transferred. Both $O_{40} \xrightarrow{1000,4} O_3.ELM$ and $O_3.ELM \xrightarrow{1000,4} O_3.ELM$ are hot edges: their frequencies and the total number of bytes copied are much larger than those of other edges.

When there exists a performance problem in the program, a better design might be needed to eliminate these copies.

It is important to note again that nodes that represent different objects may be merged due to the employed abstraction. For example, although variable `t` at line 9 points to different objects at run time, the array element node `t[count++]` is represented by a single node $O_3.ELM$, regardless of the `List` object that owns the array. Consider the self-pointing edge $\xrightarrow{1000,4}$ at node $O_3.ELM$. The edge captures the data flow illustrated in Figure 3.4. This sequence of copies moves object references from the array pointed-to by $O_{39}.elems$ to the array pointed-to by $O_{31}.elems$. Since both arrays are represented by O_3, their elements are merged into $O_3.ELM$ in the copy graph and this self-pointing edge is generated.

Merging of nodes could lead to spurious copy chains that are inferred from the copy graph. For example, from Figure 3.5(a), one could imprecisely conclude that both O_{40} and O_{42} will eventually be consumed, because both edges $\xrightarrow{1000,4}$ and $\xrightarrow{5,4}$ can lead to consumer node C. The cause of the problem is the *context-insensitive* object naming scheme, which maps each run-time object to its allocation site, regardless of the larger data structure in which the object appears. In order to model copy chains more precisely, we introduce a context-sensitive object naming scheme.

3.2.3 Context Sensitivity

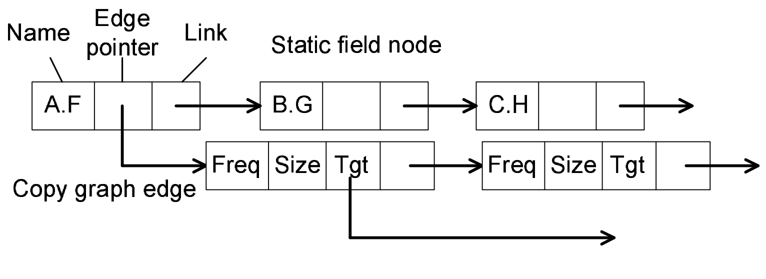
When naming a run-time object, a context-sensitive copy graph construction algorithm takes into account both the allocation site and the calling context of the method in which the object is allocated. Existing static analysis work proposes two major types of context sensitivity for object-oriented programs: call-chain-based context sensitivity (i.e., k-CFA) [125], which considers a sequence of call sites invoking the analyzed method, and object-sensitivity [91], in which the context is the sequence of static abstractions of the objects (i.e., allocation sites) that are run-time receivers of methods preceding the analyzed method on the call stack. Of particular interest for our work is the object-sensitive naming scheme because, to a large degree, it reflects object ownership and is suitable for improving the analysis precision for real-world applications making use of a large number of object-oriented data structures.

Figure 3.5(b) shows the 1-object-sensitive version of the copy graph, in which an object is named using its allocation site together with the allocation site of the receiver object of the method in which the object is created. For objects created in a constructor, the context is usually their run-time owner. By adding context sensitivity, paths that start from O_{40} and O_{42} do not share any nodes. Note that there are no contexts for nodes $O_{39}, \ldots, O_{45}$ because they are created in static method `main` which does not have a receiver object. Although longer context strings may increase precision, our tool limits the length of the context to 1 since it could be prohibitively expensive (both in time and space) to employ longer contexts in a dynamic analysis.

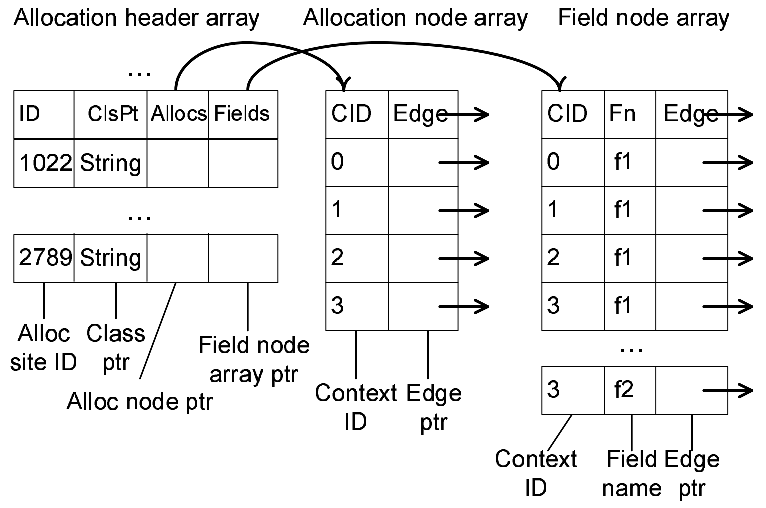
3.3 Copy Graph Construction

This section presents the details of the copy profiling technique. As the program executes and application data is read or written, the information flow analysis presented in Chapter 2 updates the corresponding tracking data.

The copy graph construction algorithm consists of two main components: (1) "compile time" instrumentation, which occurs at run time during JIT compilation,



(a) Data structure for static field nodes



(b)Data structure for allocation nodes and instance fields nodes for 1-object-sensitive copy graph

Figure 3.6: Data structure overview.

and (2) run-time profiling. To avoid having to modify both the interpreter and the JIT, we run the VM in a JIT-only mode such that all methods in the program are compiled by the JIT prior to their first invocation, allowing the tool to track data flow throughout the entire program.

3.3.1 Data Structure Design

The data structure design for the copy graph is important for minimizing overhead. The goal of the design is to allow efficient mapping from a run-time heap location to its name (which in our analysis is a copy graph node address). Figure 3.6 shows an overview of the data structures for the copy graph. Static field nodes are stored in a singly-linked-list that is constructed at instrumentation time. The node address is hard-coded in the generated executable code, so that the retrieval of nodes does not contribute to running time (thus, the analysis does not need to use the shadow locations for static fields). Each node has an edge pointer, which points to a linked list of copy graph edges that leave this node. Edge adding occurs at run time. If an existing edge is found for a pair of a source node and a target node, a new edge is not added. Instead, the frequency field of the existing edge is incremented. The size field (i.e., number of bytes) can be determined at compile time by inspecting the type of data that the copy transfers.

Allocation site nodes and instance field nodes are implemented using arrays to allow fast access. For each allocation site, a unique integer ID is generated at compile time (the IDs start from 0). The ID is used as the index into an array of *allocation headers*. Each allocation header corresponds to one ID, and points to an array of *allocation nodes* and to an array of *field nodes*, both specific to this ID. For a context-insensitive copy graph, the allocation node array for the ID has only one element. For the context-sensitive copy graph that requires a unique allocation node for each calling context (i.e., the allocation site ID of the receiver object of the surrounding method), each element of the allocation node array corresponds to a different calling context. In the current implementation the array does not grow dynamically, thus the number of calling contexts for each allocation site is limited to a pre-defined value c. We have experimented with different values of c and these results are reported in Section 3.6. An encoding function maps an allocation site ID representing a context to a value in $[0, c-1]$; currently, we use a simple mod operation $contextAllocId \,\%\, c$ to encode contexts. As reported in Section 3.6, very few contexts for an object have conflicts (i.e., they map to the same value) when using this function. A default c value of 4 was used for the studies described in Section 3.6.

The field node array is created similarly. The order of different fields in the array is dependent on the offsets of these fields in the class. We build a class metadata table at the time the class is resolved by the JIT. The table sorts fields based on their offsets, and maps each field to a unique ID (starting from 0) indicating its order in the field node array. For each instance field declared in the type (and all its supertypes) instantiated at the allocation site, there are 1 (i.e., for context-insensitive naming) or c (i.e., for 1-object-sensitive naming) entries in the field node array. For example, consider an instance field dereference $a.f$ for which the allocation site ID of the object pointed-to by a is 1000, the corresponding context allocation ID is 245, the offset of f is 12, and this offset (at compile time) is mapped to field ID $i = class\_metadata[12]$. The corresponding copy graph node address can be obtained from the element with index $c * i + 245 \,\%\, c$ in the array pointed-to by column *Fields* of $alloc\_headers[1000]$.

3.3.2 Instrumentation Relation CG $\Rightarrow^a$ CG$'$

Our instrumenter takes the assembly-like J9 intermediate representation (IR) as input, and feeds the instrumented IR to the code generator. The goal of the instrumentation is to insert code to propagate the address of a copy graph node at run time. The copy graph node represents the heap location from which a piece of data comes.

[1. local=alloc]

$$\frac{\begin{array}{l} SH(i) = (AllocID(new\ O), SH(this)\&0xFFFFFFFF) \\ CG' = CG \cup CreateAllocHeaderEntry(O, AllocID(new\ O)) \\ \mathbb{E} \vdash SH(i) : addr_{rhs} \\ shadow_i = addr_{rhs} \end{array}}{CG \Rightarrow^{i=new\ O} CG'}$$

[2. local=static]

$$\frac{\begin{array}{l} \mathbb{E} \vdash F : addr_{rhs} \\ CG' = CG \\ shadow_i = addr_{rhs} \end{array}}{CG \Rightarrow^{i=F} CG'}$$

[3. local=instance field dereference]

$$\frac{\begin{array}{l} \mathbb{E} \vdash (SH(a), Offset(f)) : addr_{rhs} \\ CG' = CG \\ shadow_i = addr_{rhs} \end{array}}{CG \Rightarrow^{i=a.f} CG'}$$

[4. local=local]

$$\frac{\begin{array}{l} CG' = CG \\ shadow_i = shadow_j \end{array}}{CG \Rightarrow^{i=j} CG'}$$

[5. static=local]

$$\frac{\begin{array}{l} \mathbb{E} \vdash F : addr_{lhs} \\ CG' = CG \cup CreateEdge(shadow_i, addr_{lhs}) \end{array}}{CG \Rightarrow^{F=i} CG'}$$

[6. instance field dereference=local]

$$\frac{\begin{array}{l} \mathbb{E} \vdash (SH(a), Offset(f)) : addr_{lhs} \\ CG' = CG \cup CreateEdge(addr_{lhs}, shadow_i) \end{array}}{CG \Rightarrow^{a.f=i} CG'}$$

[7. local=computation]

$$\frac{\begin{array}{l} edge_c = CreateEdge(shadow_c, C) \\ edge_d = CreateEdge(shadow_d, C) \\ CG' = CG \cup edge_c \cup edge_d \end{array}}{CG \Rightarrow^{i=c+d} CG'}$$

Figure 3.7: Run-time effects of instrumentation.

The intraprocedural instrumentation is illustrated at a high-level in Figure 3.7. Based on the techniques described earlier, the name environment $\mathbb{E}$ maps each heap location to the address of its corresponding copy graph node. Function *SH* (i.e., shadow heap) returns, for each object, its allocation site ID and its context allocation site ID. For example, $\mathbb{E} \vdash SH(i) : addr_{rhs}$ in rule 1 says that given the (allocation ID, context ID) pair for the heap object pointed-to by local variable i, $\mathbb{E}$ maps this pair to the copy graph node at address $addr_{rhs}$. Here $addr_{rhs}$ and $addr_{lhs}$ represent the addresses of the copy graph nodes for the heap locations corresponding to the right/left-hand-side expressions of an instruction. Each rule describes the update of the copy graph (i.e., CG) for a type of instruction, with unprimed and primed symbols representing the copy graph before and after the instruction is executed.

In rule 1, the shadow of local variable i is assigned the address of the copy graph allocation node representing the newly-created heap object. If the method containing the allocation site is an instance method, the context object is the object referenced by `this`. The bit operation (& *0xFFFFFFFF*) retrieves the lower 4 bytes from the shadow heap location, which stores the allocation site ID for `this` itself (while the higher 4 bytes contain the allocation site ID of `this`'s context). A static method does not have a context.

Before each call site in a caller, the shadow variables for the actual parameters are pushed on the tracking stack, and they are popped at the entry of the callee method. Similarly, at the exit of the callee method, the shadow variable for the returned value is pushed, and it is popped after the call site in the caller. Data carried by exception flow is not tracked by the tool.

Once a heap load operation is seen (rules 2 and 3), the address of the node representing the heap location is stored in the shadow variable. Upon a heap store (rules 5 and 6), an edge with the source node address (contained in the shadow variable) and target node address (obtained from the heap location) is created, and the graph is updated with this new edge. In rule 7, once data comes to a computation instruction, we create edges to connect the copy graph node for each participating variable with the consumer node C.

3.4 Copy Graph Client Analyses

This section presents three client analyses implemented in J9. These clients analyze the copy graph and generate reports that are useful for understanding run-time behavior and pinpointing performance bottlenecks.

3.4.1 Hot Copy Chains

Given a copy chain with frequency n and data size s, its copy volume is $n \times s$. The copy volume of a chain is the total amount of data transmitted along that chain. Chains with large copy volumes are more likely to be sources of performance problem. Another important metric is chain length—the longer a copy chain is, the more wasteful memory operations it contains. Considering both factors, we compute a *waste factor* (WF) for each chain as the product of length and copy volume. The goal of the hot chain analysis is to find copy chains that have large WF values.

The first issue is how to recover chains from copy graph edges. We use a brute-force approach which traverses the copy graph and computes the set of all distinct paths whose length is smaller than a pre-defined threshold value. If a path is a true copy chain, all its edges should have the same frequency. Based on this observation, the WF for each path is computed by using its smallest edge frequency as the path frequency. The resulting copy graph paths are ranked based on their WF values, and the top paths are reported. An example of a chain reported for benchmark antlr from DaCapo is as follows:

```
(355162, 2):
array[antlr/PreservingFileWriter:61].ELM
        — [java/io/BufferedWriter.write:198, 177581, 2] →
array[java/io/BufferedWriter:108].ELM
        — [sun/io/CharToByteUTF8.convert:262, 177759, 2] →
array[sun/nio/cs/StreamEncoder$ConverterSE:237].ELM
```

The chain contains three nodes connected by two edges. The pair (355162,2) shows the WF and the chain length. Each node in this example is an array element node. For instance field nodes and array element nodes, the allocation site of the base object is also shown. In this example, line 61 in class antlr.PreservingFileWriter creates the array whose elements are the sources of the copy chain. An edge shows the method where its last copy operation occurs (e.g., line 198 in method java.io.BufferedWriter.write), the edge frequency (e.g., 177581), and the data size (e.g., 2 bytes).

3.4.2 Clone Detector

Many applications make expensive clones of objects. A cloned object can be obtained via field-to-field copies from another object (e.g., as usually done in `clone` methods), or by adding data held by another object during initialization (e.g., many container classes have constructors that can initialize an object from another container object). Although clones are sometimes necessary, they indicate the existence of wasteful operations and redundant data. For instance, in our running example, `slowClone` initializes a new list by copying data from an existing list. Invoking this method many times may cause performance problems. The goal of this analysis is to find pairs of allocation sites, each of which represents the top (i.e., root) of a heap object subgraphs, such that a large amount of data is copied from one subgraph to the other.

For each copy graph edge $O_1.f \xrightarrow{a,b} O_2.g$, where f and g are instance fields, the value of $a \times b$ is counted as part of the direct flow from O_1 to O_2. The total direct flow for pair (O_1, O_2) shows how many bytes are copied from fields of O_1 to fields of O_2. Next, the analysis considers the indirect flow between objects. Suppose that some field of O_1 points to an object O_3, and some field of O_2 points to an object O_4. Furthermore, suppose that there is direct flow (i.e., some copy volume) from O_3 to O_4. In addition to attributing this copy volume to the pair (O_3, O_4), we want to also attribute it to the pair (O_1, O_2). This is done because O_1 may potentially be the root of an object subgraph for a data structure containing O_3. Similarly, O_2 may be the root of a data structure containing O_4. If copying is occurring for the entire data structures, the copy volume reported for pair (O_1, O_2) should reflect this.

The analysis considers all objects O_i reachable from O_1 along reference chains of a pre-defined length (length 3 was used for the experiments). Similarly, all objects O_j reachable from O_2 along reference chains of this length are considered. The copy volume reported for (O_1, O_2) is the sum of the direct copy volumes for all such pairs (O_i, O_j), including the direct flow from O_1 to O_2. To determine all relationships of the form "O' points to O", the analysis considers chains such that O is the producer node—that is, the value propagated along the chain is a reference to O. For any field node $O'.h$ in such a chain, object O' points to object O.

In the running example, `slowClone` illustrates this approach. At line 31, a new `List` object is created. Its field `elems` points to an array which is initialized with the contents of the array pointed to by the `List` created at line 39. In the first step of the analysis, volume 4000 is associated with the two array objects (1000 copies of 4-byte references to `Integer` objects). This volume is then also attributed to the two `List` objects, represented by pair (O_{39}, O_{31}), and to the two `ListClient` objects that own the lists, represented by pair (O_{44}, O_{32}). Ultimately, the reason for this entire copy volume is the cloning of a `ListClient` object, even though it manifests in the copying of the array data owned by this `ListClient`. Reporting the pair (O_{44}, O_{32}) highlights this underlying cause.

3.4.3 Not Assigned To Heap (NATH)

The third client analysis detects allocation sites that are instantiated many times and whose object references do not flow to the heap. For instance, O_{44} and O_{45} in the running example represent objects whose references are never assigned to any heap object or static field. These allocation sites are likely to represent the tops of temporary data structures that are constructed many times to provide simple services. For example, we have observed an application that creates GregorianCalendar objects inside a loop. These objects are used to construct the date fields of other objects. This causes significant performance degradation, as construction of GregorianCalendar objects is very expensive. In addition, these objects are usually temporary and short-lived, which may lead to frequent garbage collection. A simple fix that moves the object construction out of the loop can solve the problem. The escape analysis performed by a JIT usually does not remove this type of bloat, because many such objects escape the method where they are created, and are eventually captured far away from the method. Using copy graph, this analysis can be easily performed by finding all allocation nodes that do not have outgoing edges. These nodes are ranked based on the numbers of times that they are instantiated. Using the information provided by this analysis, we have found in Eclipse 3.1 a few places where NATH objects are heavily used. Running time reduction can be achieved after a simple manual optimization that avoids the creation of these objects.

3.4.4 Other Potential Clients

There are a variety of performance analyses that can take advantage of the copy graph. For example, one can measure and identify useless data by finding nodes that cannot reach the consumer node, and by aggregating them based on the objects that they belong to. As another example, developers of large applications usually maintain a performance regression test suite, which will be executed across versions of a program to guarantee that no performance degradation results from the changes. However, these performance regression tests can easily fail due to bug fixes or the addition of new features that involve extra memory copies and method invocations. It is labor-intensive to find the cause of these failures. Differentiating the copy graphs constructed from the runs of two versions of the program with the same input data can potentially help pinpoint performance problems that are introduced by the changes. A possible direction for future work is to investigate these interesting copy-graph-based analyses.

3.5 Using Copy Profiles to Find Bloat

This section presents three case studies of using copy profiles, both flat and ones derived from the copy graph, to pinpoint sources of useless work.

3.5.1 DaCapo Bloat

Inspecting the total copy count of the DaCapo bloat benchmark, we found a high volume of data copies. Averaged across all method invocations, 28% of all operations were copies from one heap location to another. This indicated that there were big opportunities for optimizing away excessive computations and temporary object construction.

When inspecting the cumulative copy profile (i.e., a copy profile that counts copies in a method and any methods it invokes), we found that approximately 50% of all data copies came from a variety of `toString` and `append` methods. Inspecting the source code, we found that most of these calls centered around code of the form: `Assert.isTrue(cond, "bug:  " + node)`. This benchmark was written prior to the existence of the Java `assert` keyword. This coding pattern meant that debugging logic resulted in entire data structures being serialized to strings, even though most of the time the strings themselves were unused; the `isTrue` method does not use the second parameter, if the first parameter is `true`. We made a simple modification to eliminate the temporary strings created during the most important copying methods[2]. This resulted in a 65% reduction in objects created, and a 29–35% reduction in execution time (depending on the JVM used; we tried Sun 1.6.0_10 and IBM 1.6.0 SR2).

The DaCapo suite is geared towards JVM and hardware designers. In the design of this suite, it is important to distinguish inefficiencies that a JIT could possibly eliminate from ones that require a programmer with good tools.

3.5.2 Java 5 GregorianCalendar

A recurring problem with the Java 1.5 standard libraries is the slow performance of calendar-related classes [142]. Many users experienced a 50× slowdown when upgrading from Java 1.4 to Java 1.5. The problems centered around methods in class `GregorianCalendar`, which is an important part of date formatting and parsing. We ran the test case provided by a user and constructed a context-sensitive copy graph. The test case makes intensive calls of the `before`, `after`, and `equals` methods. The report of hot copy chains includes a family of hot chains with the following structure:

```
array[Calendar:907].ELM
    — [Calendar.clone:2168,510000] →
array[Calendar:2169].ELM
```

[2]We commented out the `toString` methods of `Block`, `FlowGraph`, `RegisterAllocator`, `Liveness`, `Node`, `Tree`, `Label`, `MemberRef`, `Instruction`, `NameAndType`, `LocalVariable`, `Field`, and `Constant`.

This chain (and others similar to it, for the fields of a calendar) suggests that `clone` is invoked many times to copy values from one Calendar to another. To confirm this, we ran the clone detector and the top four pairs of allocation sites were as follows:

```
340000: (GregorianCalendar[GregorianCalendarTest:11],array[Calendar:2168])

340000: (array[Calendar:906],array[Calendar:2168])

340000: (array[Calendar:907],array:[Calendar:2169])

340000: (array[Calendar:908],array[Calendar:2170])
```

The first pair shows that an array created at line 2168 of `Calendar` gets a large amount of data from the `GregorianCalendar` object created in the test case. The remaining three pairs of allocation sites also suggest the occurrence of clones, because the first group of objects (i.e., at lines 906, 907, 908) are arrays created in the constructor of `Calendar`, while the second group (i.e., at lines 2168, 2169, and 2170) are arrays created in `clone`. By examining the code, we found that `clone` creates a new object by deep copying all array fields from the old `Calendar` object. These copies also include the cloning of a time zone from the *zone* field of the existing object. Upon further inspection, we found the cause of the slowdown: methods `before`, `after`, and `equals` invoke method `compareTo` to compare two `GregorianCalendar` objects, which is implemented by comparing the current times (in milliseconds) obtained from these objects. However, `getMillisof` does not compute time directly from the existing calendar object, but instead makes a clone of the calendar and obtains the time from the clone.

The JDK 1.4 implementation of `Calendar` does not clone any objects. This is because the 1.4 implementation of `getMillisof` mistakenly changes the internal state of the object when computing the current time. In order to avoid touching the internal state, the implementers of JDK 1.5 made the decision to clone the calendar and get the time from the clone. Of course, it is not a perfect solution as it fixes the original bug at the cost of introducing a significant performance problem. Our tool highlighted the useless work being done in order to work around the `getMillisof` issue.

3.5.3 DaCapo Eclipse

As a large framework-based application, Eclipse suffers from performance problems that result from the pile-up of wasteful operations in its plugins. These problems impact usability, and even programmers' choice when comparing Java development tools [68]. We ran Eclipse 3.1 from the DaCapo benchmark set and used the NATH analysis to identify allocation sites whose run-time objects are never assigned to the heap. The top nine allocation sites are shown below:

(1) 295,004: org/eclipse/jdt/internal/compiler/ISourceElementRequestor$MethodInfo

[SourceElementParser:968]

(2) 161,169: .../SimpleWordSet[SimpleWordSet:58]

(3) 145,987: .../ISourceElementRequestor$FieldInfo[SourceElementParser:1074]

(4) 46,603: .../ContentTypeCatalog$7[ContentTypeCatalog:523]

(5) 46,186: .../ISourceElementRequestor$TypeInfo[SourceElementParser:1190]

(6) 45,813: .../Path[PackageFragment:309]

(7) 44,703, .../Path[CompilationUnit:786]

(8) 37,201, .../ContentTypeHandler[ContentTypeMatcher:50]

(9) 30,939, .../HashtableOfObject[HashtableOfObject:123]

Each line shows an allocation site and the number of times it is instantiated. For example, the first line is for an allocation site at line 968 in class `SourceElementParser`, which creates 295004 objects of type `ISourceElementRequestor$MethodInfo`. Sites 4 and 8 are from plugin org.eclipse.core.resources. The remaining sites are located in org.eclipse.jdt.core. Because the Eclipse 3.1 release does not contain the source code for org.eclipse.core.resources, we inspected only the seven sites in the JDT plugin.

The first site is located in class `SourceElementParser`, which is a key part of the JDT compiler. JDT provides many source code manipulation functionalities that can be used for various purposes, such as automated formatting and refactoring. The observer pattern is used to provide source code element objects when a client needs them. Method `notifySourceElementRequestor`, which contains this site, plays the observer role: once a requestor (i.e., a client) asks for a compilation unit node (i.e., a class), the method notifies all child elements (i.e., methods) of the compilation unit by calling method `enterMethod`, which will subsequently notify source code statements in each method. Method `enterMethod` takes a `MethodInfo` object as input; this object contains all necessary information for the method that needs to be notified.

The site creates `MethodInfo` objects which are then provided to `enterMethod`. Because `enterMethod` is defined in an interface, we checked all implementations of the method. Surprisingly, none of these implementations invoke any methods on this parameter object. They extract all information about the method to be notified from fields of the object; these fields are previously set by `notifySourceElementRequestor`. The third and the fifth allocation sites from above tell the same story: these hundreds of thousands of objects are created solely for the purpose of carrying data across one-level method invocations. It is expensive to create and reclaim these objects, and to perform the corresponding heap copies. We modified the interface and all related implementations to pass data directly through parameters. This modification reduces

Class	*Modification*	*#Objs*	*#GCs*	*Time(s)*
Original	—	273991250	478	143.6
MethodInfo, Field-Info, TypeInfo	Directly pass the data	272461138	460	139.6
PackageFragment	Get IResource directly from String	272429471	448	138.3
SimpleWordSet	In-place rehash	272395776	430	136.8
HashtableOfObject	In-place rehash	272320499	424	134.0

Table 3.1: Eclipse 3.1 performance problems, fixes, and performance improvements.

the number of allocated objects by millions and improves the running time by 2.8%. In large applications with no single hot spot, significant performance improvements are possible by accumulating several such "small" improvements, as illustrated below.

Table 3.1 shows a list of several problems we identified with the help of the analyses. For each problem, the table shows the problematic class (*Class*), our code modification, the number of allocated objects (*#Objs*), the total number of GC invocations (*#GCs*), and the running times. Row *Original* characterizes the original execution. Each subsequent row shows the cumulative improvements due to our changes in the JDT plugin. The second row corresponds to allocation sites 1, 3, and 5 listed above, the third row is for sites 6 and 7, the fourth row is for site 2, and the last row is for site 9.

By modifying the code to eliminate redundant copies and the related creation of objects, we successfully reduced the number of GC runs, the number of allocated objects, and the total running time. With the help of the tool, it took us only a few hours to find these problems and to make modifications in a large application we had never studied before.

It is important to note that this effort just scratches the surface: significant performance improvement may be possible if a developer or a performance expert carefully examines the tool reports (with different tests and workloads) and eliminates the identified useless work. This is the kind of manual tuning that is already being done today for large Java applications with performance problems that cannot be attributed to a single hot spot. This tedious and labor-intensive process can be made more efficient and effective by the dynamic analyses proposed in our work. Future studies should investigate such potential performance improvements for a broad range of Java applications.

3.6 Copy Graph Characteristics

This section presents characteristics of the copy graph and its construction. The maximum heap size specified for each run was 500Mb. Hence, the size of shadow for each run was 500Mb. IBM DMS is the IBM document management server, which is run on top of a J2EE application server. Each DaCapo benchmark was run with large workload for two iterations, and the running time for the second iteration is shown. SPECjbb and IBM DMS are server applications that report throughput, not total running time; both were run for 30 minutes with a standard workload.

Table 3.2 presents the time and space overhead of context-insensitive copy graphs. The second column, labeled T_{orig}, presents the original running times in seconds. The remaining columns show the total numbers of nodes N_0 and edges E_0, the amount of memory M_0 needed by the analysis (in megabytes), the running times T_0 (in seconds), and the performance slowdowns (shown in parentheses). The slowdown for each program is T_0/T_{orig}. Because the shadow heap is 500Mb, the space overhead of the copy graph is M_0–500.

In Table 3.3, Table 3.4, and Table 3.5, the same measurements are reported for 1-object-sensitive copy graphs. To understand the impact of the number of context slots (i.e., parameter c from Section 3.3.1), we experimented with values 4, 8 and 16 when constructing the 1-object-sensitive copy graph. The slowdown for each program was calculated as T_i/T_{orig} (the original time from Table 3.2), where $i \in \{4, 8, 16\}$.

The copy graph itself consumes a relatively small amount of memory. Other than for IBM DMS, the space overhead of the copy graph does not exceed 27Mb even when using 16 context slots. As expected, a context-sensitive copy graph consumes more memory than the context-insensitive one, and using more context slots leads to larger space overhead.

The running time overheads for profiling the context-insensitive copy graph and the three versions of 1-object-sensitive copy graphs are, on average, 36×, 37×, 37×, and 37× respectively. This overhead is not surprising because the analysis tracks the execution of every instruction in the program. The overhead also comes from synchronization performed by the instrumentation of allocation sites, which sequentially executes the allocation handler to create allocation header elements. The current implementation provides a general facility for mapping an object address to a context ID. This is done even for the context-insensitive analysis, where the ID is always 0. Since the cost of this mapping is negligible, we have not created a specialized context-insensitive implementation. Hence, the difference between the running times of profiling context-insensitive and context-sensitive copy graphs is noise. The only

Program	*Original*	Context-insensitive			
	$T_{orig}(s)$	$\#N_0$	$\#E_0$	$M_0(Mb)$	$T_0(s)$ $(\times)$
antlr	8.9	12516	56703	503.7	284.2 (31.9)
bloat	157.5	14058	14471	502.2	9812.2 (62.4)
chart	32.5	18113	12810	502.5	1053.2 (32.4)
fop	3.6	12419	7675	501.8	38.2 (10.6)
pmd	46.6	11289	8418	501.7	1542.4 (33.1)
jython	74.7	25653	21893	503.2	2826.1 (37.8)
xalan	64.8	13505	28678	502.6	3030.5 (46.8)
hsqldb	13.5	12294	9102	501.7	350.0 (25.9)
luindex	12.1	10154	10227	501.6	583.4 (48.2)
lusearch	19.2	8390	13849	501.5	662.8 (34.5)
eclipse	124.7	34074	52957	506.5	4343.8 (34.8)
SPECjbb	1800*	17146	12637	502.4	1800*
IBM DMS	1800*	147517	87531	519.6	1800*

Table 3.2: Copy graph size and time/space overhead, part 1. Shown are the original running time T_{orig}, as well as the total numbers of graph nodes N_0 and edges E_0, the total amount of memory consumed M_0, the running time T_0, and the slowdown (shown in parentheses) when using a context-insensitive copy graph.

significant difference between context-insensitive and context-sensitive analysis is the space overhead.

Although significant, these overheads have not hindered us from running the tool on any programs, including real world large-scale production applications. It was an intentional design decision *not* to focus on the performance of the analysis, but instead focus on the content collected and on demonstrating that the results are useful for finding performance problems in real programs. Now that the value of the tool has been established, a possible future direction is to use sampling-based profiling to obtain the same or similar results. Another possibility is to employ static pre-analyses that reduce the cost of the subsequent dynamic analysis.

Program	1-object-sensitive ($c = 4$)			
	$\#N_4$	$\#E_4$	$M_4(Mb)$	$T_4(s)$ $(\times)$
antlr	48556	112907	506.9	294.8 (33.1)
bloat	54960	35678	504.3	10182.9 (64.7)
chart	69438	25951	504.6	1079.4 (33.2)
fop	47893	11985	503.1	37.4 (10.4)
pmd	43740	15576	503.0	1586.7 (34.0)
jython	95493	32256	505.8	2865.6 (38.4)
xalan	52485	55367	504.9	2983.3 (46.0)
hsqldb	47666	13432	503.0	358.0 (26.5)
luindex	39319	17695	502.8	568.7 (47.0)
lusearch	32354	22163	502.6	643.5 (33.5)
eclipse	131065	124043	512.3	4521.5 (36.3)
SPECjbb	66102	23909	503.3	1800*
IBM DMS	193707	180187	533.7	1800*

Table 3.3: Copy graph size and time/space overhead, part 2. The columns report the same measurements as Table 3.2, but for 1-object sensitive copy graph with 4 context slots.

Program	1-object-sensitive ($c = 8$)			
	$\#N_8$	$\#E_8$	$M_8(Mb)$	$T_8(s)$ $(\times)$
antlr	96609	159042	510.2	300.7 (33.8)
bloat	109494	48840	506.5	10147.4 (64.4)
chart	137945	39133	507.3	1054.4 (32.4)
fop	95180	13509	504.6	37.2 (10.3)
pmd	86980	19568	504.5	1568.5 (33.7)
jython	188583	37005	509.0	2879.9 (38.6)
xalan	85751	88001	507.7	3067.6 (47.3)
hsqldb	94846	15201	504.6	346.7 (25.7)
luindex	78232	22912	504.3	581.1 (48.0)
lusearch	64280	26629	503.8	651.6 (33.9)
eclipse	259168	154004	517.4	4545.3 (36.4)
SPECjbb	131413	27660	507.2	1800*
IBM DMS	381072	242049	571.2	1800*

Table 3.4: Copy graph size and time/space overhead, part 3. The columns report the same measurements for 1-object sensitive copy graph with 8 context slots.

Program	1-object-sensitive ($c = 16$)			
	$\#N_{16}$	$\#E_{16}$	$M_{16}(Mb)$	$T_{16}(s)$ $(\times)$
antlr	192713	210522	515.2	309.5 (34.8)
bloat	218558	60483	510.5	10068.2(63.9)
chart	274903	45071	511.9	1056.5 (32.5)
fop	189757	14388	507.7	36.8 (10.2)
pmd	173484	21339	507.3	1555.5 (33.4)
jython	374791	41027	515.0	2861.4 (38.3)
xalan	208119	117760	512.2	3067.6 (47.3)
hsqldb	189183	17190	507.7	345.9 (25.6)
luindex	156033	28333	507.0	564.8(46.7)
lusearch	128152	32544	506.1	658.4(34.3)
eclipse	516030	174846	526.4	4746.4 (38.1)
SPECjbb	261915	29017	511.0	1800*
IBM DMS	755829	304759	652.3	1800*

Table 3.5: Copy graph size and time/space overhead, part 4. The columns report the same measurements for 1-object sensitive copy graph with 16 context slots.

Program	*#Chains*	*Length*	*#NATH Sites*	*#NATH Objects*
antlr	250680	2.60	811	411536
bloat	6955316	4.00	1160	31217025
chart	29490	1.16	1652	15080848
fop	275835	3.36	1282	167808
pmd	436397	2.96	1062	54103059
jython	6827057	4.00	493	35926287
xalan	93263	2.60	1218	6186112
hsqldb	8595	1.80	828	3059666
luindex	30183	2.24	749	5543579
lusearch	10640	3.8	302	4200325
eclipse	10070910	1.24	3030	3494187
SPECjbb	21468	2.00	575	722800
IBM DMS	1937646	3.75	4695	1413528

Table 3.6: Copy chains and NATH objects. All copy graph paths with length ≤ 5 are traversed to compute hot chains. The columns show the total number of generated chains, the average length of these chains, and the number of NATH allocation sites and NATH run-time objects.

Program	Average fan-out				Context conflict ratio		
	CIFO	*CSFO* 4	*CSFO* 8	*CSFO* 16	*CCR* 4	*CCR* 8	*CCR* 16
antlr	4.66	2.33	1.64	1.09	0.237	0.131	0.081
bloat	1.03	0.64	0.44	0.27	0.199	0.090	0.068
chart	0.70	0.36	0.28	0.16	0.118	0.059	0.028
fop	0.62	0.25	0.15	0.08	0.134	0.060	0.043
pmd	0.75	0.35	0.22	0.12	0.131	0.059	0.051
jython	0.80	0.31	0.18	0.10	0.079	0.071	0.024
xalan	2.13	1.79	0.81	0.56	0.128	0.067	0.040
hsqldb	0.74	0.28	0.16	0.09	0.169	0.080	0.051
luindex	1.02	0.45	0.29	0.18	0.148	0.073	0.051
lusearch	1.68	0.68	0.41	0.25	0.127	0.082	0.052
eclipse	1.53	0.91	0.57	0.33	0.193	0.114	0.071
SPECjbb	0.75	0.36	0.21	0.11	0.144	0.065	0.026
IBM DMS	0.76	0.32	0.17	0.09	0.112	0.047	0.027

Table 3.7: Average node *fan-out* for context-insensitive (*CIFO*) and context-sensitive (*CSFO-i*) copy graphs, as well as average *context conflict ratios* (*CCR-i*) for the context-sensitive copy graphs.

Table 3.6 shows measurements for the copy chains obtained from a context-insensitive copy graph, including the total number of generated chains (*#Chains*) and the average length of these chains (*Length*). The table also shows the number of NATH allocation sites and NATH run-time objects. The significant numbers of NATH objects indicate that eliminating such objects may be a worthwhile goal for future work on manual and automatic optimizations.

The first part of Table 3.7 lists the average node *fan-out* for the context-insensitive copy graph (*CIFO*) and the three versions of context-sensitive copy graphs (*CSFO-i*, where i is the number of context slots for each object). A node's fan-out is the number of its outgoing edges. The average fan-out indicates the degree of node sharing among paths in the graph. Note that *CIFO* and *CSFO-i* are small, because there exist a large number of producer nodes (allocation site) that do not have outgoing edges. In addition, the more slots are used to represent contexts, the smaller the average fan-out, because more nodes are created to avoid path sharing.

In addition, for each context-sensitive copy graph, the table reports the average *context conflict ratio* (*CCR-i*). The CCR for an object o is defined as follows:

$$\mathit{CCR\text{-}i}(o) = \begin{cases} 0 & \max_{0 \le k \le i}(nc[k]) = 1 \\ \max(nc[k]) / \sum nc[k] & \text{otherwise} \end{cases}$$

Here $nc[k]$ represents the number of distinct contexts that fall into context slot k. The CCR value captures the degree to which our encoding function (i.e., $id \,\%\, k$) causes distinct contexts to be merged in the copy graph. For example, the CCR is 0 if each context slot represents at most one distinct context; the CCR is 1 if all contexts for the object fall into the same slot. The table reports the average CCR for all allocation sites in the copy graph. As expected, the average CCR decreases with an increase in the number of context slots. Note that very few context conflicts occur even when c = 4, because a large number of objects have only one distinct context during theirs lifetimes.

3.7 Summary and Interpretation

In large-scale systems, data-based activities such as large volumes of copies are sometimes stronger signs of excess than control-based activities such as method execution time, used primarily in compilers and tools. Based on this observation, this chapter introduces a technique to help developers find performance improvements that are beyond the scope of what is typically achieved by current JIT technology. Using real-world examples, we show that this analysis can quickly guide the programmer to the problematic areas of the program, allowing them to inspect the code to find optimization opportunities. This analysis could also expose additional opportunities for interprocedural JIT optimizations, such as specializing across multiple components, or hoisting complex, many-layered computations.

The success of this technique has clearly demonstrated that there exist large optimization opportunities that remain untapped by the JIT compiler. As long as we provide developers with tools that can help them make better sense of heaps and executions, their insights and experience will soon empower them to find opportunities that the JIT cannot find, and quickly fix problems that would require dozens of sophisticated analyses to combine together to fix automatically.

The study presented in this chapter confirms that data-based activities (e.g., copying of data) can sometimes be more interesting than control-based activities (e.g., method invocations) in bloat detection. When method invocation counts and execution times fail to expose performance bottlenecks (e.g., in the document management server discussed in Chapter 1), these data-oriented observations become valuable and more informative. Excessive copying is just one example of such an activity. In the next three chapters we present another three profiling techniques, focused on different kinds of data-based activities. All these techniques are effective in uncovering bloat and helping developers diagnose performance problems.

CHAPTER 4: Making Sense of Cost and Benefit

Bloat can be also be caused by inappropriate choices of data structures and implementation algorithms, leading to computations with *high cost* (i.e., expensive to execute) and *low benefit* (i.e., produce unnecessary data). As an example, in a large Java program we found that the programmer creates many lists and adds thousands of elements to each one of them, only for the purpose of obtaining list sizes. The values contained in most fields of the list objects are never used; these values have non-zero costs but zero benefits for the rest of the execution. Correct choices are hard to make, as they require deep understanding of implementation logic and a great deal of programming experience. These decisions often involve tradeoffs between space and time, between reusability and performance, and between short-term development goals and long-term maintenance.

Querying the costs and benefits of certain data structures is a natural and effective way for a programmer to understand the performance of her program in order to make appropriate choices. For example, questions such as "What is the cost of using this data structure?" and "Why does this expensive call produce a rarely-used value?" are often asked during software development and performance tuning/debugging. Currently, these questions are answered mostly manually, typically through a few labor-intensive rounds of code inspection, or with coarse-grained cost measurements (e.g., method running times and numbers of created instances) with the help of existing profiling tools. The answers are usually approximations that are far from the actual causes of problems, making it extremely hard to track down the performance bottlenecks. The goal of our work is to provide automated support for the performance expert to measure costs and benefits at a fine-grained (instruction) level, thus improving the precision of the answers and making the tuning tasks easier.

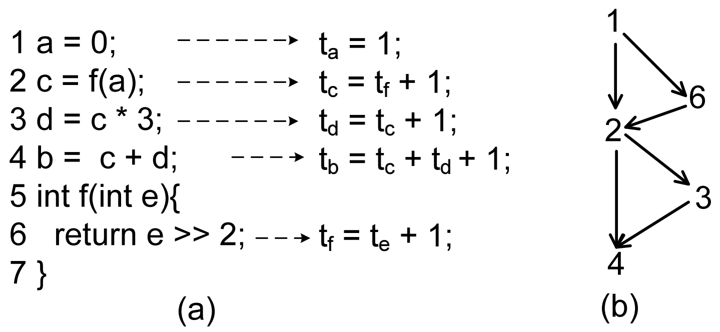
In addition to finding individual values that are likely to be results of wasteful operations, computing cost and benefit automatically provides many other advantages for resolving performance issues. For example, a number of high-level performance-related program properties can be quickly exposed by aggregating the costs and benefits of values contained in individual storage locations. These properties include, for example, whether a container is overpopulated (i.e., contains many objects but retrieves only a few of them), whether an object contains dead fields, and whether an object field is rewritten before it is read, etc. Such questions can be answered efficiently by our cost-benefit analyses presented in this chapter. To the best of our knowledge, these dynamic analyses are the first attempt to attack performance problems with a cost-benefit computation.

4.1 Technical Challenges and The Basic Idea

Cost and benefit The cost c_v of a value v can be defined as the total number of bytecode instructions (transitively) required to produce v. Each instruction is treated as having unit cost; the actual cost difference (e.g., *add* vs *mul*) is irrelevant for cost-benefit analyses of large-scale applications that execute extremely large numbers of instruction. Control dependences are not considered in the computation of c_v, because otherwise many instructions that contribute only to decision making would be taken into account, making it hard to quantify the effort made *only* to produce v. Instead, they are treated in a special manner, as discussed later. It is not obvious how to define the benefit of v, as there does not exist any explicit metric related to the "goodness" of consuming a value during the execution. We propose different benefit definitions for different analyses (i.e., for different targeted properties). For example, the benefit of v is zero when v is never used, or contributes only to the generation of never-used values.

Technical challenges The computation of cost for each value during the execution appears to be a problem which can be solved by associating a small amount of *tracking data* with each storage location and by updating this data as the corresponding location is written. A similar technique has been adopted, for example, in taint analysis [103], where the tracking data is a simple taint mark. In the setting of cost computation, the tracking data associated with each location records the cumulative cost of producing the value that is written into the location. Figure 4.1(a) shows a simple program and the update of the tracking data at each execution step; t_x denotes the tracking data for location x. For an instruction s, a simple approach of updating the tracking data for the left-hand-side variable is to store in it the sum of the tracking data for all right-hand-side variables, plus the cost of s itself.

It is easy to calculate in Figure 4.1(a) that the value of t_b is 8. However, the number of instructions that the instruction at line 4 transitively depends on is 5, shown in Figure 4.1(b). These two numbers differ because c contributes to the generation of both d and b, and including the costs of both d and c in the cost of b actually counts the cost of c twice. While this difference is small for the example, it can quickly accumulate and become significant when the size of the program increases. For example, we have observed, using our tool, that this cost can quickly grow and



```
1 a = 0;              ------→ t_a = 1;
2 c = f(a);           ------→ t_c = t_f + 1;
3 d = c * 3;          ------→ t_d = t_c + 1;
4 b =  c + d;           ---→ t_b = t_c + t_d + 1;
5 int f(int e){
6   return e >> 2;     --→ t_f = t_e + 1;
7 }
```

$$\begin{tikzcd} 1 \arrow[d] \arrow[dr] & \\ & 6 \arrow[dl] \\ 2 \arrow[dd] \arrow[dr] & \\ & 3 \arrow[dl] \\ 4 & \end{tikzcd}$$

Figure 4.1: (a) A simple program and the updates of the tracking data; (b) Corresponding data dependence graph; an edge $b \rightarrow a$ shows that instruction a uses the value defined by instruction b.

cause a 64-bit integer to overflow for even moderate-size applications. In addition, such dynamic tracking cannot provide any additional information about the data flow, and thus, its usefulness for helping diagnose performance problems is limited.

Backward dynamic flow problems To avoid double-counting in the computation of cost c_b, one could record all instruction instances before b is written and their dependences, i.e., the dependence graph in Figure 4.1(b). Cost c_b can then be computed by traversing backward the dependence graph and counting the number of instructions. There are many other dynamic analysis problems that have similar characteristics. These problems, for example, include `null` value propagation analysis [24], dynamic object type-state checking [9], event-based execution fast forwarding [169], copy chain profiling [156], etc. One common feature of this class of dynamic analyses is that they require additional trace information recorded for the purpose of diagnosis or debugging, as opposed to simpler approaches such as taint analysis. Because the solutions these analyses need to compute are history-related and can be obtained by traversing backward the recorded traces, we will refer to them as *backward dynamic flow* (BDF) problems.

In general, BDF problems can be solved by dynamic slicing [150,165,166,168]. In our example of cost computation, c_b is essentially the size of the data-dependence-based backward dynamic slice starting from the instruction that directly produces b (i.e., $b = c + d$). While there exists a range of static optimization [108,166,169] and online data compression [79,150,167] techniques, it is still extremely expensive to perform whole-program dynamic slicing, and no existing algorithms have been shown to scale to large and long-running Java applications. This is because the amount of memory needed for whole-program dynamic slicing is unbounded, and is determined completely by the run-time behavior of the program.

We introduce *abstract dynamic thin slicing*, a technique that applies dynamic thin slicing [134] over bounded abstract domains. The resulting dependence graph contains abstractions of instructions, rather than their actual run-time instances. In the context of cost computation, instead of calculating the actual costs for the values produced by concrete instruction instances, they are approximated using the *abstract costs* for the abstractions of instructions, which are essentially the aggregations of costs for concrete instructions. This technique has two advantages. First, the amount of memory required for the dependence graph is bounded by the number of abstractions, which significantly reduces the space overhead. Second, while slicing over concrete domains can be more precise, the costs computed for concrete instructions eventually have to be aggregated in order to present a meaningful report to the user. When applying slicing over abstract domains, it is not necessary to perform aggregation, because dependence graph nodes are already abstractions of concrete instructions. Section 4.2 shows that, in addition to cost-benefit analysis, such slicing can solve a range of other BDF problems.

Thin slicing [134] is a technique that considers only direct locations that are part of the data flow in the generated slice, while filtering out locations that are indirectly used to obtain the direct locations. For instance, for a seed statement $a.f = b$, its thin slice consists of the statements that contribute to the generation of the value in location $a.f$, but excludes the statements that contribute to the generation of the object reference a. This technique is particularly suitable for our cost and benefit computation, because the cost of producing the value in $a.f$ should accurately reflect the effort made only to produce this value, but not the cost of forming the reference value in a. This property makes thin slicing especially attractive for computing costs for programs that make extensive use of object-oriented data structures—with traditional slicing, the cost of each data element retrieved from a data structure would include the cost of producing the object references that form the layout of the data structure, resulting in significant imprecision.

To help the programmer diagnose performance problems, we propose several cost-benefit analyses that take as input the abstract dynamic dependence graph and report information related to the underlying causes. Section 4.3 defines in detail one of these analyses, which computes costs and benefits at the level of data structures by aggregating costs and benefits for individual heap locations.

The proposed analyses were implemented in a J9 and were successfully applied to large-scale and long-running applications such as `derby`, `tomcat` and `trade`. Section 4.4 presents an evaluation of analysis cost. Similarly to previous work [102], a shadow heap is used to record information about fields of live objects; this shadow heap has the same size as the original Java heap. The use of a shadow heap is not essential; for example, it could be replaced with a hash table. Due to the abstractions being used, the additional memory needed to maintain the dependence graph is small. The current prototype implementation imposes an average slowdown of 71 times when whole-program tracking is enabled. While this overhead is too high for production runs, it is acceptable for performance tuning and debugging, as it can reduce significantly the amount of human effort required to track down complex performance problems. Even without any overhead reduction efforts, the current implementation is able to analyze large production applications. We also show that it is possible to significantly reduce the overhead (i.e., by up to 10 times) by enabling tracking only for relevant components instead of the entire program.

Section 4.4 describes six case studies with real-world applications. Using the tool, we found hundreds of performance problems, and eliminating these problems resulted in 2% – 37% performance improvement. These problems include inefficiencies caused by common programming idioms, repeated work whose results need to be cached, computation of redundant data, and choices of unnecessary expensive operations. Some of these finding also provide useful insights for automatic code optimization in compilers.

The contributions of this work are:

- *Cost and benefit profiling*, a methodology that identifies run-time inefficiencies by understanding the cost of producing values and the benefit of consuming them.

- *Abstract dynamic thin slicing*, a general technique that performs dynamic thin slicing over bounded abstract domains. It produces much smaller and more relevant slices than traditional dynamic slicing and can be used for cost-benefit analysis and for other dynamic analyses.

- *Relative cost-benefit analysis* which reports data structures that have unbalanced cost-benefit rates.

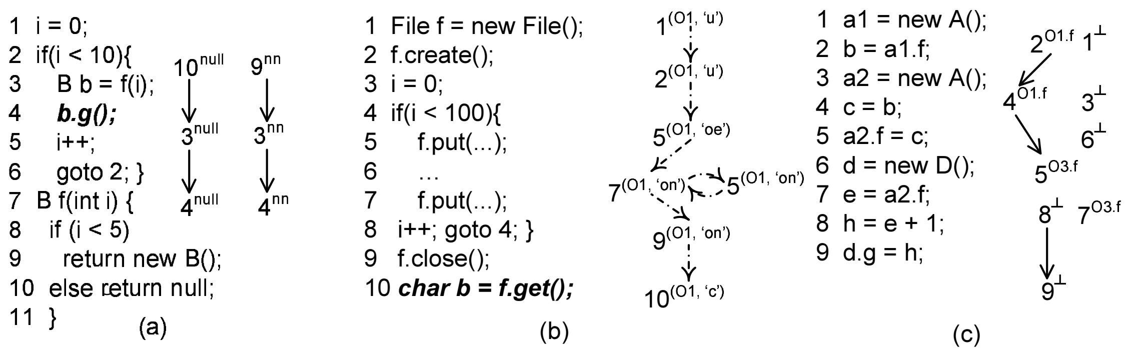
- A J9-based implementation and six case studies using real-world programs, demonstrating that the tool can help a programmer to find opportunities for performance improvements.

4.2 Cost Computation Using Abstract Slicing

In this section, we first formalize our *abstract dynamic thin slicing* technique and show example clients that can take advantage of this technique. We then give the definition of cost and present a runtime profiling technique that constructs the dependence graph. While our tool works on the low-level JVM intermediate representation, the presentation of the algorithms requires the three-address-code representation of the program. In this representation, each statement corresponds to a bytecode instruction (i.e., it is either an assignment or a computation store that contains only one operator). This makes it explicit to traverse the dependence graph to compute cost, as each statement has unit cost. We will use the term *statement* and *instruction* interchangeably in this chapter, both meaning a statement in the three-address-code representation.

4.2.1 Abstract Dynamic Thin Slicing

In dynamic slicing [76], the instrumented program is first executed to obtain an execution trace with control flow and memory reference information. At a pointer



```
1  i = 0;
2  if(i < 10){
3     B b = f(i);
4     b.g();
5     i++;
6     goto 2; }
7  B f(int i) {
8    if (i < 5)
9      return new B();
10  else return null;
11  }
```

```
1  File f = new File();
2  f.create();
3  i = 0;
4  if(i < 100){
5     f.put(...);
6     ...
7     f.put(...);
8  i++; goto 4; }
9  f.close();
10 char b = f.get();
```

```
1  a1 = new A();
2  b = a1.f;
3  a2 = new A();
4  c = b;
5  a2.f = c;
6  d = new D();
7  e = a2.f;
8  h = e + 1;
9  d.g = h;
```

Figure 4.2: Data dependence graphs for three BDF problems. Line numbers are used to represent the corresponding instructions. Arrows with solid lines are def-use edges. (a) Null origin tracking. (b) Typestate history recording; arrows with dashed lines represent "next-event" relationships. (c) Extended copy profiling; O_i denotes the allocation site at line i.

dereference, both the data that is referenced and the pointer value (i.e., the address of the data) are captured. Our technique considers only data dependences and the control predicates are treated in a special way as described later. Based on a dynamic data dependence graph inferred from the trace, a slicing algorithm is executed. Let $\mathcal{I}$ be the domain of static instructions and $\mathcal{N}$ be the domain of natural numbers.

Definition 4.2.1 (Dynamic Data Dependence Graph). *A dynamic data dependence graph $(\mathcal{V}, \mathcal{E})$ has node set $\mathcal{V} \subseteq \mathcal{I} \times \mathcal{N}$, where each node is a static instruction annotated with an integer j, representing the j-th occurrence of this instruction in the trace. An edge from a^j to b^k ($a, b \in \mathcal{I}$ and $j, k \in \mathcal{N}$) shows that the j-th occurrence of a writes a location that is then used by the k-th occurrence of b, without an intervening write to that location. If an instruction accesses a heap location through $v.f$, the reference value in stack location v is also considered to be used.*

Thin slicing [134] is a static technique that focuses on statements that flow values to the seed, ignoring the uses of base pointers. In this chapter, the technique is restated for dynamic analysis. A *thin data dependence graph*, formed from the execution trace, has exactly the same set of nodes as its corresponding dynamic data dependence graph. However, for an access $v.f$, the base pointer value in v is not considered to be used. A thin data dependence graph contains fewer edges and leads to smaller slices. Both for standard and thin dynamic slicing, the amount of memory required for representing the dynamic dependence graph cannot be bounded before or during the execution.

For some BDF problems, there exists a certain pattern of backward traversal that can be exploited for increased efficiency. Among the instruction instances that are traversed, equivalence classes can usually be seen. Each equivalence class is related to a certain property of an instruction from the program code, and distinguishing instruction instances in the same equivalence class (i.e., with the same property) does not affect the analysis precision. Moreover, it is only necessary to record one instruction instance online as the representative for that equivalence class, leading to significant space reduction of the generated execution trace. Several examples of such problems will be discussed shortly.

To solve such BDF problems, we propose to introduce the semantics of a target analysis into profiling, by defining a problem-specific bounded abstract domain $\mathcal{D}$ containing identifiers that define equivalence classes in $\mathcal{N}$. An unbounded subset of elements in $\mathcal{N}$ can be mapped to an element in $\mathcal{D}$. For a particular instruction $a \in \mathcal{I}$, an abstraction function $f_a : \mathcal{N} \rightarrow \mathcal{D}$ is used to map a^j, where $j \in \mathcal{N}$, to an abstracted instance a^d. This yields an abstraction of the dynamic data dependence graph. For our purposes we are interested in thin slicing. The corresponding dependence graph will be referred as an *abstract thin data dependence graph.*

Definition 4.2.2 (Abstract Thin Data Dependence Graph). *An abstract thin data dependence graph ($\mathcal{V}'$, $\mathcal{E}'$, $\mathcal{F}$, $\mathcal{D}$) has node set $\mathcal{V}' \subseteq \mathcal{I} \times \mathcal{D}$, where each node is a static instruction annotated with an element $d \in \mathcal{D}$, denoting the equivalence class of instances of the instruction mapped to d. An edge from a^j to b^k ($a, b \in \mathcal{I}$ and $j, k \in \mathcal{D}$) shows that an instance of a mapped to a^j writes a location that is used by an instance of b mapped to b^k, without an intervening write to that location. If an instruction accesses the heap $v.f$, the base pointer value v is* not *considered to be used. $\mathcal{F}$ is a family of abstraction functions f_a, one per instruction $a \in \mathcal{I}$.*

For simplicity, we will use "dependence graph" to refer to the abstract thin data dependence graph defined above. The number of static instructions (i.e., the size of $\mathcal{I}$) is relatively small even for large-scale programs, and by carefully selecting domain $\mathcal{D}$ and abstraction functions f_a, it is possible to require only a small amount of memory for the graph and yet preserve necessary information needed for a target analysis.

Many BDF problems exhibit bounded-domain properties. Their analysis-specific dependence graphs can be obtained by defining the appropriate abstraction functions. The following examples show a few analyses and their formulations in our framework. Note that although some of these analyses can be implemented in simpler ways (e.g., they do not need tracking all instructions and their dependences), we formulate them as abstract slicing problems to show the general applicability of our framework. In reality, their implementations can be easily derived from our framework by, for example, profiling other kinds of edges between instruction abstractions (rather than def-use edges).

Propagation of `null` Values When a `NullPointerException` is observed in the program, this analysis locates the program point where the `null` value starts propagating and the propagation flow. Compared to exiting `null` value tracking approaches (e.g., [24]) that track only the origin of a `null` value, this analysis also provides information about how this value flows to the point where it is dereferenced, allowing the programmer to quickly track down the bug. Here, $\mathcal{D}$ contains two elements *null* and *not_null*. Abstraction function $f_a(j) = null$ if a^j produces `null` and *not_null* otherwise. Based on the dependence graph, the analysis traverses backward from node a^{null} where $a \in \mathcal{I}$ is the instruction whose execution causes the `NullPointerException`. The node that is annotated with *null* and that does not have incoming edges represents the instruction that created the `null` value originally. Figure 4.2(a) shows an example of this analysis. Annotation `nn` denotes *not_null*. A `NullPointerException` is thrown when line 4 is reached.

Recording Typestate History Proposed in QVM [9], this analysis tracks the typestates of the specified objects and records the history of the state changes. When the typestate protocol of an object is violated, it provides the programmer with the recorded history. Instead of recording every single event in the trace, a summarization approach is employed to merge these events into DFAs. We show how this analysis can be formulated as an abstract slicing problem, and the DFAs can be easily derived from the dependence graph.

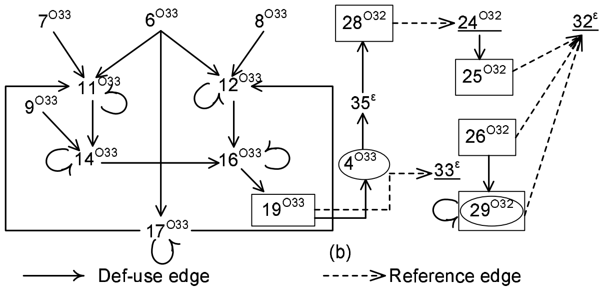
Domain $\mathcal{D}$ is $\mathcal{O} \times \mathcal{S}$, where $\mathcal{O}$ is a specified set of allocation sites (whose objects need to be tracked) and $\mathcal{S}$ is a set of predefined states $s_0, s_1, \ldots, s_n$ of the objects created by the allocation sites in $\mathcal{O}$. Abstraction function $f_a(j) = (\texttt{alloc}(a^j), \texttt{state}(a^j))$ if instruction instance a^j invokes a method on an object $\in \mathcal{O}$, and the method can cause the object to change its state. The function is undefined otherwise (i.e., all other instructions are not tracked). Here `alloc` is a function that returns the allocation site of the receiver object at a^j, and function `state` returns the state of this object immediately before a^j. The state can be stored as a tag of the object, and updated when a method is invoked on this object.

An example is shown in Figure 4.2(b). Consider the object O_1 created at line 1, with states 'u' (uninitialized), 'oe' (opened and empty), 'on' (opened but not empty), and 'c' (closed). Arrows with dashed lines denote the "next-event" relationships. These relationships are added to the graph for constructing the DFA described in [9], and they can be easily obtained by memorizing the last event on each tracked object. When line 10 is executed, the typestate protocol is violated because the file is read after it is closed. The programmer can easily identify the problem when she inspects the graph and finds that line 10 is executed on a closed file. While the example shown in Figure 4.2(b) is not strictly a dependence graph, the "next-event" edges can be conceptually thought of as def-use edges among nodes that write and read the object state tag.

Extended Copy Profiling Work described in Chapter 3 [156] describes how to profile copy chains that represent the transfer of the same data without any computation. Nodes in a copy chain are fields of objects represented by their allocation sites (e.g., $O_i.f$). An edge connects two field nodes, abstracting away intermediate stack copies. Each stack variable has a shadow stack location, which records the field from which its value originated. An extended version of this analysis is to include intermediate stack nodes along copy chains, because they are important for understanding the methods through which the values are transferred.

```
1 class A{
2   int t = 0;
3   int getTotal(){
4       return this.t; }
5   void foo(int n){
6     int i = 0;
7     int a = 0;
8     int b = 0;
9     int total = 0;
10    if(i < n){
11       a = a + i;
12       b = b * i;
13       if(i < 500){
14          total = total + a;
15       else
16          total = total + b;
17       i = i + 1;
18       goto 10; }
19    this. t = total; }}
20 class IntList{
21   int[] arr;
22   int size;
23   List() {
24      int[] tmp = new int[100];
25      this.arr = tmp;
26      size = 0; }
27   void add(int i){
28      arr[size] = i;
29      this.size = this.size + 1;
30 }}
31 void main(){
32   IntList l = new IntList();
33   A o = new A();
34   o.foo(1000);
35   int s = o.getTotal();
36   l.add(s);
37   }
38 }
```

(a)



Node	*Freq*	*AC*		*Node*	*Freq*	*AC*
6^{O33}	1	1		12^{O33}	1000	2002
7^{O33}	1	1		14^{O33}	500	2503
8^{O33}	1	1		16^{O33}	500	4004
9^{O33}	1	1		17^{O33}	1000	1001
11^{O33}	1000	2002		19^{O33}	1	4005

(c)

Node	*1-RAC*	*1-RAB*	*2-RAC*	*2-RAB*
O_{24}^{O32}	2	0	2	0
O_{32}^{ε}	4	1	6	1
O_{33}^{ε}	4005	4	4005	4

(d)

Figure 4.3: (a) Code example. (b) Corresponding dependence graph G_{cost}; nodes in boxes/circles write/read heap locations; underlined nodes create objects. (c) Nodes in method `A.foo` (*Node*), their frequencies (*Freq*), and their abstract costs (AC); (d) Relative abstract costs i-RAC and benefits i-RAB for the three allocation sites; i is the level of reference edges considered.

Now we show that this complex dynamic analysis can also be instantiated in our framework using abstract slicing. The key to the analysis is the ability of distinguishing instructions that access data originating from fields of different allocation sites. Hence, the abstraction domain $\mathcal{D}$ is a Cartesian set $\mathcal{O} \times \mathcal{F}$, where $\mathcal{O}$ is the domain of allocation sites, and $\mathcal{F}$ is the domain of field identifiers. We allow $\mathcal{D}$ to have a special element $\perp$, representing that the current data does not come from any field (e.g., it is a constant, a newly-created object, or a result of a computation instruction).

Abstraction function f_{cp} $(ins,\ i)$ = $\texttt{map}(\texttt{shadow}(ins^i))$, if ins is an assignment instruction; or $\perp$, otherwise. Here function `shadow` maps an instruction instance to the tracking data (i.e., an object field) contained in the shadow location of its lhs variable, and function `map` maps a field $o.g$ to its static abstraction `new` $O.g$. An example of this analysis is shown in Figure 4.2(c). For example, in order to identify the intermediate stack locations in the copy chain between locations $O_1.f$ and $O_3.f$, one can backward traverse the generated dependence graph from node $6^{O3.f}$ (that writes field f of the object created by allocation site O_3). The traversal follows nodes that have the same annotation $O_1.f$ until it reaches the node that reads the field $O_1.f$ (i.e., node $2^{O1.f}$).

Similarly to the way that a data flow analysis [116, 120] or an abstract interpreter [39] deals with static data flow, abstract slicing employs abstract domains to handle dynamic data flow problems, recognizing that it is only necessary to distinguish instruction instances that are critical to the program property that the target analysis intends to discover. Note that there are many dynamic analyses that cannot be formulated as abstract slicing problems, because they do not exhibit bounded-domain properties. For example, an analysis that automatically finds bugs by comparing traces of a successful run and a failed run has to inspect the two entire execution paths. Abstractions in the dependence graph may cause the differencing algorithm to report imprecisely the cause of the bug.

4.2.2 Cost Computation

Definition 4.2.3 *(*Absolute Cost*). Given a non-abstract thin data dependence graph G and an instruction instance a^j $(a \in \mathcal{I}, j \in \mathcal{N})$ that produces a value v, the* absolute cost *of v is the number of nodes that can reach a^j in G.*

Absolute costs are expensive to compute and it does not make much sense to present them to the programmer, unless they are aggregated in some meaningful way across instruction instances so that they can help understand the overall execution. In our approach the instructions are abstracted based on dynamic calling contexts. The contexts are represented with object sensitivity [91], which is well suited for modeling of object-oriented data structures.

A calling context is represented by a chain of static abstractions (i.e., allocation sites $O_i \in \mathcal{O}$) of the receiver objects for the invocations on the call stack. Domain $\mathcal{D}_{cost}$ contains all possible chains of allocation sites. Abstraction function $f_a(j) =$ `objCon`$(\texttt{cs}(a^j))$, where function `cs` takes a snapshot of the call stack when a^j is executed, and function `objCon` maps this snapshot to the corresponding chain of allocation sites O_i for the run-time receiver objects. $\mathcal{D}_{cost}$ is not finite in the presence of recursion, and even for a recursion-free program its size is exponential. We limit the size of $\mathcal{D}_{cost}$ further to be a fixed number s (short for "slots"), specified by the user as a parameter of the profiling tool. Now the domain is simply the set of integers 0 to $s-1$. An encoding function `h` is used to map an allocation site chain to such an integer; the description of `h` will be presented shortly. With this approach, the amount of memory required for the analysis is linear in program size.

Each node in the dependence graph is annotated with an integer, representing the execution frequency of the node. Based on these frequencies, an *abstract cost* for each node can be computed as an approximation of the total costs of values produced by the instruction instances represented by the node.

Definition 4.2.4 (Abstract Cost). *Given a dependence graph G_{cost}, the abstract cost of a node n^k is $\Sigma_{a^j | a^j \rightsquigarrow n^k}$ freq(a^j), where $a^j \rightsquigarrow n^k$ if there is a path from a^j to n^k in G_{cost}, or $a^j = n^k$.*

Example Figure 4.3 shows a code example and its dependence graph for cost computation. While some statements (line 29) may correspond to multiple bytecode instructions, they are still considered to have unit costs. These statements are shown for illustration purposes and will be broken into multiple ones by our tool.

All nodes are annotated with their object contexts (i.e., elements of $\mathcal{D}_{cost}$). For ease of understanding, the contexts are shown in their original forms, and the tool actually uses the encoded forms (through function `h`). Nodes in boxes represent instructions that write heap locations. Dashed arrows represent reference edges; these edges can be ignored for now. The table shown in part (c) lists nodes for the execution of method `A.foo` (invoked by the call site at line 34), their frequencies, and their abstract costs.

The abstract cost of a node computed by this approach may be larger than the exact sum of absolute costs of the values produced by the instruction instances represented by the node. This is because for a node a such that $a \leadsto n$, there may not exist any dependences between some instruction instances of a and some instruction instances of n. This difference can be large when the abstract cost is computed after traversing long dependence graph paths, and the imprecision gets magnified. More importantly, this cost represents the *cumulative effort* that has been made from the very beginning of the execution to produce the values. It may still not make much sense for the programmer to diagnose problems using abstract costs, as it is almost certain that nodes representing instructions executed later will have larger costs than those representing instructions executed earlier. In Section 4.3, we address this problem by computing a *relative abstract cost*, which measures execution bloat at the object level by traversing dependence graph paths connecting nodes that read and write object fields.

Special nodes and edges in G_{cost} To measure execution bloat, we augment the graph with two special kinds of nodes: *predicate* nodes and *native* nodes, both representing the consumption of data. A predicate node is created for each `if` statement, and a native node is created for each call site that invokes a native method. These nodes do not have associated contexts. In addition, we mark nodes that allocate objects (underlined in Figure 4.3 (b)), that read heap locations (nodes in circles), and that write heap locations (nodes in boxes). These nodes are later used to identify object structures.

Reference edges are used to represent reference relationships. For each heap store $a.f = b$, a reference edge is created to connect the node representing this store (i.e., a boxed node) and the node allocating the object that flows to a (i.e., an underlined node). For example, there exists a reference edge from $28^{O_{32}}$ to $24^{O_{32}}$, because $24^{O_{32}}$ allocates the array object and $28^{O_{32}}$ stores an integer to the array (which is similar to writing an object field). These edges will be used to aggregate costs for individual heap locations to form costs for objects and data structures.

4.2.3 Construction of G_{cost}

Selecting encoding function h There are two steps in mapping an allocation site chain to an integer $d \in \mathcal{D}_{cost}$ (i.e., $[0, \ldots, s-1]$). The first step is to encode the chain into a *probabilistically unique value* that will accurately represent the original object context chain. An encoding function proposed in [21] is adapted to perform this mapping: $\mathsf{g}_i = 3 * \mathsf{g}_{i-1} + o_i$, where o_i is the i-th allocation site ID in the chain and g_{i-1} is the probabilistic context value computed for the chain prefix with length $i-1$. While simple, this function exhibits very small context conflict rate, as demonstrated in [21]. In the second step, this encoded value is mapped to an integer in the range $[0, \ldots, s-1]$ using a simple `mod` operation.

Profiling for constructing G_{cost} Instead of recording the full execution trace and building a dependence graph offline, we use an online approach that combines dynamic flow tracking and slicing. The key issue is to identify the data dependences online, which can be done using shadow locations [99, 102]. For each location l, a shadow location l' contains the address of the dependence graph node representing the instruction instance that wrote the last value of l. When an instruction is executed, the node n to which this instruction instance is mapped is retrieved, and all nodes m_i that last wrote the locations read by the instruction are identified. Edges are then added between n and each m_i.

For a local variable, its shadow location is a new local variable on the stack. For heap locations we use a *shadow heap* [102] that has the same size as the Java heap. To enable quick access, there is a predefined distance *dist* between the starting addresses of these two heaps. For a heap location l, the address of l' can be quickly obtained as l + *dist*.

A *tracking stack* is maintained in parallel with the call stack to pass data dependence relationships across calls. For each invocation, the tracking stack also passes the receiver object chain for the caller. The next context is obtained by concatenating the caller's chain and the allocation site of `this`.

Instrumentation semantics Figure 4.4 shows a list of inference rules defining the instrumentation semantics. Each rule is of the form $\mathsf{V}, \mathsf{E}, \mathsf{H}, \mathsf{S}, \mathsf{P}, \mathsf{T} \Rightarrow^{a:i=\ldots} \mathsf{V}', \mathsf{E}', \mathsf{H}', \mathsf{S}', \mathsf{P}', \mathsf{T}'$ with unprimed and primed symbols representing the state before and after the execution of statement a. In cases where a set does not change (e.g.,

Assign

$$\frac{\begin{array}{l}\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\mathtt{h}(c)} \rhd \mathsf{S}(k)\}\end{array}}{\mathsf{V}, \mathsf{E}, \mathsf{S} \Rightarrow^{a:i=k} \mathsf{V}', \mathsf{E}', \mathsf{S}'}$$

Computation

$$\frac{\begin{array}{l}\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\mathtt{h}(c)} \rhd \mathsf{S}(k)\} \cup \{a^{\mathtt{h}(c)} \rhd \mathsf{S}(l)\}\end{array}}{\mathsf{V}, \mathsf{E}, \mathsf{S} \Rightarrow^{a:i=k \oplus l} \mathsf{V}', \mathsf{E}', \mathsf{S}'}$$

Predicate

$$\frac{\begin{array}{l}\mathsf{V}' = \mathsf{V} \cup \{a^{\epsilon}\} \quad \mathsf{S}' = \mathsf{S} \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\epsilon} \rhd \mathsf{S}(i)\} \cup \{a^{\epsilon} \rhd \mathsf{S}(k)\}\end{array}}{\mathsf{V}, \mathsf{E}, \mathsf{S} \Rightarrow^{a:if\ (i>k)\{\ldots\}} \mathsf{V}', \mathsf{E}', \mathsf{S}'}$$

Load Static

$$\frac{\begin{array}{l}\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\mathtt{h}(c)} \rhd \mathsf{S}(A.f)\}\end{array}}{\mathsf{V}, \mathsf{E}, \mathsf{S} \Rightarrow^{a:i=A.f} \mathsf{V}', \mathsf{E}', \mathsf{S}'}$$

Store Static

$$\frac{\begin{array}{l}\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[A.f \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\mathtt{h}(c)} \rhd \mathsf{S}(i)\}\end{array}}{\mathsf{V}, \mathsf{E}, \mathsf{S} \Rightarrow^{a:A.f=i} \mathsf{V}', \mathsf{E}', \mathsf{S}'}$$

Alloc

$$\frac{\begin{array}{l}\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{H}' = \mathsf{H}[a^{\mathtt{h}(c)} \mapsto ('U', (new\ X)^{\mathtt{h}(c)}, '\ ')] \\ \mathsf{P}' = \mathsf{P}[o \mapsto (new\ X)^{\mathtt{h}(c)}]\end{array}}{\mathsf{V}, \mathsf{H}, \mathsf{S}, \mathsf{P} \Rightarrow^{a:i=new\ X} \mathsf{V}', \mathsf{H}', \mathsf{S}', \mathsf{P}'}$$

Load Field

$$\frac{\begin{array}{l}\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\mathtt{h}(c)} \rhd \mathsf{S}(o_v.f)\} \\ \mathsf{H}' = \mathsf{H}[a^{\mathtt{h}(c)} \mapsto ('C', \mathsf{P}(o_v), f)]\end{array}}{\mathsf{V}, \mathsf{E}, \mathsf{H}, \mathsf{S} \Rightarrow^{a:i=v.f} \mathsf{V}', \mathsf{E}', \mathsf{H}', \mathsf{S}'}$$

Store Field

$$\frac{\begin{array}{l}\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[o_v.f \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\mathtt{h}(c)} \rhd \mathsf{S}(i)\} \\ \mathsf{H}' = \mathsf{H}[a^{\mathtt{h}(c)} \mapsto ('B', \mathsf{P}(o_v), f)]\end{array}}{\mathsf{V}, \mathsf{E}, \mathsf{H}, \mathsf{S} \Rightarrow^{a:v.f=i} \mathsf{V}', \mathsf{E}', \mathsf{H}', \mathsf{S}'}$$

Method Entry

$$\frac{\begin{array}{l}\mathsf{S}' = \mathsf{S}[t_i \mapsto \mathsf{T}(i)] \text{ for } 1 \leq i \leq n \\ \mathsf{T}' = (\mathsf{T}(n+1) \circ \textsc{AllocId}(\mathsf{P}(o_{this})), \mathsf{T}(n+1), \mathsf{T}(n+2), \ldots)\end{array}}{\mathsf{S}, \mathsf{T} \Rightarrow^{a:m(t_1, t_2, \ldots, t_n)} \mathsf{S}', \mathsf{T}'}$$

Return

$$\frac{\mathsf{T}' = (\mathsf{S}(i), \mathsf{T}(2), \mathsf{T}(3), \ldots)}{\mathsf{T} \Rightarrow^{a:return\ i} \mathsf{T}'}$$

Figure 4.4: Inference rules defining the run-time effects of instrumentation.

when $\mathsf{S} = \mathsf{S}'$), it is omitted. Node domain V contains nodes of the form $a^{h(c)}$, where a denotes the instruction and $h(c)$ denotes the encoded integer of the object context c. Edge domain $\mathsf{E} : \mathsf{V} \times \mathsf{V}$ is a relation containing dependence relationships of the form $a^l \rhd k^n$, which represents that an instance of a abstracted as a^l is data dependent on an instance of k abstracted as k^n. Shadow environment $\mathsf{S} : \mathsf{M} \rightarrow \mathsf{V}$ maps a run-time storage location to the content in its corresponding shadow location (i.e., to its tracking data). Here M is the domain of memory locations. For each location, its shadow location contains the (address of the) node that performs the most recent write to this location. Rules Assign, Computation, Predicate, Load Static, and Store Static update the environments in expected ways. In rule Predicate, instruction instances are not distinguished and the node is represented by a^{ϵ}.

Rules Alloc, Load Field and Store Field additionally update heap effect environment H, which is used to construct reference edges in G_{cost}. $\mathsf{H} : \mathsf{V} \rightarrow \mathsf{Z}$ maps a node $a^l \in \mathsf{V}$ to a heap effect triple (*type*, *alloc*, *field*) $\in$ domain Z of heap effects. Here, *type* can be $'U'$ (i.e., underlined) representing the allocation of an object, $'B'$ (i.e., boxed) representing a field store, or $'C'$ (i.e., circled) representing a field load. Elements *alloc* and *field* denote the object and the field on which the effect occurs. For instance, triple $('U', O, '\ ')$ means that a node contains an allocation site O, while triple $('B', O, f)$ means that a node writes to field f of an object created by allocation site O. A reference edge can be added between a (store) node with effect $('B', O, *)$ and another (allocation) node with effect $('U', O, '\ ')$, where $*$ represents any field name. In order to perform this matching, we need to provide access to the allocation site ID for each run-time object. This is done using tag environment P that maps a run-time object to its allocation site ID.

However, the reference edge could be spurious if the store node and the allocation node are connected using only allocation site ID O, because the two effects (i.e., $'B'$ and $'U'$) could occur on different instances created by O. To improve the precision of the client analyses, object context is used again to annotate allocation sites. For example, in rule Alloc, H is updated with effect triple $('U', (\textit{new } X)^{\mathtt{h}(c)}, '\ ')$, where the allocation site *new X* is annotated with the encoded context integer $\mathtt{h}(c)$. This triple matches only (store) node with effect $('B', (\textit{new } X)^{\mathtt{h}(c)}, *)$, and many spurious reference edges can thus be eliminated. In rule Alloc, $(\textit{new } X)^{\mathtt{h}(c)}$ is used to tag the newly-created run-time object o (by updating tag environment P), and this information will be retrieved later when o is dereferenced. In rules LOAD FIELD and STORE FIELD, o_v denotes the run-time object that variable v points to. $\mathsf{P}(o_v)$ is used to retrieve the allocation site (annotated with the context) of o_v, which is previously set as o_v's tag upon its allocation.

The last two rules show the instrumentation semantics at the entry and the return site of a method, respectively. At the entry of a method with n parameters, tracking stack T contains the tracking data for the actual parameters of the call, as the n top elements $\mathsf{T}(1), \ldots, \mathsf{T}(n)$, followed by the receiver object chain for the caller of the method (as element $\mathsf{T}(n+1)$). In rule METHOD ENTRY, the tracking data for a formal parameter t_i is updated with the tracking data for the corresponding actual parameter (stored in $\mathsf{T}(i)$). The new object context is computed by applying concatenation operator $\circ$ to the old chain $\mathsf{T}(n+1)$ and the allocation site of the run-time receiver object o_{this} pointed to by *this* (or an empty string if the current method is static). Function ALLOCID removes the context annotation from the tag of o_{this}, leaving only the allocation site ID. The stack is updated by removing the tracking data for the actuals, and storing the new context on the top of the stack. This new context is available for use by all rules applied in the body of the method (denoted by c in those rules). At the return site, T is updated to remove the current context and to store the tracking data for the return variable i.

The rule for call sites is not shown in Figure 4.4, as it requires splitting a call site into a *call* part and a *return* part, and reasoning about both of them. Immediately before the call, the tracking data for the actual parameters is pushed on tracking stack T. Immediately after the call, the tracking data for the returned value is popped from

T and used to update the dependence graph and the shadow location for the left-hand-side variable at the call site. If the method invoked at the call site is a native method, we create a node (without context) for it, and add edges between each node contained in the shadow locations of the actual parameters and this node, representing that the values of parameters are consumed by this native method.

Implementation of P A natural idea of implementing object tagging is to save the tag in the header of each object (i.e., the header usually has unused space). However, in the J9 VM that we use, this 64-bit header cannot be modified. To solve this problem, the corresponding 64 bits on the shadow heap are used to store the object tag. Hence, although environments P and S have different mathematical meanings, both are implemented using shadow locations.

4.3 Relative Object Cost-Benefit Analysis

This section describes a novel diagnosis technique that identifies data structures with high cost-benefit rates. As discussed in Section 4.4, this analysis effectively uncovers significant optimization opportunities in six large real-world applications. We propose to compute a *relative abstract cost* for an object, which measures the effort of constructing the object from data already available in fields of other objects (rather than the cumulative effort from the beginning of the execution). Similarly, we compute a *relative abstract benefit* for an object, which explains how the data contained in the object is used to construct other objects. These metrics can help a programmer pinpoint specific objects that are expensive to construct (e.g., there are large costs of computing the data being written into this object) but are not very useful (e.g, the only use of this object is to make a clone of it and then invoke methods on the clone).

We first develop an *object cost-benefit analysis* that aggregates relative costs and benefits for individual fields of an object in order to compute the cost and benefit for the object itself. Next, the cost and benefit for a *higher-level data structure* is obtained in a similar manner, by gathering costs and benefits of lower-level objects/data structures accessible through reference edges.

4.3.1 Analysis Algorithm

Definition 4.3.1 (Relative Abstract Cost). *Given G_{cost}, the heap-relative abstract cost (HRAC) of a node n^k is $\Sigma_{a^j | a^j \rightharpoonup n^k}$ freq(a^j), where $a^j \rightharpoonup n^k$ if $a^j \leadsto n^k$ and there exists a path from a^j to n^k such that no node on the path reads from a static or object field. The relative abstract cost (RAC) for an object field represented by $O^d.f$ is the average HRAC of store nodes n^k that write to $O^d.f$.*

Consider the entire flow of a piece of data (from the input of the program to its output) during the execution. This flow consists of multiple hops of data transformations among heap locations. Each hop performs the following three steps: reading values from heap locations, performing stack copies and computations on them, and writing the results to other heap locations. Consider one single hop with multiple sources and one target along the flow, which reads values from heap locations $l_1, l_2, \ldots, l_n$, transforms them to produce a new value, and writes it back to heap location l'. The RAC of l' measures the amount of work needed (on the stack) to complete this hop of transformations.

The computation of HRAC for a node n^k requires a backward traversal from n^k, which finds all nodes on the paths between each heap-reading node and n^k, and calculates the sum of their frequencies. For example, the HRAC for node 35^ϵ is only 1 (instead of 4007), because the node depends directly on a node (i.e., $4^{O_{33}}$) that reads heap location `this.t`. The RAC for a heap location is the average HRAC of the nodes that can write this location. For example, the RAC for $O_{33}^{\epsilon}.t$ is the HRAC for $19^{O_{33}}$, which is 4005. The RAC for $O_{24}^{O_{32}}.ELM$ (i.e., the elements of the array object) is 2, which equals the HRAC of node $28^{O_{32}}$ that writes this field.

Definition 4.3.2 (Relative Abstract Benefit). *Given G_{cost}, the heap-relative abstract benefit (HRAB) of a node n^k is $\Sigma_{a^j | n^k \rightharpoondown a^j}$ freq(a^j), where $n^k \rightharpoondown a^j$ if $n^k \rightsquigarrow a^j$ and there exists a path from n^k to a^j such that no node on the path writes to a static or object field. The relative abstract benefit (RAB) for an object field represented by $O^d.f$ is the average HRAB of load nodes n^k that read from $O^d.f$.*

Symmetric to the definition of RAC that focuses on how a heap value is *produced*, the RAB for l explains how a heap value is *consumed*. Consider again one single hop (but with one source and multiple targets) along the flow, which reads a value from location l, transforms this value (together with values read from other locations), and writes the results to a set of other heap locations $l'_1, l'_2, \ldots, l'_n$. The RAB of l measures the amount of work performed (on the stack) to complete this hop of transformations. For example, the RAB for $O_{33}^{\epsilon}.t$ is the HRAB of node $4^{O_{33}}$ that reads this field, which is 2 (because the relevant nodes a^j are only $4^{O_{33}}$ and 35^ϵ). Figure 4.5 (a) illustrates the computation of RAC and RAB.

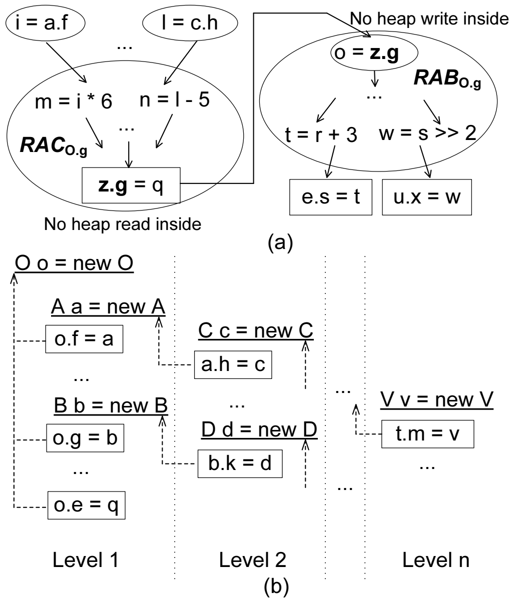


Figure 4.5: (a) Relative abstract cost and benefit. Nodes considered in computing RAC and RAB for $O.g$ (where O is the allocation site for the object referenced by z) are included in the two circles, respectively; (b) Illustration of n-RAC and n-RAB for the object created by $o = new\ O$; dashed arrows are reference edges.

This definition of benefit captures both the frequency and the complexity of data use. First, the more target heap values that the value read from l is used to (transitively) produce, the larger benefit location l can have for the construction of these other objects. Second, the more effort is made to transform the value from l to other heap values, the larger benefit l can have. This is because the purpose of writing a value into a heap location is, intuitively, to keep the value so that it can be reused later and the (heavy) cost of re-computing it can be avoided. Whether to store a value in a heap location is essentially a decision involving space-time tradeoffs. If l's value v can be easily converted to some other value v' and v' is immediately stored in another heap location (i.e., little computation performed), the benefit of keeping v in l becomes less obvious, since v and v' may differ slightly and it may not be necessary to use two different heap locations to cache them. In the extreme case where v' is simply a copy of v, the RAB for l is 1 and storing v is not desirable at all if the RAC for l is large. Special treatment is applied to consumer nodes: we assign a large RAB to a heap location if the value it contains can flow to a predicate or a native node. This means the value contributes to control decision making or is used by the JVM, and thus benefits the overall execution.

Definition 4.3.3 (n-RAC and n-RAB). *Consider an object reference tree RT_n of height n rooted at O^d. The n-RAC for O^d is the sum of the RACs for all fields $O_i^k.f$, such that both O_i^k and the object $O_i^k.f$ points to are in RT_n. Similarly, the n-RAB for O^d is the sum of the RABs for all such fields $O_i^k.f$.*

The object reference (points-to) tree can be constructed by using reference edges in the dependence graph, and by removing cycles and nodes more than n reference edges away from O^d. We aggregate the RACs and RABs for individual fields through the tree edges to form the RACs and RABs for objects (when $n = 1$) and high-level data structures (when $n > 1$). Figure 4.5 (b) illustrates n-RAC and n-RAB for an object created by $o = new\ O$. The n-RAC(RAB) for this object includes the RAC(RAB) of each field written by a boxed node (i.e., heap store) shown in the figure. For all case studies and experiments, $n = 4$ was used as this is the reference chain length for the most complex container classes in the Java collection framework (i.e., `HashSet`).

Table (d) in Figure 4.3 shows examples of 1- and 2- RACs and RABs. Both the 1-RAB and the 2-RAB for $O_{24}^{O_{32}}$ are 0, because the array element is never used in the code. Objects O_{32}^{ϵ} and O_{33}^{ϵ} have large cost-benefit rates, which indicates the existence of wasteful operations. This is indeed the case in this example: for O_{32}^{ϵ}, there is an element added but never retrieved; for O_{33}^{ϵ}, there is a large cost of computing the value

```
class ClasspathDirectory{
   boolean isPackage(String packageName){
        return directoryList(packageName) != null;
   }

   List directoryList(String packageName){
       List ret = new ArrayList();  /*problematic*/
       //try to find all the files in the dir packageName
       //if nothing is found, set ret to null
       …
       return ret;
   }
}
```

Figure 4.6: Real-world example that our analysis found in `eclipse`.

stored in its field `t`, and the value is copied to another heap location (in `IntList`) immediately after it is calculated. The creation of object O_{33}^{ϵ} is not beneficial at all because this value could have been stored directly to the array.

Finding bloat Several usage scenarios are intended for this cost-benefit analysis. First, it can find long-lived objects that are written much more frequently than being read. Second, it can find containers that contain many more objects than they should. These containers are often the sources of memory leaks. The analysis can find that they have large RAC/RAB rates because few elements are retrieved and assigned to other heap locations. Third, it can find allocation sites that create large volumes of temporary (short-lived) objects. These objects are often created simply to carry data across method invocations. Data that is computed and written into them is read somewhere else and assigned to other object fields. This simple use of the data causes these objects to have large cost-benefit rates. The next section shows that our tool finds all three categories of problems in real-world Java applications.

Real-world example Figure 4.6 shows a real-world example that illustrates how our analysis works. An object with high costs and low benefits is highlighted in the figure. The code in the example is extracted from `eclipse` 3.1.2, a popular Java development tool. Method `isPackage` returns true/false based on whether the given package name corresponds to an actual Java package. This method is implemented by calling (reusing) `directoryList` which invokes many other methods to compute a list of files and directories under the package specified by the parameter. `isPackage` then returns whether the list computed by `directoryList` is `null`. While the reference to list `ret` is used in a predicate, its fields are not read and do not participate in computations. Hence, when the RACs and RABs for its fields are aggregated based on the object hierarchy, the imbalance between the cost and benefit for the entire `List` data structure can be seen. To optimize this case, we created a specialized version of `directoryList`, which returns immediately when the package corresponding to the given name is found.

4.3.2 Comparison with Other Design Choices

Cost/benefit for computation vs cost/benefit for cache Note that the relative cost and benefit for an object are essentially measured in terms of the *computations* that produce values written into the object. The goal of this analysis is to find objects such that they contain (relatively) useless values and these values are produced by (relatively) expensive operations. Upon identifying these operations, the user may find more efficient ways to achieve the same functionality. This is orthogonal to measuring the usefulness of a data structure as a *cache*, where the cost of the cache should include only the instructions executed to create the data structure itself (i.e., without the cost of computing the values being cached) and the benefit should be (re-)defined as a function of the amount of work cached and the number of times the cached values are used. It would be interesting to investigate, in future work, how these new definitions of cost and benefit can be used to find inappropriately-used caches.

Single-hop cost/benefit vs multi-hop cost/benefit The analysis limits the scope of tracked data flow to one single hop—that is, reading data from the heap, transforming it through stack locations, and writing the results back to the heap. While this design choice can produce easy-to-understand reports, it could miss problematic data structures because of its "short-sightedness". For example, our tool may consider a piece of data that is ultimately-dead to be appropriately used, because it is indeed involved in complex computations within the one hop seen by the analysis. To alleviate this problem, we have developed an additional analysis, based on G_{cost}, which identifies computations that can reach ultimately-dead values. Section 6.5 presents measurements of redundant computations based on this analysis.

A different way of handling this issue is to consider multiple hops when computing costs and benefits based on graph G_{cost}, so that more detailed information about data production and consumption can be obtained. For example, costs and benefits for an instruction can be recomputed by traversing multiple heap-to-heap hops on G_{cost} backward and forward, respectively, starting from the instruction. Of course, extending the inspected region of the data flow would make the report hard to verify as the programmer has to inspect larger program scopes to understand the detected problems. In future work, it would be interesting to compare empirically problems found using different scope lengths, and to design particular tradeoffs between the scope length considered and the difficulty of explaining the report.

Considering vs ignoring control decision making Our analysis does not consider the effort of making control decisions as part of the costs of computing values under these decisions. The major reason is that by doing so we could potentially include the costs of computing many values that are irrelevant to the value of interest into the cost of that value, leading to imprecise and hard-to-understand reports. However, ignoring this effort of control decision making could lead to information loss. For example, the tool may miss problematic objects due to the underestimation of the cost of constructing them. In future work, we will also be interested in accounting for this effort, and investigating the relationship between the scope of control flow decisions considered (e.g., the closest n predicates on which an instruction is control-dependent) and the usefulness of the analysis output.

Other analyses Graph G_{cost} (annotated with other information) can be used as basis for discovering a variety of performance-related program properties. For example, we have implemented a few clients that can answer specific performance-related queries. These clients include an analysis that computes method-level costs (i.e., the cost of producing the return value of a method relative to its inputs), an analysis that detects locations that are re-written before being read, an analysis that identifies nodes producing always-true or always-false predicate conditions, and an analysis that searches for problematic collections by ranking collection objects based on their RAC/RAB rates. While these analyses are currently implemented inside a JVM, they could be easily migrated to an offline heap analysis tool that provides user-friendly interfaces and improved usability (i.e., the JVM only needs to write G_{cost} to external storage).

4.4 Evaluation

We have performed a variety of studies with our technique using the DaCapo benchmark set [15], which contains 11 programs in its original version (from `antlr` to `eclipse` in Table 4.1 and Table 4.2) and an additional set of 7 programs in its new release (from `avrora` to `tradesoap`). We were able to run our tool on all these 18 large programs, including both client and server applications. 16 programs (except `tradesoap` and `tradebeans`) were executed with their large workloads. `tradesoap` and `tradebeans` were run with their default workloads, because these two benchmarks were not stable enough and running them with large workloads can fail even without our tool. All experiments were conducted on a 1.99GHz Dual Core machine. The evaluation has several components: cost graph characteristics, evaluation of the time and space overhead of the tool, the measurements of bloat based on nodes producing dead values, and six case studies that describe problems found by the tool in real applications.

4.4.1 G_{cost} Characteristics and Bloat Measurement

Parts (a) and (b) in Table 4.1 report, for two different values of s (the number of slots for each object used to represent context), the numbers of nodes and edges in G_{cost}, as well as the space overheads and the time overheads of the tool. Note that all programs can successfully execute when we increase s to 32, while the offline traversal of the graph (to generate statistics) can make the tool run out of memory for some large programs. The space overhead does not include the size of shadow heap, which

Program	(a) $s = 8$					(b) $s = 16$				
	$\#N$(K)	$\#E$(K)	M(Mb)	$O(\times)$	CR(%)	$\#N$	$\#E$	M	O	CR
antlr	183	689	10.2	82	0.066	355	949	16.1	77	0.041
bloat	201	434	9.8	78	0.089	396	914	17.4	76	0.051
chart	288	306	13.2	76	0.068	567	453	22.6	76	0.047
fop	195	120	8.4	45	0.067	381	162	14.0	46	0.045
pmd	184	187	8.0	55	0.075	365	313	13.6	96	0.052
jython	288	275	12.6	28	0.065	666	539	26.1	27	0.042
xalan	168	594	8.5	75	0.066	407	1095	18.1	74	0.044
hsqldb	192	110	8.0	88	0.072	379	132	13.7	86	0.050
luindex	160	177	6.7	92	0.073	315	331	11.5	86	0.040
lusearch	139	110	5.5	48	0.079	275	223	11.0	52	0.053
eclipse	525	2435	28.8	47	0.072	1016	5724	53.1	53	0.047
avrora	189	108	7.9	67	0.086	330	125	11.2	56	0.034
batik	361	355	15.8	85	0.086	662	614	24.9	89	0.049
derby	308	314	13.9	63	0.080	425	530	22.1	57	0.049
sunflow	206	152	8.2	92	0.076	330	212	10.3	91	0.040
tomcat	533	1100	25.4	94	0.098	730	2209	48.6	92	0.063
tradebeans	825	1010	38.2	89/8*	0.053	1568	1925	58.9	82/8*	0.036
tradesoap	860	1370	41	82/17*	0.062	1628	2536	63.6	81/16*	0.040

Table 4.1: Characteristics of G_{cost} (I). Reported are the numbers (in thousand) of nodes (N) and edges (E), the memory overhead (in megabytes) excluding the size of the shadow heap (M), the running time overhead (O), and the context conflict ratio (CR).

is 500Mb for all programs. Note that the shadow heap is *not* compulsory for using our technique. For example, it can be replaced by a global hash table that maps each object to its tracking data (and an object entry is removed when the object is garbage collected). The choice of shadow heap in our work is just to allow quick access to the tracking information. When the number of context slots s grows from 8 to 16, the space overhead increases while the running time is almost not affected. The instrumentation significantly increases the running times (i.e., 71$\times$ slowdown on average for $s = 8$ and 72$\times$ for $s = 16$ when the whole-program tracking is enabled). This is because (1) G_{cost} is updated at each instruction instance and (2) the creation of G_{cost} nodes and edges needs to be synchronized to guarantee that the tool is race-free. It was an intentional decision *not* to focus on the performance of the profiling, but instead to focus on the collected information and on demonstrating that the results

Program	(c) Bloat measurement for $s = 16$			
	$\#I$(B)	*IPD*(%)	*IPP*(%)	*NLD*(%)
antlr	4.9	3.7	96.2	17.5
bloat	91.2	26.9	69.9	19.3
chart	9.4	8.0	91.7	30.0
fop	0.2	28.8	60.9	30.5
pmd	5.6	7.5	92.1	27.0
jython	14.6	13.1	81.9	26.8
xalan	25.5	17.8	82.0	19.4
hsqldb	1.3	6.4	92.4	31.0
luindex	3.5	4.6	93.0	24.6
lusearch	9.1	9.3	65.2	29.1
eclipse	28.6	21.0	78.3	22.0
avrora	3.3	3.2	94.8	34.5
batik	2.4	27.1	71.1	26.7
derby	65.2	5.0	94.0	23.7
sunflow	82.5	32.7	43.7	31.7
tomcat	29.1	24.2	72.2	23.1
tradebeans	15.1	14.9	80.0	22.3
tradesoap	41.0	24.5	59.4	20.1

Table 4.2: Characteristics of G_{cost} (II). This table reports the total number (in billion) of instruction instances (I), the percentages of instruction instances (directly and transitively) producing values that are ultimately dead (*IPD*), the percentages of instruction instances (directly or transitively) producing values that end up only in predicates (*IPP*), and the percentages of G_{cost} nodes such that all the instruction instances represented by these nodes produce ultimately-dead values (*NLD*).

are useful for finding bloat in real-world programs. One effective way of reducing overhead is to choose only relevant components to track. For example, for the two transaction-based applications `tradebeans` and `tradesoap`, there is 5-10× overhead reduction when we enable tracking only for the load runs (i.e., the application is not tracked for the server startup and shutdown phases). Hence, it is possible for a programmer to identify suspicious program components using lightweight profiling tools such as a method execution time profiler or an object allocation profiler, and run our tool on the selected components for detailed diagnosis. It is also possible to employ various sampling-based or static pre-processing techniques (e.g., from [166]) to reduce the dynamic effort in data collection.

A small amount of memory is required to store the graph, and this is achieved primarily by employing abstract domains. The space reduction resulting from abstract slicing can also be seen from the comparison between the number of nodes in the graph (N) and the total number of instruction instances (I), as N represents the size of the abstract domain employed in the analysis while I represents the size of the actual concrete domain that fully depends on the run-time behavior of the application. *CR* measures the degree to which distinct contexts are mapped to the same slots by our encoding function `h`. Following [156], *CR-s* for an instruction i is defined as:

$$\mathit{CR}\text{-}s(i) = \begin{cases} 0 & \max_{0 \leq j \leq s}(dc[j]) = 1 \\ \max(dc[j]) / \sum dc[j] & \text{otherwise} \end{cases}$$

where $dc[j]$ represents the number of distinct contexts that fall into context slot j. *CR* is 0 if each context slot represents at most one distinct context; *CR* is 1 if all contexts fall into the same slot. The table reports the average *CR* for all instructions in G_{cost}. Note that both *CR-8* and *CR-16* show very small numbers. This is because many methods in a program only have a small number of distinct object chains throughout the execution.

Columns *IPD* and *IPP* in part (c) of Table 4.2 report the measurements of inefficiency for $s = 16$. *IPD* represents the percentage of instruction instances that produce only dead values. Suppose D is a set of non-consumer nodes in G_{cost} that do not have any outgoing edges (i.e., no other instructions are data-dependent on them), and D^* is a set of nodes that can lead only to nodes in D. Hence, D^* contains nodes that ultimately produce only dead values. *IPD* is calculated as the ratio between the sum of execution frequencies of the nodes in D^* and the total number of instruction instances during the execution (shown in column I). Similarly, suppose P^* is the set of nodes that can lead only to predicate consumer nodes, and *IPP* is calculated as the ratio between the sum of execution frequencies of the nodes in P^* and I. Programs such as `bloat`, `eclipse` and `sunflow` have large *IPD*s, which indicates that there may exist large optimization opportunities. In fact, these three programs are the ones for which we have achieved the largest performance improvement after removing bloat (as discussed shortly in case studies). Clearly, a significant portion of the set of instruction instances is executed to produce only control flow conditions. While this does not help performance diagnosis directly, a high *IPP* indicates the program performs a large amount of comparisons-related work, which may be a sign of over-protective or over-general implementations.

Column *NLD* in part (c) reports the percentage of nodes in D^*, relative to the total number of graph nodes. The higher *NLD* a program has, the easier it is for a programmer to find problems from G_{cost}. Despite the merging of a large number of instruction instances in a single graph node, there are on average 25.5% nodes in the graph that have this property. Large performance opportunities may be found by inspecting the report to identify these wasteful operations.

4.4.2 Case Studies

We have carefully inspected the tool reports for the following six large applications: `bloat`, `eclipse`, `sunflow`, `derby`, `tomcat`, and `trade`. These applications have large code bases, and are representatives of various kinds of real-world applications, including program analysis tools (`bloat`), Java development tools (`eclipse`), image renders (`sunflow`), database servers (`derby`), servlet containers (`tomcat`), and transaction-based enterprise applications (`trade`). We have found significant optimization opportunities for unoptimized programs, such as `bloat` (37% speedup). For the other five applications that have been well maintained and tuned, the removal of the bloat detected by our tool can still result in considerable performance improvement (2%-15% speedup). More insightful changes could have been made if we were familiar with the overall design of functionality and data models. We use the DaCapo versions of these programs, because the server applications are converted to run fixed loads, and the performance can be measured simply by using running time rather than other metrics such as throughput and the number of concurrent users. It took us about 2.5 weeks to find the problems and implement the fixes for these six applications that we had never studied before.

sunflow Because it is an image rendering tool, much of its functionality is based on matrix and vector computations, such as *transpose* and *scale*. However, each such method in class Matrix and Vector starts with cloning a new Matrix or Vector object and assigns the result of the computation to the new object. Our tool reported that these newly created (short-lived) objects have extremely large unbalanced costs and benefits, as they serve primarily the purpose of carrying data across method invocations. Another few lines of the report directed us to an int array where some slots of the array are used to contain float values. These float values are converted to integers using method `Float.floatToIntBits` and assigned to the array elements. Later, the encoded integers are read from the array and converted back to float values. These operations occur in the most-frequently executed methods in the program and are therefore are expensive to perform. By eliminating unnecessary clones and bookkeeping the float values that need to be passed across method boundaries (to avoid the back-and-forth conversions), we observed 9%-15% running time reduction.

eclipse Some of the allocation sites that have the largest cost-benefit rates create objects of inner classes and Iterators, which implement visitor patterns to traverse the workspace. These visitor objects do not contain any data and are passed into iterators, where their `visit` method is invoked to process individual children elements of the workspace. However, the Iterator used here is a stack-based class that provides general functionality for traversing different types of data structures (e.g., graph, tree, etc.), while the workspace has a very simple tree structure. We replaced the visitor implementation with a worklist implementation, and this simple specialization eliminated millions of run-time objects. The second major problem found by the tool is with the hash computation implemented in a set of Hashtable classes in the JDT plugin. One of the most frequently used classes in this set is called `HashtableOfArrayToObject`, which uses arrays of objects as keys. Every time the Hashtable is expanded, its `rehash` method needs to be invoked and the hash codes of all existing entries have to be recomputed. Because the key can be a big object array, computing its hash code can trigger invocations of the `hashcode` method in many other objects, and can thus take considerably large amount of time. We created an int array field in the Hashtable class to cache the hash codes of the entries, and the recorded hash codes are used when `rehash` is executed. To conclude, by removing these high-cost-low-benefit operations, we have managed to reduce the running time by 14.5% (from 151s to 129s), and the number of objects by 2% (5.5 million).

bloat Previous work [156] has found that `bloat` suffers from excessive string creations. This finding is confirmed by our tool report. 46 allocation sites out of the top 50 that have the largest cost-benefit rates are `String` and `StringBuffer` objects created in the set of `toString` methods. Most of these objects eventually flow into methods `Assert.isTrue` and `db`, which print the strings when certain debugging-related conditions hold. However, in production runs where most bugs have been fixed, such conditions can rarely evaluate to true, and there is no benefit in constructing these objects. Another problem exposed by our tool (but not reported in [156]) is the excessive use of objects of an inner class `NodeComparator`, which contains no data but methods to compare a pair of AST nodes. The comparison starts with the given root nodes, and recursively creates `NodeComparator` objects to compare children nodes. Comparing two large trees usually requires the allocation (and garbage collection) of hundreds of objects, and such comparisons occur in almost all methods related to ASTs, even including `hashcode` and `equals`. Eliminating the unnecessary `String` and `StringBuffer` objects and replacing the visitor pattern with a breadth-first search algorithm result in 37% reduction in running time, and 68% reduction in the number of objects created.

derby The tool report shows that an int array in class `FileContainer` has large cost-benefit rates. After inspecting the code, we found it is an array containing the information of a file-based container. Every time the (same) container is written into a page, the array needs to be updated. Hence, it is written much more frequently (with the same data) than being read. To solve the problem, we modify the code to update this array only before it is read. Another set of objects that were found to have unbalanced cost-benefit rates are the strings representing IDs for different ContextManagers. These strings are used to retrieve the ContextManagers in a variety of ways, but mostly serve as HashMap keys. Because the database contexts are frequently switched, clear performance improvement can be seen when we replaced these strings with integer IDs. Eventually, the running time of the program was reduced by 6%, and the number of objects created was reduced by 8.6%.

tomcat `tomcat` is a well-tuned JSP and servlet container. There are only a few objects that have large cost-benefits according to the tool report. One set of such objects is arrays used in `util.Mapper`, representing the (sorted) list of existing contexts. Once a context is added or removed from the manager, an update algorithm is executed. The algorithm creates a new array, inserts the new context at the right position in this new array, copies the old context array to the new one, and discards the old array. To remove this bloat, we maintain only two arrays, using them back and forth as the main context list and the backing array used for the update algorithm. Another problem reported by our tool pointed to string comparisons in various `getContents` and `getProperty` methods. These methods take a property name and a `Class` object (representing the type of the property) as input, and return the value corresponding to the property using reflection. To decide the type of the property, the implementations of these methods first obtain the names of the argument classes and compare them with the embedded names such as "Integer" and "Boolean". Because a property can have only a few types, we remove such string comparisons and insert code to directly compare the `Class` objects. After the modifications, the program could run 3 seconds faster (about 2% reduction).

tradebeans `tradebeans` is an EJB application that performs database queries to simulate a stock trading process. One problem that our tool reported was with the use of `KeyBlock` and its iterators. This class represents a range of integers that will be given as IDs for the accounts and holdings when they are requested. We found that for each ID request, the class needs to perform a few redundant database queries and updates. In addition, a simple int array can suffice to represent IDs since the `KeyBlock` and the iterators are just wrappers over integers. By removing the additional database queries and using directly the int array, we have manged to make the application run 9 seconds faster (from 350s to 341s, 2.5% reduction). The number of objects created was reduced by 2.3%. DaCapo has another implementation (`tradesoap`) of `trade`, which uses the SOAP protocol to perform client-server communication and runs much slower than `tradebeans`. An interesting comparison between these two benchmarks is that the major high-cost-low-benefit objects reported for `tradesoap` are the bean objects created in the set of `convertXBean` methods. As part of the SOAP protocol, these methods perform large volumes of copies between different representations of the same bean data, resulting in significant performance slowdown.

Summary With the help of the cost-benefit analyses, we have found various performance problems in these large applications with which we do not have any experience. These problems include inefficiencies caused by common programming idioms such as visitor patterns, repeated work whose result needs to be cached (e.g., the hash code example in `eclipse`), computation of data not necessarily used (e.g., strings in `bloat`), and choices of expensive operations (e.g., string comparison in `tomcat` and the use of SOAP in `tradesoap`). For specific bloat patterns such as the use of inner classes, it is also possible for the compiler/optimizer designers to take them into account and develop optimization techniques that can remove the bloat, while not having to restrict programmers from using these patterns.

4.5 Summary and Interpretation

What is the cost of constructing this object? Is that really worth performing such a heavyweight operation? Such questions often arise during software development. They represent the most natural and explicit form in which a programmer can express her concern on performance. Tuning could be much easier if there exists tool support that allows these questions to be (even partially) automatically answered. As a step towards achieving this goal, this chapter introduces a dynamic analysis of data flow, motivated by the observation that much functionality in a large application is about propagating and transforming data. Optimizations based on control flow information (e.g., execution frequency) are insufficient to capture redundancies that accumulate during data manipulation. Our approach defines measurements of the cost of generating a piece of data, and computes an assessment of the way this data is used. This assessment is made at the level of data structures, as object-oriented data structures are extensively used and programmers often are not aware of their internal implementations.

It is interesting in future work to consider the space of other design choices discussed in Section 4.3. We are also interested in investigating future extensions and applications of abstract slicing as a general technique, so that it could potentially benefit a wider range of dynamic analyses. For example, we mentioned the use of abstract slicing to implement extended copy profiling. Future work may consider ways to realize this idea to give developers more information of the reported copy chains.

It is also interesting to extend the notions of cost and benefit (defined in terms of computations in this chapter) in many other ways to help performance evaluation and problem diagnosis. One example is to measure the effectiveness of data structures used as *caches*. The way of redefining costs and benefits for caches was discussed in Section 4.3. As another example, one can adapt the proposed cost and benefit for data to measure performance of control-flow entities, such as methods, components, and plugins. Faced with a large and complex application, a developer would need to first identify such coarser-grained program constructs that can potentially cause performance issues, in order to track down a performance problem through subsequent more detailed profiling.

CHAPTER 5: Making Sense of Container Usage: Memory Leak Detection Using Container Profiling

A major category of run-time memory bloat is the leak of memory during the execution. While garbage-collected languages can reduce memory-related bugs such as dangling pointers, programs written in these languages can still suffer from memory leaks caused by keeping references to useless objects. Leaks degrade run-time performance and significant leaks even cause the program to run out of memory and crash. In addition, memory leak bugs are notoriously difficult to find. Static analyses can be used to attempt the detection of such leaks. However, this detection is limited by the lack of scalable and precise reference/heap modeling (a well-known deficiency of static analyses), reflection, multiple threads, scalability for large programs, etc. Thus, in practice, identification of memory leaks is more often attempted with dynamic analyses. Existing dynamic approaches for heap diagnosis have serious limitations. Commercial tools such as JProfiler [51], JProbe [109] and LeakHunter [80] were developed to help understand types, instances and memory usage. However, this information is insufficient for programmers to locate a bug. As an example already mentioned in Chapter 1, in most cases, the fact that type `java.util.HashMap$Entry` has the highest number of instances tells the programmer nothing about the hash maps that hold these entries. Research tools for memory leak detection typically focus on heap differencing [43, 44, 71] and fine-grained object tracking [20, 61, 62, 107].

Of existing dynamic techniques, LeakBot [95], Cork [71], and Sleigh [20] represent the state of the art. There are two major research challenges in Java memory leak detection. Imprecision can result if these problems are not appropriately handled.

Challenge 1: definition of memory leak symptom All the dynamic approaches start with observing memory leak symptoms during the execution. What is a good indicator of a memory leak? Both LeakBot and Cork use heap growth as a heuristic, treating the increase of instances of certain types across garbage collection runs as a memory leak symptom. This could result in false positives, because growing instances are not necessarily true leaks and they may be collected later during the execution. Sleigh, on the other hand, uses staleness (time since last use) to find leaks. This approach could lead to imprecision as well. As an example, a frame in a Java Swing program cannot be treated as a leak, although it may never be used after it is created. In addition, larger objects that are less stale may have greater contribution towards the leak. For example, more attention should be paid to a big container that is not used for a while than to a never-used string.

Challenge 2: from-symptom-to-cause diagnosis All existing tools follow a traditional *from-symptom-to-cause* approach that starts from tracking all objects and finds those that could potentially be useless (symptom). It then tries to find the leaking data structure (cause) by analyzing direct and transitive references to these useless objects. However, the complex run-time reference relationships among objects in modern Java software significantly increases the difficulty of locating the source of the leak, which could lead to imprecise leak reports. It becomes even harder to find the cause of a leak if there are multiple data structures that are contributing to the problem. For example, as reported in [71], it took the authors a significant amount of time to find the sources of leaks after they read the reports generated by Cork.

Our proposal The inefficient use of containers is an important source of systemic bloat. Programming languages such as Java include a collection framework which provides abstract data types for representing groups of related data objects (e.g., lists, sets, and maps). Based on this collection framework, one can easily construct application-specific container types such as trees and graphs. Real-world programs make extensive use of containers, both through collection classes and through user-defined container types. Programmers allocate containers in thousands of code locations, using them in a variety of ways including storing data, implementing unsupported language features such as returning multiple values, and wrapping data in APIs to provide general service for multiple clients.

Arguably, misuse of (user-defined or Java built-in) containers is a major source of memory leak bugs in real-world Java applications. For example, most of the memory leak bugs reported in the Sun bug repository [141] were caused (directly or indirectly) by inappropriate use of containers. We propose a novel technique for Java that detects memory leaks using container profiling. The key idea behind the proposed technique is to *track operations on containers rather than on generic objects, and to report containers that are most likely to leak.* The major difference between our technique and the from-symptom-to-cause diagnosis approach is that we start by suspecting that all containers are leaking, and then use the "symptoms" to rule out most of them. Hence, we avoid the process of symptom-to-cause searching that can lead to imprecision and reduced programmer productivity.

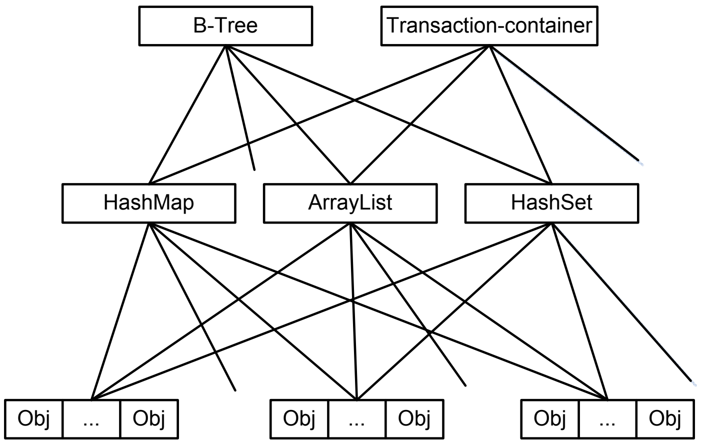


Figure 5.1: Container hierarchy in Java.

Figure 5.1 shows the container hierarchy typically used in a Java program: user-defined containers in the top layer use containers provided by the Java collection framework (illustrated in the second layer), which eventually store data in arrays (the bottom layer). The focus of our technique are containers in the first and second layers, because in most cases these containers are directly manipulated by programmers and hence are usually sources of leaks. Our technique does not track arrays, since in real-world Java programs they rarely cause leaks directly. Approaches such as [123] can be used to complement our technique in order to detect leaks directly caused by arrays.

Our technique requires ahead-of-time lightweight modeling of container behavior: users of the tool need to build a simple "glue layer" that maps methods of each container type to primitive operations (e.g., ADD, GET, and REMOVE). An automated tool instruments the application code and uses the user-supplied annotations to connect invocations of container methods with our run-time profiling libraries. In order to write this glue code, users have to be familiar with the container types used in the program. This does *not* increase the burden on the programmers: when using existing leak detection tools [20, 71, 95], they have to inspect the code to gain similar knowledge about containers so that they can interpret the tool-generated reports. Using our approach simply requires learning such knowledge in advance. Of course, the tool embeds pre-defined models for containers from the Java collection framework, and therefore programmers need to model only user-defined containers. As shown in our studies, running the tool even without modeling user-defined containers can still provide useful insights for finding leaks: in our reports, top-level Java library containers (the second layer in Figure 5.1) can direct one's attention to their direct or transitive owners, which are likely to be user-defined containers (the first layer in Figure 5.1) that are the actual causes of bugs.

Unlike previous approaches, our technique computes a heuristic *leaking confidence* value for each container based on a combination of its memory consumption and the staleness of its data elements, which could yield more accurate results compared to existing approaches [20, 71, 95]. For each container, the technique also ranks *call sites* in the source code, based on the average staleness of the elements retrieved at these sites. This container ranking and the related call site ranking provides information that can assist a programmer to quickly identify the source of the memory leak. The conceptual model used to compute these values and our implementation of the technique for Java are presented in Section 5.1 and Section 5.2, respectively. Our tool achieved high precision in reporting causes for two memory leak bugs from the Sun bug database [141] and a known memory leak bug in SPECjbb [138]—in fact, the top containers in the reports included the ones that leaked memory. In addition, an evaluation of the run-time performance of our technique showed that it has acceptable overhead for practical use.

Contributions The main contributions of this work are:

- A dynamic analysis that computes a confidence value for each container, which provides the basis for ranking and reporting of likely-leaking containers.
- A memory leak detection technique for Java based on the confidence analysis.
- A tool that implements the proposed technique.
- An experimental study of leak identification and run-time performance. The results indicate that our technique can precisely detect memory leak bugs with practical run-time overhead.

5.1 Leak Confidence Analysis

This section presents a confidence analysis that computes leaking confidence values for tracked containers. The goal of the analysis is to quantify the contribution of a container to memory leaks. Before describing the details of the analysis, we first provide some basic definitions.

Definition 5.1.1 (Container). *A* container type Γ *is an abstract data type with a set* Σ *of element objects, and two basic operations ADD and GET that manipulate* Σ*. A* container object γ^n *is an instantiation of* Γ *with* n *elements in its element set* Σ_γ*. An element can be of any subtype of* O*, which denotes the root of the type tree. ADD is a mappings of the form* $(\Gamma, O) \rightarrow \Gamma$ *that map a pair of container object and element object to a container object. GET is a mapping* $\Gamma \rightarrow O$ *from a container object to one of its elements. The effects of the operations are as follows:*

- *ADD*(γ^n_{pre}, *o*) : γ^m_{post} $\equiv$ $o \notin \Sigma_{\gamma_{pre}} \wedge o \in \Sigma_{\gamma_{post}} \wedge m{=}n{+}1 \wedge \forall p : p \in \Sigma_{\gamma_{pre}} \wedge p \neq o : p \in \Sigma_{\gamma_{post}}$
- *GET*(γ^n) : *o* $\equiv$ $o \in \Sigma_\gamma$
- *REMOVE*(γ^n_{pre}, *o*) : γ^m_{post} $\equiv$ $o \in \Sigma_{\gamma_{pre}} \wedge o \notin \Sigma_{\gamma_{post}} \wedge m{=}n\text{-}1 \wedge \forall p : p \in \Sigma_{\gamma_{pre}} \wedge p \neq o : p \in \Sigma_{\gamma_{post}}$

We treat all (Java library and user-defined) containers as implementations of the container ADT. Here and later in this dissertation, we use the term "container" to denote a container object. Tracking operations on a container requires user-supplied annotations to bridge the gap between methods defined in the Java implementations and the three basic ADT operations. We have already defined such annotations for the container types from the standard Java libraries.

During the execution of a program, let the program's memory consumption at a timestamp τ_i be m_i. In cases when τ_i is a moment immediately after garbage collection (we will refer to such moments as *gc-events*), it will be denoted by τ_i^{gc} and its memory consumption will be denoted by m_i^{gc}. A program written in a garbage-collected language has a *memory leak symptom* within a time region [τ_s, τ_e] if (1) for every gc-event τ_i^{gc} in the region, $m_s \leq m_i^{gc} \leq m_e$, and (2) in this region, there exists a subsequence $ss = (\tau_1^{gc}, \tau_2^{gc}, \ldots, \tau_n^{gc})$ of gc-events such that $\tau_i^{gc} < \tau_{i+1}^{gc}$ and $m_i^{gc} < m_{i+1}^{gc}$ for $i = 1, \ldots, n-1$. The period [τ_s, τ_e] will be referred to as a *leaking region*.

This definition helps to identify the appropriate time region to analyze, because most programs do not leak from the beginning. Moment τ_e can be specified by tool users as an analysis parameter, and can be different for different kinds of analyses. For post-mortem off-line diagnosis, τ_e is either the ending time of the program, or the time when an OutOfMemory error occurs. For on-line diagnosis done while the program is running, τ_e could be any time at which the user desires to stop data collection and to start analysis of this collected data. We use gc-events as "checkpoints" because at these times the program's memory heap consumption does not include objects that are unreachable.

The definition of a memory leak symptom does not require the amount of consumed memory at each gc-event to be larger than it was at the previous one, because in many cases some gc-events reclaim large amounts of memory, while in general the memory footprint still keeps increasing. The ratio between the number of elements n in the subsequence ss and the size of the entire sequence of gc-events within the leaking region can be defined by tool users as another analysis parameter, in order to control the length of the leaking region. There could be multiple definitions of starting moment τ_s corresponding to this user-defined ratio. Our approach chooses the smallest such value as τ_s, which defines the longest leaking region and allows more precise analysis. (Additional details are described in Section 5.2.)

A container σ is *memory-leak-free* if either (1) at time τ_e, it is in the state σ^0 (i.e., empty), or (2) it is garbage collected within the leaking region. That is, σ^n does not leak memory if at time τ_e, its accumulated number of ADD operations is equal to its accumulated number of REMOVE operations, assuming we treat the deallocation of σ^n as being equivalent to n REMOVE operations. Containers that are not memory-leak-free contribute to the memory leak symptom and are subject to further evaluations. However, this does not necessarily mean that all of them leak memory. For example, if OutOfMemory error occurs before some REMOVE operations of a container, this container is not memory-leak-free according to the above definition, although in reality it may very well be leak-free.

For each container that is not memory-leak-free by this definition, we compute a confidence value that indicates how much contribution it makes to the memory leak symptom. As mentioned earlier, our technique considers both the memory consumption and the staleness when computing the confidence for a container.

5.1.1 Memory Contribution

One factor that characterizes a container's contribution to the leak is the amount of memory the container consumes during its lifetime. We quantify this factor by defining a *memory time graph* which captures a container's memory footprint.

The relative memory consumption of a container σ at time τ is the ratio between the sum of the memory consumption of all objects that are reachable from σ in its object graph, and the total amount of memory consumed by the program at τ. The memory time graph for σ is a curve where the X-axis represents the relative time of program execution (i.e., τ_i/τ_e for timestamp τ_i) and the Y-axis represents the relative memory consumption of σ (i.e., $mem(\sigma)_i/total_i$ corresponding to X-point τ_i/τ_e). The starting point of the X-axis is τ_0/τ_e where τ_0 is $max(\tau_s$,allocation time of $\sigma)$, and the ending point of the X-axis is τ_1/τ_e where τ_1 is $min(\tau_e$,deallocation time of $\sigma)$.

A sample graph is illustrated in Figure 5.2. The X-axis starts at 0.4 relative time (i.e., $0.4 \times \tau_e$ absolute time), which represents either the starting time of the leak region τ_s, or σ's allocation time, whichever occurs second. The graph indicates that σ does not get freed within the leak region, because the X-axis ends at 1, which represents the ending time τ_e of the leak region.

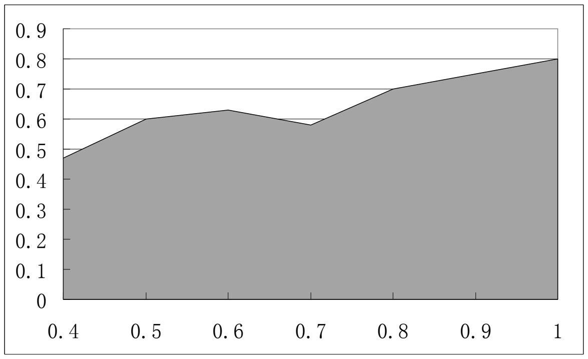


Figure 5.2: A sample memory time graph.

Using the memory time graph, a container's *memory contribution* (MC) is defined to be the area covered by the memory consumption curve in the graph. In the example in Figure 5.2, this area is shown in dark. Because the memory time graph starts from τ_s (or later), the MC considers only a container's memory consumption within the leaking region. For a particular container, both its memory consumption and its lifetime contribute to its MC. Since MC should reflect the influence of both the container itself and all objects (directly or transitively) referenced by it, the memory consumption of the container is defined as the amount of memory consumed by its entire object graph.

Because relative values (i.e., between 0 and 1) are used to measure the memory consumption and the execution time, the MC of a container is also a value between 0 and 1. Containers that have larger MC contribute more to the memory leak symptom. Note that in practice it is likely to be too expensive to compute the exact MC value for a container, because the container's memory consumption changes frequently as the program executes. Section 5.2 presents a sampling approach that can be used to approximate this value.

5.1.2 Staleness Contribution

The second factor that characterizes a container's contribution to the leak is the staleness of the container's elements. The staleness of an object is defined in [20] as the time since the object's last use. Our work provides a new definition of staleness in terms of a container and its elements.

The *staleness* of an element object o in a container σ is $\tau_2 - \tau_1$ where REMOVE(σ, o) occurs at τ_2, GET(σ):o occurs at τ_1, and there does not exist another GET operation that returns o in the region $[\tau_1, \tau_2]$. If $\tau_1 < \tau_s$, τ_1 is redefined to be τ_s. If $\tau_2 < \tau_s$, the staleness is undefined. In other words, the staleness of o is the distance between the time when o is removed from σ and the most recent time when o is retrieved from σ. If o is never retrieved from σ, τ_1 should correspond to the ADD operation that adds o to σ. If o is never removed from σ, τ_2 is either the deallocation time of σ, or the ending time of the leaking region τ_e. The intuition behind the definition is that if the program no longer needs to retrieve an element from a container, the element becomes useless to that container. Hence, the staleness of the element measures the period of time when the element becomes useless but is still being kept by the container. In addition, tracking occurs only within the leaking region: if an element's removal time τ_2 is earlier than the starting time of the leaking region, we do not compute the staleness for the element. Note that the last GET operation of a container element may not correspond to the last use site of this object—a reference obtained from the GET operation may be stored somewhere else and used later from there. However, it is important to keep in mind that the staleness of the element, as defined above, is a measurement of a container's misbehavior—the object no longer needs to be obtained from the container, but the container still references it. Hence,

ID	*Type*	*LC*	*MC*	*SC*
11324773	util.HashMap	0.449	0.824	0.495
18429817	util.LinkedList	0.165	0.820	0.194
8984226	util.LinkedList	0.050	0.809	0.062
2263554	util.WeakHashMap	0.028	0.820	0.034
15378471	util.LinkedList	0.018	0.029	0.256
5192610	swing.JLayeredPane	0.011	0.824	0.013
30675736	swing.JPanel	0.011	0.824	0.013
19526581	swing.JRootPane	0.011	0.824	0.013
17933228	util.Hashtable	0.000023	0.0007	0.026
33263898	util.ArrayList	0.00000026	0.0000032	0.046

Table 5.1: Partial report of LC, MC, and SC values.

regardless of whether an object is stored elsewhere via other references, a container behaves properly as long as it removes the object once the container does not need to keep it.

The *staleness contribution* (SC) of a container σ is the ratio of $(\sum_{i=1}^{n} staleness(o_i) / n)$ and $(\tau_e - \tau_s)$, where the sum is over all elements $o_1, \ldots, o_n$ that have been added to σ and whose staleness is well-defined. Thus, SC is the average staleness of elements that have ever been added to σ, relative to the length of the leaking region. In addition, the removal time of these elements must be within the leaking region. Because the staleness of each individual element is $\leq$ the length of leaking region, SC is a value between 0 and 1. Containers that have larger SC values contribute more to the memory leak symptom.

5.1.3 Putting it All Together: Leaking Confidence

Based on the memory contribution and the staleness contribution, we define a container's *leaking confidence* (LC) to be computed as $SC \times MC^{1-SC}$. Clearly, LC is a value between 0 and 1; also, increasing either SC or MC while keeping the other factor unchanged increases LC. We define LC as an exponential function of SC to show that staleness is more important than memory consumption in determining a memory leak. This definition of LC has several desirable properties:

- MC=0 and SC$\in$[0, 1] $\Rightarrow$ LC=0. If the memory contribution of a container is small enough (i.e., close to 0), the confidence of this container is close to 0, no matter how stale its elements are. This property helps filter out containers that hold small objects, such as strings.

- SC=0 and MC$\in$[0, 1] $\Rightarrow$ LC=0. If every element in a container gets removed immediately after it is no longer used (i.e., the time between the GET and REMOVE operations is close to 0), the confidence of this container is 0, no matter how large the container is.

- SC=1 and MC$\in$[0, 1] $\Rightarrow$ LC=1. If all elements of a container never get removed after they are added (i.e., every element crosses the entire leaking region), the confidence of the container is 1, no matter how large the container is.

- MC=1 and SC$\in$[0, 1] $\Rightarrow$ LC=SC. If the memory contribution of a container is extremely high (close to 1), the confidence of this container is decided by its staleness contribution.

Our study shows that this definition of confidence effectively separates containers that are the sources of leaks from those that do not leak. A sample report that includes LC, MC, and SC for several containers is illustrated in Table 5.1. This table is a part of the report generated by our tool when analyzing Sun's bug #6209673.

The first container in the table is the one that actually leaks memory. Note that the LC value of this container is much larger than the LC values for the remaining containers. Based on this report, it is straightforward for a programmer to find and fix this bug.

5.2 Memory Leak Detection for Java

Based on the leak confidence analysis, this section presents our memory leak detection technique for Java.

5.2.1 Container Modeling

For each container type, there is a corresponding "glue" class. For each method in the container type that is related to ADD, GET, and REMOVE operations, there is a static method in the glue class whose name is the name of the container method plus the suffix "_before" or "_after". The suffix indicates whether calls to the glue method should be inserted before or after call sites invoking the original method. The parameter list of the glue method includes a call site ID, the receiver object, and the formal parameters of the container method. For the suffix "_after", the return value of the container method is also added. Figure 5.3 shows the modeling of container class `java.util.HashMap`. It is important to note that most of this glue code can be generated automatically using predefined code templates.

The glue methods call our profiling library to pass the following data: the call site ID (`csID`), the container object, the element object, the number of elements in the container before the operation is performed, and the operation type. The call site ID is generated by our tool during instrumentation.

```
class HashMap{
  Object put(Object key, Object value){...}
  Object get(Object key){...}
  Object remove(Object key){...}
  ...
}
```

(a) Container class HashMap

```
class Java_util_HashMap{
  static void put_after(int csID, Map receiver, Object key,
                        Object value, Object result) {
     /* if key does not exist in the map */
     if(result == null){
       /* use user-defined hash code as ID */
       Recorder.v().useUserDefHashCode();
       /* record operation ADD(receiver, key) */
       Recorder.v().record(csID, receiver, key,
                    receiver.size()-1, Recorder.EFFECT_ADD);
     }
  }
  static void get_after(int csID, Map receiver, Object key,
                        Object result){
     /* if an entry is found */
     if(result != null){
       Recorder.v().useUserDefHashCode();
       /* record operation GET(receiver):key */
       Recorder.v().record(csID, receiver, key, receiver.size(),
                           Recorder.EFFECT_GET);
     }
  }
  static void remove_after(int csID, Map receiver, Object key,
                           Object result){
     if(result != null){
       Recorder.v().useUserDefHashCode();
       /* record operation REMOVE(receiver, key) */
       Recorder.v().record(csID, receiver, key,
                    receiver.size()+1, Recorder.EFFECT_REMOVE);
     }
  }
}
```

(b) Glue class for HashMap

Figure 5.3: Modeling of container `java.util.HashMap`.

The container object, the element object, and the operation type are used to compute the SC for the container. Recording the number of elements in a container is needed because once the leaking region is decided, we want to analyze only the data collected within the region. Knowing the number of elements of a container at the starting time of the region can help to avoid scanning the data before the region.

	Container Method Call	*Interpretation*
(a)	A.add(o)	ADD(A, o)
	o=A.get(..)	o=GET(A)
	o=A.remove(..)	REMOVE(A, o)
	A.addAll(B)	$\forall\ o \in B$, o=GET(B) $\forall\ o \in B$, ADD(A, o)
	A.removeAll(B)	$\forall\ o \in B$, o=GET(B) $\forall\ o \in A \cap B$, REMOVE(A, o)
	A.retainAll(B)	$\forall\ o \in B$, o=GET(B) $\forall\ o \in A \setminus B$, REMOVE($A$, o)
	A.containsAll(B)	$\forall\ o \in B$, o=GET(B)
	A.toArray()	$\forall\ o \in A$, o=GET(A)
	A.iterator()	$\forall\ o \in A$, o=GET(A)
(b)	v = A.get(k)	k=GET(A) if $v \neq$ null
	r = A.put(k, v)	ADD(A, k) if $r \neq$ null k=GET(A) otherwise
	r = A.remove(k)	REMOVE(A, k) if $r \neq$ null
	A.putAll(B)	$\forall\ k \in B.keySet()$, k=GET(B) $\forall\ k \in B.keySet()$: if $k \in A.keySet()$, k=GET(A) otherwise, ADD(A, k)
	A.keySet()	$\forall\ k \in A.keySet()$, k=GET(A)
	A.values()	$\forall\ k \in A.keySet()$, k=GET(A)
	A.entrySet()	$\forall\ k \in A.keySet()$, k=GET(A)
	A.clear()	$\forall\ k \in A.keySet()$, REMOVE($A$, k)

Table 5.2: Mapping between actual container methods and abstract container operations: (a) methods defined in `java.util.Collection`; (b) methods defined in `java.util.Map`.

In order to reduce the run-time overhead, we use an integer ID to track each object (i.e., container and element). The first time a container performs its operation, we tag the container object with the ID (using JVMTI). The ID for a container object (e.g., `receiver` in Figure 5.3) is its identity hash code determined by its internal address in the JVM. For an element object, the identity hash code is used as element ID if the container does not have hash-based functions; otherwise, the element ID is the user-defined hash code. For example, in Figure 5.3, calls to `useUserDefHashCode` inform our library that the ID for `key` should be its user-defined hashcode. For `HashMap`, we only track `key` as a container element, because `key` is representative of a map entry. Methods that retrieve the entire set of elements, such as `toArray` and `iterator` are treated as a set of GET operations performed on all container elements. Note that it may not be precise to model `iterator` in this way, since a particular object is retrieved only when method `iterator.next()` is invoked. However, it is common in Java programs that all elements of a container are retrieved after an iterator over the container is obtained by invoking method `iterator()`. In addition, this treatment avoids the use of heavyweight static analysis that has to be employed to relate `Iterator` objects with their corresponding container objects. The mapping of a typical set of container methods defined in interfaces `java.util.Collection` and `java.util.Map` is illustrated in Table 5.2. Upper-case letters and lower-case letters are used to represent containers and elements, respectively.

5.2.2 Instrumentation

Our tool uses the Soot program analysis framework [132, 147] to perform code instrumentation. For each call site in an application class at which the receiver is a container, calls to the corresponding glue method are inserted before and/or after the site. For a container object, code is also inserted after its allocation site in order to track its allocation time.

Naively instrumenting a Java program can cause tracking of a large number of containers, which may introduce significant run-time overhead. Because thread-local and method-local containers[3] are not likely to be the source of a leak, we employ an

[3]Containers that are not reachable from multiple threads, and whose lifetime is limited within their allocating methods.

Name	*Description*
GC_T	GC timestamps
GC_M	Total live memory after GCs
CON_M	Memory taken up by containers
CON_T	Timestamps when measuring CON_M
CON_A	Allocation times of containers
CON_D	Deallocation times of containers
OPR	Operations (csID, container, element, #elements, type)
	Purpose
GC_T	To identify the leaking region
GC_M	To identify the leaking region
CON_M	To compute MC for containers
CON_T	To compute MC for containers
CON_A	To compute MC and SC for containers
CON_D	To compute MC and SC for containers
OPR	To compute SC for containers

Table 5.3: Data collected by our profiler.

escape analysis to identify a set S of thread-local and method-local objects. We do not instrument call sites if the points-to sets of their receiver variables are subsets of S.

5.2.3 Profiling

Table 5.3 lists the types of data that need to be obtained by our profiler. In order to identify the leaking region, we need to collect GC finishing times (GC_T), and live memory at these times (GC_M). This can be done by using JVMTI agents.

In order to compute MC for containers, we need to collect amounts of memory taken up by the entire object graphs of containers (CON_M), and the corresponding collection times (CON_T). We measure the memory usage of a container by traversing the object graph starting from the container (using reflection). As mentioned in

Section 5.1, it is impractical to compute the exact value of MC. Sampling is used during the execution, and the obtained values are used to approximate the memory time graph. Frequent sampling results in precise approximation, but increases run-time overhead.

We launch periodic object graph traversals (for a set of tracked containers) every time after a certain number of gc-events is seen. The number of gc-events between two traversals can be given as a parameter to our tool to control precision and overhead. Our experimental study indicates that choosing 50 as the number of gc-events between traversals can keep the overhead low while achieving high precision.

Because an object graph traversal can be expensive, this task is assigned to a newly-created thread executing in parallel with the main program. Note that such a solution is particularly well-suited for modern architectures with multi-core processors. Once a container operation is performed (i.e., `record` in Figure 5.3 is invoked), `record` adds the ID of the container to a global queue. When the given number of gc-events complete, our JVMTI agent activates this thread, which reads IDs from the queue, retrieves the corresponding objects, and performs graph traversals. The allocation time of a container (CON_A) can be collected by the instrumentation at the allocation site, and our JVMTI agent can provide the deallocation time of a tagged container (CON_D).

In order to compute SC for containers, we have to record every operation that a tracked container performs (OPR). Because OPR events can result in large amounts of data, we use a data compression strategy to reduce space overhead. The OPR data is stored in a tree structure. Data at a higher level of the tree is likely to be more frequently repeated. For example, type `java.util.HashMap`, which is at the highest

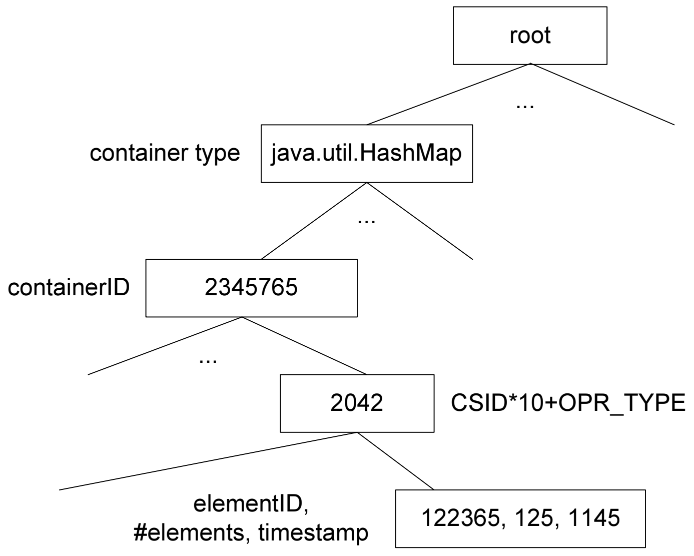


Figure 5.4: Compressed recording of OPR events.

level of the tree, appears in the event sequence for many container IDs. Similarly, for a single container ID, many call sites and operations need to be recorded. The tree representation is illustrated in Figure 5.4. The type of container is a parent of the container ID. A child of the container ID is a combination of the call site ID and the operation type (encoded as a single integer `csID*10+opr_type`). The leaf nodes contain tuples of element ID, number of elements in the container before this operation, and a timestamp.

Keeping too much profiling data in memory degrades program performance. We periodically record the data to disk to reduce its influence on the run-time execution. The frequency of recording is the same as that of object graph traversal: our JVMTI agent creates a recording thread that is activated at the same time as the graph traversal thread is activated. All the threads synchronize when recording to disk is about to start. As discussed earlier, selecting an appropriate recording (and sampling) rate is key to reducing the run-time overhead.

5.2.4 Data Analysis

Our current implementation performs an offline analysis of the collected data after the program finishes or runs out of memory. Thus, the end of the leaking region τ_e is the ending time of the program. The implementation can easily be adapted to run the analysis online (in another process) and generate the report while the original program is still running.

The first step of the analysis is to scan GC_T and GC_M information to determine the leaking region. The current implementation employs 0.5 as the ratio used to define this region, which means that at least half of the gc-events form a subsequence with increasing memory consumption (recall the leak region definition from Section 5.1). After the smallest τ_s that satisfies this constraint is found, each container's OPR data is uncompressed into individual operations and they are sorted by timestamp. The container ID and its operation list are stored in map *oper_map*. For each container, the analysis also determines the first operation that is performed after τ_s; the container ID and the number of container elements at this first operation are stored in map *size_map*. Operations that occurred before τ_s are discarded.

For each container, CON_M and CON_T data is used to approximate the memory time graph and the MC value. The approximation assumes that the memory used by the container does not change between two samples. Thus, MC is $\sum_{i=0}^{n-1}(\text{CON}_{T,i+1} - \text{CON}_{T,i}) \times \text{CON}_{M,i}$ where i represents the i-th sample.

Algorithm 1: Computing SC for containers.

```
FIND_SC(Double $\tau_e$, Double $\tau_s$, Map size_map, Map oper_map)
/* operation list for each container */
List oper_list
/* The result map contains each container ID and its SC */
Map result = $\emptyset$
for each container ID c in oper_map do
  Map temp = $\emptyset$ /* a temporary helper map */
  oper_list = oper_map.get(c)
  Integer total = 0 /* total number of elements */
  Double sum = 0 /* $\sum$ staleness */
  /* Number of elements in c at time $\tau_s$ */
  Integer ne = size_map.get(c)
  for each operation opr in oper_list do
    if opr.type == "ADD" then
      temp.add(opr.elementID, opr.timestamp)
    end if
    if opr.type == "GET" then
      update temp with (opr.elementID, opr.timestamp)
    end if
    if opr.type == "REMOVE" then
      if temp.contains(opr.elementID) then
        Integer lastget = temp.get(opr.elementID)
        sum += opr.timestamp − lastget
        total += 1
        temp.remove(opr.elementID)
      else
        /* The element is added before $\tau_s$ */
        sum += opr.timestamp − $\tau_s$
        total += 1
        ne −= 1
      end if
    end if
  end for
  if temp.size > 0 then
    /* These elements are never removed */
    for each elementID in temp do
      Integer lastget = temp.get(elementID)
      sum += $\tau_e$ − lastget
      total +=1
    end for
  end if
  if ne > 0 then
    /* Elements are added before $\tau_s$ and never removed */
    sum += $(\tau_e - \tau_s) \times$ ne;
    total += ne
  end if
  c.SC = (sum/total)/$(\tau_e - \tau_s)$
  result.add(c, c.SC)
end for
return result
```

Algorithm 1 shows the computation of SC for containers. The algorithm scans a container's operation list, and for each element ID, finds its last GET operation, its REMOVE operation, and the distance between them. (Recall that the deallocation of the container is treated as a set of REMOVE operations on all elements.) For an element that is added before τ_s (lines 27–30), staleness is the distance between the REMOVE operation and τ_s. For an element that is never removed (lines 34–39), staleness is the distance between τ_e and the last GET operation. For elements that are added before τ_s and never removed (lines 42–45), staleness is $\tau_e - \tau_s$.

Leaking call sites For each element in a container, the analysis finds the call site ID corresponding to its last GET operation. Then, it computes the average staleness of elements whose last GET operations correspond to that same call site ID. These call site IDs are then sorted in decreasing order of this average value. Thus, the tool reports not only the potentially leaking containers (sorted by the LC value), but also, for each container, the potentially leaking call sites (with their source code location) sorted in descending order by their average staleness. Our experience indicates that this information can be very helpful to a programmer trying to identify the source of the memory leak bug.

5.3 Empirical Evaluation

To evaluate the proposed technique for container-based memory leak detection for Java, we performed a variety of experimental studies focusing on leak identification and execution overhead. Section 5.3.1 illustrates the ability of our technique to help a programmer find and fix real-world bugs. Section 5.3.2 presents a study of the incurred overhead.

5.3.1 Detection of Real-World Memory Leaks

The experiments were performed on a 2.4GHz dual-CPU PC with 2GB RAM, running Windows XP. Three different sampling/recording rates were used: 1/15gc, 1/50gc, and 1/85gc (i.e., once every 15, 50, or 85 gc-events). The experimental subjects were two memory leak bugs reported in the Sun bug database [141], a known leak in SPECjbb [138], as well as a bug contained in a leak example from an IBM developerWorks column [57].

Java AWT/Swing Bugs

About half of the memory leak bugs in the JDK come from AWT and Swing. This is the reason we chose two AWT/Swing related leak bugs #6209673 and #6559589 for evaluation. The first bug has already been fixed in Java 6, while the second one is still open and unresolved.

Bug report **#6209673** describes a bug that manifests itself when switching between a running Swing application that shows a JFrame and another process that uses a different display mode (e.g., a screen saver)—the Swing application eventually runs out of memory. According to a developer's experience [104], the bug was very difficult to track down before it was fixed. We instrumented the entire `awt` and `swing` packages, and the test case provided in the bug report. We then ran the instrumented program and reproduced the bug. Figure 5.5 shows the tool reports with three sampling rates. Each report contains the top three containers, for each container the top three potentially leaking call sites (`---cs`), and the time used to analyze the data.

Sampling rates 1/15gc and 1/50gc produce the same containers, in the same order. The first container in the reports is a `HashMap` in class `javax.swing.RepaintManager`. We inspected the code of `RepaintManager` and found that the container was an instance field called `volatileMap`. The call site in the report (with average staleness 0.507) directed us to line 591 in the source code of the class, which corresponds to a GET operation

```
Container:11324773 type: java.util.HashMap
         (LC: 0.449, SC: 0.495, MC: 0.825)
---cs: javax.swing.RepaintManager:591 (Average staleness: 0.507)

Container:18429817 type: java.util.LinkedList
         (LC: 0.165, SC: 0.194, MC: 0.820)
---cs: java.awt.DefaultKeyboardFocusManager:738 (0.246)

Container:8984226 type: java.util.LinkedList
         (LC: 0.051, SC: 0.062, MC: 0.809)
---cs: java.awt.DefaultKeyboardFocusManager:851 (0.063)
---cs: java.awt.DefaultKeyboardFocusManager:740 (0.025)
Data analyzed in 149203ms
```
(a) 1/15gc sampling rate

```
Container:29781703 type: java.util.HashMap
        (LC: 0.443, SC: 0.480, MC: 0.855)
---cs: javax.swing.RepaintManager:591 (Average staleness: 0.480)

Container:2263554 type: class java.util.LinkedList
        (LC: 0.145, SC:0.172, MC: 0.814)
---cs: java.awt.DefaultKeyboardFocusManager:738 (0.017)

Container:399262 type: class javax.swing.JPanel
        (LC: 0.038, SC:0.044, MC: 0.860)
---cs: javax.swing.JComponent:796 (0.044)
Data analyzed in 21593ms
```
(b) 1/50gc sampling rate

```
Container:15255515 type: java.util.HashMap
        (LC: 0.384, SC:0.426, MC: 0.835)
---cs: javax.swing.RepaintManager:591 (0.426)

Container:19275647 type: java.util.LinkedList
        (LC: 0.064, SC:0.199, MC: 0.244)
---cs: java.awt.SequencedEvent:176 (0.204)
---cs: java.awt.SequencedEvent:179 (0.010)
---cs: java.awt.SequencedEvent:128 (1.660E-4)

Container:28774302 type: javax.swing.JPanel
        (LC: 0.036, SC:0.042, MC: 0.839)
---cs: javax.swing.JComponent:796 (0.042)
Data analyzed in 10547ms
```
(c) 1/85gc sampling rate

Figure 5.5: Reports for JDK bug #6209673.

```
image = (VolatileImage)volatileMap.get(config)
```

The tool report indicates that the image obtained at this call site may not be properly removed from the container. For a programmer that is familiar with the code, this information may be enough to identify the bug quickly. Since the code was new for us, we had to learn more about this class and the overall display-handling strategy of Swing to understand the bug. Because the bug was already resolved, we examined the bug evaluation, which confirmed that `volatileMap` is the root of the leak. The cause of the bug is caching by `RepaintManager` of all VolatileImage objects, regardless of whether or not they are currently valid. Upon a display mode switch, the old GraphicsConfiguration objects under the previous display mode get invalidated and will not be used again. However, the VolatileImage for an obsolete GraphicsConfiguration is never removed from `volatileMap`, and hence all resources allocated by the image continue taking up memory until an OutOfMemory error occurs.

Note that the report with sampling rate 1/85gc "loses" the LinkedList in `DefaultKeyboardFocusManager`, which appears as the second container in the other two reports. Although this container is not the source of the bug, it demonstrates that sampling at 1/85gc may not be frequent enough to maintain high precision. Note that analysis time decreases with the decrease in sampling rate, because the tool loads and processes less data during the analysis.

Compared to our reports, existing approaches that keep track of generic objects (i.e., do not have our container-centric view) would report allocation sites of some types of objects that either (1) continuously grow in numbers or (2) are not used for a while. For bug #6209673, for example, there are growing numbers of objects of numerous types that are reachable by VolatileImage and GraphicsConfiguration objects. Tools such as Cork [71] have to backward-traverse the object graph from the growing objects to find the type of objects that do not grow in numbers. However, the useless objects are inter-referenced, and moreover, traversing back from these growing objects can potentially find multiple types whose instances remain unchanged. In this case, the container that holds GraphicsConfigurations, the JFrame window, the GraphicsDevice object, the map that holds VolatileImages, etc. can all be data structures that are backward-reachable from the growing objects and whose numbers of instances do not grow. Tools such as Sleigh [20] report errors based solely on the staleness of objects. In this case, the JFrame object would be the most stale object because it is never used after it is created. In addition, there are numerous types of objects that are more stale than VolatileImages, such as all the components in the frame. Hence, Sleigh could report all these objects as the sources of the leak, including many false positives. Finally, both of these existing approaches require non-standard JVM modifications and support, while our technique uses only code instrumentation and the standard JVMTI interface.

Currently, the report generated by our tool does not contain calling context information, which has been considered to be useful in locating the cause of a bug. This, in fact, does not undermine the practical effectiveness of our tool. Unlike tools that track arbitrary objects and therefore need this information to locate the bug-inducing operations, our tool pinpoints the cause containers and the last GET operations, which are strong indications of the location of the bug. For instance, many call chains reported by Sleigh from [20] go from the last use sites of stale objects backward to call sites that invoke container methods, which are important to investigate. These call chains could not be as useful if the call sites corresponding container operations are directly reported (as in our tool).

Report **#6559589** describes a bug in Java 6 build 1.6.0_01: calling method `JScrollPane.updateUI()` in a Swing program that uses `JScrollPane` causes the

```
Container:5678233 type: java.util.Vector
       (LC: 0.890, SC: 0.938, MC: 0.427)
---cs: java.awt.Window:1825 (0.938)

Container:3841106 type: java.beans.PropertyChangeSupport
       (LC: 0.645, SC:0.779, MC: 0.427)
---cs: java.awt.Component:7007 (0.779)

Container:24333128 type: javax.swing.UIDefaults
       (LC: 0.644, SC:0.875, MC: 0.087)
---cs: javax.swing.UIDefaults:334 (0.868)
---cs: javax.swing.UIDefaults:308 (0.660)
Data analyzed in 454ms
     (a) 1/15gc sampling rate
```

```
Container:5678233 type: java.util.Vector
       (LC: 0.890, SC:0.938, MC: 0.427)
---cs: java.awt.Window:1825 (0.938)

Container:30318493 type: java.beans.PropertyChangeSupport
       (LC: 0.668, SC:0.828, MC: 0.288)
---cs: java.awt.Component:7007 (0.828)

Container:9814147 type: javax.swing.UIDefaults
       (LC: 0.101, SC: 0.327, MC: 0.175)
---cs: javax.swing.UIDefaults:334 (0.984)
---cs: javax.swing.UIDefaults:308 (0.903)
Data analyzed in 282ms
     (b) 1/50gc sampling rate
```

```
Container:5678233 type: java.util.Vector
       (LC: 0.293, SC:0.425, MC: 0.525)
---cs: java.awt.Window:1825 (0.425)

Container:30502607 type: javax.swing.JLayeredPane
       (LC: 0.117, SC:0.221, MC: 0.441)
---cs: javax.swing.JComponent:796 (0.162)

Container:2665317 type: javax.swing.UIDefaults
       (LC: 0.096, SC:0.363, MC: 0.124)
---cs: javax.swing.UIDefaults:334 (0.359)
---cs: javax.swing.UIDefaults:308 (0.340)
Data analyzed in 297ms
     (c) 1/85gc sampling rate
```

Figure 5.6: Reports for JDK bug #6559589.

number of listeners to grow. Because it is common knowledge that PropertyChange-Listeners are managed by `java.bean.PropertyChangeSupport`, we modeled this class as a container and wrote a glue class for it. The generated reports are shown in Figure 5.6. The first container in all three reports is a vector in `java.awt.Window`,

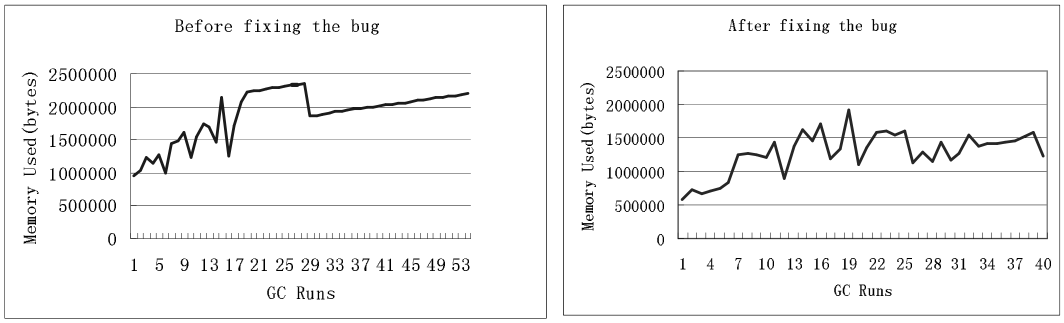


Figure 5.7: Memory footprint before and after fixing JDK bug #6559589.

which corresponds to an instance field `ownedWindowList`. Line 1825 of `Window` contains an ADD operation

```
ownedWindowList.addElement(weakWindow)
```

Field `ownedWindowList` is used to hold all children windows of the current window. The reporting of this call site by the tool indicates that when a `Window` object is added to the vector, it may not be properly removed later. We quickly concluded that this cannot be the source of the bug, because windows in a Swing program usually hold references to each other until the program finishes. This forced us to look at the second container in reports (a) and (b), which is a `PropertyChangeSupport` object in `java.awt.Component`. The reported call site at line 7007 of `Component` is

```
changeSupport.addPropertyChangeListener(listener)
```

The container is an instance field `changeSupport`, which stores all PropertyChangeListeners registered in this component. The call site indicates that the bug may be caused by some problem in `JScrollPane` that does not appropriately remove listeners. Registering and unregistering of listeners for `JScrollPane` is done in a set of `ScrollPaneUI` classes. The test case uses a metal look and feel, which is represented by class `MetalScrollPaneUI`, a subclass of `BasicScrollPaneUI`. We checked method `uninstallListeners` in `MetalScrollPaneUI`, which is supposed to release listeners from the component, and found that this method calls the method with the same name in its super class, but does not remove the `scrollBarSwapListener` object held by a private field in the subclass. Further investigation revealed an even more serious problem: method `uninstallListeners` in the subclass was not executed at all, because its signature was different from the signature of the method with the same name in superclass `BasicScrollPaneUI`:

```
  /* BasicScrollPaneUI */
  void uninstallListeners(JComponent c)
  /* MetalScrollPaneUI */
  void uninstallListeners(JScrollPane scrollPane)
```

Hence, the causes of the bug are (1) `uninstallListeners` in `MetalScrollPaneUI` fails to override the appropriate method in superclass `BasicScrollPaneUI`, and (2) the listener defined in subclass `MetalScrollPaneUI` is not removed by its own `uninstallListeners`. We modified the code accordingly, and the memory leak disappeared. The memory footprint before and after fixing the bug is illustrated in Figure 5.7. We have submitted our modification as a comment in the bug database. Again, the report that used 1/85gc sampling rate failed to include the `PropertyChange Support` object, which is the source of the leak.

SPECjbb Bug

Benchmark SPECjbb2000 simulates an order processing system and is intended for evaluating server-side Java performance [138]. The program contains a known memory leak bug that manifests itself when running for a long time without changing

```
Container:4451472 type: java.util.Hashtable
        (LC: 0.135, SC: 0.190, MC: 0.659)
---cs: spec.jbb.StockLevelTransaction:225 (0.214)
---cs: spec.jbb.StockLevelTransaction:211 (0.190)

Container:7776424 type: java.util.Hashtable
        (LC: 0.110, SC:0.157, MC: 0.659)
---cs: spec.jbb.StockLevelTransaction:211 (0.157)
---cs: spec.jbb.StockLevelTransaction:225 (0.114)

Container:28739781 type: java.util.Hashtable
        (LC: 0.102, SC:0.146, MC: 0.654)
---cs: spec.jbb.StockLevelTransaction:211 (0.146)
---cs: spec.jbb.StockLevelTransaction:225 (0.122)
Data analyzed in 4078ms
 (a) before modeling of longBTree, using 1/50gc


Container:27419736 type: spec.jbb.infra.Collections.longBTree
        (LC: 0.687, SC: 0.758, MC: 0.666)
---cs: spec.jbb.District:264 (0.826)
---cs: spec.jbb.StockLevelTransaction:225 (0.624)
---cs: spec.jbb.StockLevelTransaction:211 (0.519)

Container:21689791 type: spec.jbb.infra.Collections.longBTree
        (LC: 0.685, SC: 0.757, MC: 0.662)
---cs: spec.jbb.District:264 (0.783)
---cs: spec.jbb.StockLevelTransaction:211 (0.370)
---cs: spec.jbb.District:406 (2.944E-4)

Container:27521273 type: spec.jbb.infra.Collections.longBTree
        (LC: 0.667, SC: 0.727, MC: 0.727)
---cs: spec.jbb.Warehouse:456 (0.798)
---cs: spec.jbb.District:264 (0.784)
---cs: spec.jbb.StockLevelTransaction:211 (0.484)
 (b) after modeling of longBTree, using 1/50gc
```

Figure 5.8: Report for the SPECjbb2000 bug.

warehouses. The report generated by our tool for rate 1/50gc is shown in Figure 5.8. Due to the imprecision of using sampling rate 1/85gc, the report for it is not shown. We also do not show the report of using sampling rate 1/15gc, because the containers and their order in this report are the same as in the report for 1/50gc.

The program was first instrumented without modeling any user-defined containers. The result is shown in Figure 5.8(a). It is straightforward to see that none of the containers in the list are likely to leak memory, because their confidences are very small. The first container in the report refers to a hashtable that holds stocks of an order line. We did not find any problem with the use of this container. However, we observed that the order lines are actually obtained from an order table, which has a type of `longBTree`. We found that `longBTree` is a container class that implements a BTree data structure and is used to hold orders. It took us several minutes to write a glue class for `longBTree`. We then re-instrumented and re-ran the program. The resulting tool report is shown in Figure.5.8(b). The top three containers in the report are now instances of `longBTree`.

Line 264 of `spec.jbb.District` is an ADD operation

```
    orderTable.put(anOrder.getId(), anOrder)
```

which indicates that `orderTable` may leak memory. Methods `removeOldestOrder`, `removeOldOrders`, and `destroy` contain REMOVE operations for `orderTable`. We focused on the first two methods, because `destroy` could not be called when a district is still useful. Using a standard IDE, we found the callers of these methods: `removeOldestOrder` is called only once within `DeliveryTransaction`, and `removeOldOrders` is never called. Therefore, when a transaction completes, it removes only the oldest order from the table, which causes the heap growth. Inserting code to remove orders from the table fixed the bug. We used less time (a few hours) than the authors of [71] did (a day) to locate the bug in this program, which we had never studied before.

Memory Leak from Java DeveloperWorks

An IBM developerWorks column [57] contains a sample leak bug that uses behaviors of three containers to illustrate different levels of leaking severity. Specifically, the first container never removes elements. The second container removes elements after they become useless, but its usage has an error that leads to several Integer objects not being removed in each iteration. The third container removes all elements before the end of each iteration. The program executes in iterations and exercises operations of all the three containers in each iteration. We modify the program by adding small objects (arrays of 1000 integers) to the first container (that causes the quick leak), adding large objects (arrays of 10000 integers) to the second container (that causes the slow leak), and adding even larger objects (arrays of 50000 integers) to the third container. Although this is not a real-world leak bug, the presence of leaks of multiple levels of severity is useful for us to evaluate the sensitivity of the proposed leak confidence model in separating containers that have different contributions to the leak. We instrumented the program and ran it until OutofMemoryError was caught. The generated report for this program is shown in Figure 5.9.

Note that the leak confidence computed for the first container is much larger than those computed for the remaining two containers. The primary reason is that this container does not remove any elements during the execution. In other words, its large staleness contribution dominates the fact that the other two containers contain large objects. It is interesting to see that the third (leak-free) container is ranked higher than the second (slowly-leaking) one: they have similar staleness contributions, but the third one consumes much more memory. According to this report, the third container is actually the one to which we need to pay more attention, since it has more significant influence on run-time performance than the second one. Removing elements earlier, after they become useless (instead of waiting until the end of an iteration) can fix the problem. From this experiment, it can be seen that the leak confidence model presented in this chapter is not only able to find leaking containers,

```
Container:18296328 type: class java.util.ArrayList
        (LC: 0.921, SC: 0.985, MC: 0.011)
---cs: LeakExample:25 (0.327)   // first container

Container:24764524 type: class java.util.ArrayList
        (LC: 0.008, SC:0.035, MC: 0.241)
---cs: LeakExample:57 (0.031)  // third container

Container:16224256 type: class java.util.ArrayList
        (LC: 0.007, SC: 0.037, MC: 0.180)
---cs: LeakExample:39 (0.033)  // second container
Data analyzed in 47275ms
 (a) Report for the bug from IBM developerWorks, using 1/15gc


Container:18296328 type: class java.util.ArrayList
        (LC: 0.845, SC: 0.971, MC: 0.01)
---cs: LeakExample:25 (0.327)   // first container

Container:12077888 type: class java.util.ArrayList
        (LC: 0.008, SC:0.031, MC: 0.245)
---cs: LeakExample:57 (0.031)  //third container

Container:16224256 type: class java.util.ArrayList
        (LC: 0.005, SC: 0.033, MC: 0.141)
---cs: LeakExample:39 (0.032)   //second container
Data analyzed in 26198ms
 (b) Report for the bug from IBM developerWorks, using 1/50gc
```

Figure 5.9: Report for the leak bug from IBM developerWorks.

but can also help to understand run-time performance issues and to find execution bottlenecks.

Leak-Free Programs

The tool was also used to analyze several programs that have been used widely and tested extensively for years, and do not have any known memory leaks. Table 5.4 shows the confidence values computed for these programs. The goal of this experiment was to determine whether the tool produced any false positives on these (almost certainly) leak-free programs. The low confidence values reported by the tool are the expected and desirable outcome for this experiment.

antlr	4.1E-5	chart	2.7E-6	fop	1.3E-5
hsqldb	4.4E-7	jython	5.0E-8	luindex	9.1E-5
lusearch	2.3E-2	pmd	4.3E-6	xalan	5.2E-5
jflex	1.8E-7				

Table 5.4: Confidences for leak-free programs.

5.3.2 Static and Dynamic Overhead

This section describes our study of the overhead introduced by our technique. This study utilizes the three real-world bugs described earlier, as well as a set of Java programs shown in Table 5.4. The benchmark set includes 9 programs from the DaCapo suite [40], and jflex, a lexer generator. For each DaCapo program, we ran it with default workload size, and the input for jflex is a grammar file corresponding to a finite state machine with 21769 states. We ran the instrumented programs with rates 1/15gc and 1/50gc. The maximum JVM heap size for each run was set to 512MB (JVM option `Xmx512m`). For each sampling rate, we ran the programs once with the default initial heap size and once with a large initial heap size (JVM option `Xms512m`), in order to observe different numbers of gc-events. We checked the generated reports under the four configurations (i.e., 1/15gc with 512M heap, 1/50gc with 512M heap, 1/15gc with default heap, and 1/50gc with default heap) and found that for all programs, the top five containers reported by our tool under these configurations are the same. The remaining containers (other than the top five) in each report had fairly small confidence values, and therefore, we can assume that the precision loss under configurations with 512M initial heap or 1/50gc sampling rate is small enough so that it is not hurtful in the practical use of the tool.

Program	*#IS*	*#IS$_e$*	*IT*(s)
antlr	176	123	87
chart	894	867	202
fop	1378	1375	125
hsqldb	684	674	116
jython	443	416	135
luindex	442	409	65
lusearch	442	388	81
pmd	814	690	111
xalan	755	752	114
jflex	522	438	92
bug 1	3109	2768	487
bug 2	3105	2770	502
specjbb	74	73	142

Table 5.5: Static analysis running time.

Table 5.5, Table 5.6, and Table 5.7 describe the static analysis running time, the dynamic overhead of the tool when using 1/15gc, and when using 1/50gc sampling rate, respectively. In Table 5.5 columns IS and IS_e represent the numbers of call sites instrumented without and with employing escape analysis, respectively. Column IT ("instrumentation time") represents the static overhead of the tool—that is, the time (in seconds) it takes to produce the escape-analysis-based instrumented version of the original code. Column RT_o ("running time") in both Table 5.6 and Table 5.7 contains the original running times of the programs.

The dynamic overhead of the approach is described in the remainder of Table 5.6 and Table 5.7. Columns GC_d and GC_l show the numbers of gc-events with the default and with the large initial heap size, respectively. Similarly, RT_d and RT_l show the program running times with these two choices of initial heap size. OH_d and OH_l, respectively, represent run-time overhead introduced by our tool when executed with

Program	(a)	(b) 1/15gc				(c)	
	RT_o(s)	$\#GC_d$	RT_d(s)	$\#GC_l$	RT_l(s)	$\%OH_d$	$\%OH_l$
antlr	17.9	387	18.4	10	18.1	2.8%	1.1%
chart	8.5	5368	38.0	185	35.4	347.0%	316.4%
fop	4.5	693	8.6	24	7.8	91.1%	73.3%
hsqldb	4.3	54	4.7	8	4.4	9.3%	2.3%
jython	7.3	1653	31.8	126	28.2	335.6%	286.3%
luindex	19.5	1446	24.4	40	23.7	25.1%	21.5%
lusearch	2.9	418	9.1	21	3.9	213.8%	34.5%
pmd	5.9	2938	26.9	716	18.4	355.9%	211.9%
xalan	1.4	655	7.7	30	4.0	450%	185.7%
jflex	45.1	4171	170.7	1493	130.3	278.5%	188.9%
bug 1	–	18630	600	7420	600	–	–
bug 2	38.1	512	53.0	243	42.3	39.1%	11.0%
specjbb	–	18605	3600	15080	3600	–	–
average	–	–	–	–	–	195.3%	121.2%

Table 5.6: Dynamic overhead I: (a) original running time; (b) running with 1/15gc rate; (c) run-time overhead.

the default heap size and the 512M heap size, which correspond to RT_d and RT_l in columns (c). More specifically, these two values are computed as

$OH_d = (RT_d - RT_o)/RT_d$

$OH_l = (RT_l - RT_o)/RT_l$

The average overhead for each configuration is added at the bottom of the tables. For `bug 1` and `specjbb`, we ran the test case for 10 minutes and an hour, respectively, because the execution of these two programs does not terminate.

Applying escape analysis reduces the number of call sites that need to be tracked (the reduction varies from 3 to 124 call sites), while still maintaining reasonable instrumentation time. For the purpose of illustration and comparison, the dynamic overhead under different configurations is shown in Figure 5.10.

Program	(a)	(b) 1/50gc				(c)	
	RT_o(s)	$\#GC_d$	RT_d(s)	$\#GC_l$	RT_l(s)	$\%OH_d$	$\%OH_l$
antlr	17.9	387	18.4	10	18.1	2.8%	1.1%
chart	8.5	4109	36.5	185	35.1	329.4%	312.9%
fop	4.5	545	8.9	24	6.4	97.8%	42.2%
hsqldb	4.3	54	4.7	8	4.4	9.3%	2.3%
jython	7.3	1440	31.4	126	28.5	330.1%	290.4%
luindex	19.5	1390	23.9	40	23.7	22.6%	21.5%
lusearch	2.9	326	8.2	23	3.2	182.8%	10.3%
pmd	5.9	2766	25.2	37	6.6	327.1%	11.9%
xalan	1.4	605	6.2	18	3.7	342.9%	164.3%
jflex	45.1	2126	165.8	665	88.05	267.6%	95.2%
bug 1	–	11457	600	1983	600	–	–
bug 2	38.1	413	52.2	37	42	37.0%	10.2%
specjbb	–	16789	3600	10810	3600	–	–
average	–	–	–	–	–	177.2%	87.5%

Table 5.7: Dynamic overhead II: (a) original running time; (b) running with 1/50gc rate; (c) run-time overhead.

Using the same sampling rate, running a program with a large initial heap size takes less time, because this configuration reduces the number of gc-events, which in turn reduces the numbers of thread synchronizations, disk accesses, and object graph traversals performed by the dynamic analysis. For the same reason, decreasing the sampling rate reduces the run-time overhead.

By employing both larger initial heap and smaller sampling rate, the average run-time overhead for the programs can be reduced to 87.5%. Such overhead is acceptable for bug detection, but it may be too high for production runs. One possible approach to reducing overhead is to selectively instrument a program. Based on the manifestation of the bug, developers may have preferences and hints as to where to focus the effort of the tool. For example, certain parts of the program that are decided not to be the cause of the bug do not need to instrumented and tracked at

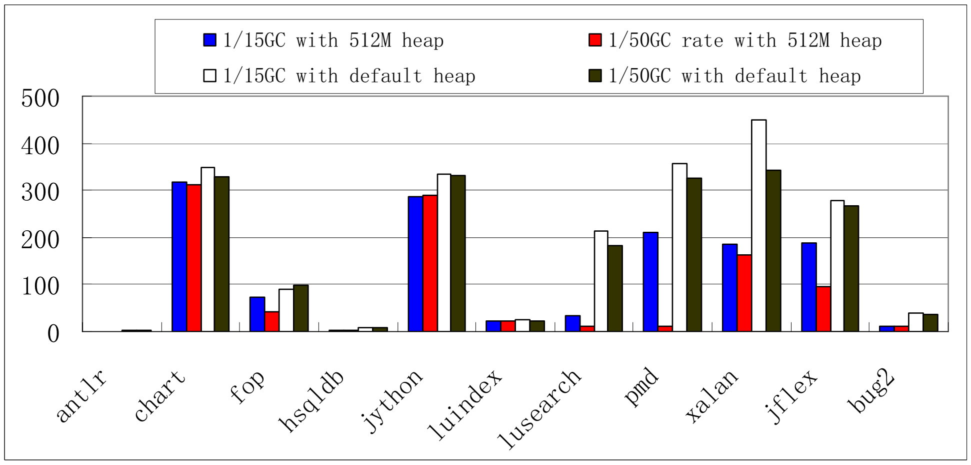


Figure 5.10: Dynamic overhead under different configurations.

run time. The continuous optimization of the tool is part of our future work. For example, the optimization may focus on reducing threads synchronizations within the dynamic analysis. Our current implementation uses JVMTI, which runs agents and invokes event callbacks in threads. The running of these threads adds synchronization overhead. In addition, the retrieval of a container object from its tag through JVMTI also contributes to the execution overhead. Hence, another part of our future work is to re-implement the tool within an existing open-source JVM, such as the Jikes RMV [70], in order to avoid the overhead caused by JVMTI.

5.4 Summary and Interpretation

This chapter presents a novel technique for detecting memory leaks for Java. Unlike existing "from-symptom-to-cause" dynamic analyses for leak detection, the proposed approach employs a higher-level abstraction, focusing on container objects and their operations. The kind of data-based activity considered in this chapter is, thus, data flow between containers and the rest of the program. The approach uses both memory contribution and staleness contribution to decide how significant a container's leaking behavior is. This chapter presents an implementation of this technique and a set of experimental studies, demonstrating that the proposed tool can produce precise bug reports at a practical cost. These promising results indicate that the technique and any future generalizations are worth further investigation.

In this work, method semantics is provided by programmers as annotations. It would be interesting and useful to develop an analysis that could automatically infer such semantic information from the program source code. In Chapter 7, we propose such a static analysis that can automatically infer these annotations. This analysis may also be used by the approach described in this chapter, in order to understand method semantics so that programmers' annotation efforts can be avoided.

This is the first memory leak detection technique that explicitly uses semantic information about the target program to help problem diagnosis. Given the size and complexity of today's large-scale applications, there is little hope that a completely automatic analysis can precisely identify performance bottlenecks. We believe that developers' insight is key to improving the precision and real-world usefulness of similar performance analysis tools. Research presented in this chapter has demonstrated that a program-semantics-aware bloat detector can produce much more relevant information than an analysis that does not rely on any semantic information. The next chapter presents another semantics-aware memory leak detection technique, called LeakChaser. In that work, the approach relies on user-provided specifications of object lifetime relationships to help diagnose memory leaks.

CHAPTER 6: Making Sense of Transaction Behaviors: Memory Leak Detection Using LeakChaser

A Java memory leak is an important source of memory bloat which can have significant impact on program performance and scalability. Existing leak detectors attempt to find root causes of memory leaks by starting from the objects that look suspicious (i.e., the leak symptom) and traversing the object graph on the heap. Due to the complexity of this graph, such an attempts is often a heuristics-based process that ends up reporting a sea of likely problems with the true causes being buried among them. A detailed discussion of existing memory leak detectors can be found in the previous chapter. Because a memory leak often occurs due to the inappropriate handling of certain events, and existing tools profile the whole program execution without any focus, there is little hope that a completely automated tool can precisely pinpoint the problematic area(s) for large-scale Java applications. While the container profiling work (described in the previous chapter) avoids heuristics by focusing on containers [158], the root causes of many memory leaks are not containers, but instead cached references that the programmer forgot to invalidate, as represented by the leak cases in SPECjbb2000 and Eclipse bug #115789 (Section 6.4).

For a more focused leak detector, it is necessary to take advantage of human insights and use programmer specifications to guide leak detection. While prior work proposes heap assertions that can be checked during GC [1, 9, 112, 149], these low-level assertions may only be employed by programmers who have deep understanding of an application and its algorithms, data structures, etc. This requirement limits significantly their usefulness in leak detection for existing Java applications (where no assertions were written during development), as very few performance experts have sufficient program knowledge to use these assertions during postmortem tuning and debugging.

Insight Almost all memory leaks we have studied are in regularly occurring program events: unnecessary references can quickly accumulate and cause the memory footprint to grow if each such event forgets to clean up a small number of references.

The techniques proposed in this chapter are based on the following two observations about these frequently executed code regions. First, in these regions there usually exist implicit invariants among lifetimes of objects (e.g., "objects a and b must die together," or "the old configuration object must die before the new configuration object is created"). When such invariants are violated, memory leaks result. It is often not easy to use heap assertions from prior work (which are typically based on reachability properties in the object graph) to express these lifetime relationships. Second, for each such region, there often exist objects that are strongly correlated with the liveness of the entire region. For example, consider a (web or database) transaction in an enterprise Java application. A typical lifecycle consists of a sequence of events such as the creation of the transaction object, the creation of all other objects used in this transaction, the deallocation of these (other) objects, and the deallocation of the transaction object. For this lifecycle, the transaction object is the first one that is created and the last one that is reclaimed, and it is thus the object that controls the liveness of this entire transaction region.

Our proposal We propose a three-tier approach that exploits these insights to help a programmer quickly identify unnecessary references leading to a memory leak. This approach can be used by both novices and experts to diagnose memory problems. The key idea is to introduce high-level semantics by explicitly considering coarse-grained events where leaks are observed. We refer to these events as *transactions*. As extensions of—and inspired by—enterprise transaction models (e.g., EJB transactions), our transactions describe frequently executed code regions with user-defined boundaries. Objects associated with a transaction fall into three categories: (1) transaction identifier object, (2) transaction-local objects, and (3) objects shared among transactions.

The identifier object of a transaction controls the lifetime of the transaction: all transaction-local objects should be created after this object is created and should die before it dies. Only shared objects are allowed to live after the identifier object dies. Next, we introduce the three tiers of our approach in descending order of their levels of abstraction, which is often the order in which a programmer uses our tool to solve a real-world problem.

Tier H (the high-level approach): At this highest level of the approach, our framework attempts to automatically *infer* transaction-local and shared objects from the execution, while the programmer only needs to specify transaction boundaries and identifier objects. The tool starts to work in inference mode, and once objects shared among transactions are identified, it switches to check whether the (inferred) shared objects can actually be used in other transactions. Violations are reported if these shared objects are not used for a certain period of time. The programmer's task at this level is the easiest to perform: we found that even when we had not studied a program before, we could quickly identify these (coarse-grained) events in order to perform the diagnosis.

Tier M (the medium-level approach): In this tier, the user needs to specify not only the boundary of a transaction and the transaction identifier object, but also the objects shared across transactions, which the user does not specify in tier H.

LeakChaser *checks* the given specifications, and reports violations if the specified transaction-local objects in one instance of the transaction escape to another instance. While employing this tier of the tool requires the user to have a deeper understanding of the program and the likely cause of the problem, it can generate a more precise report.

Tier L (the low-level approach): This lowest level is essentially an assertion framework that allows the user to specify lifetime invariants for focused memory leak detection. The framework contains binary assertions that directly express object lifetime relationships. For instance, one important assertion is to specify that object a must die before object b. This assertion fails only when LeakChaser observes definitive evidence, e.g., b is dead while a is still live.

Compared to reachability-based assertions [1,9,112,149], our framework has three advantages. First, passing/failing of our assertions does not depend on where GC runs are triggered and thus, assertion checking does not produce false positives. Second, program locations where our assertions are placed have no influence on the evaluation of these assertions. For example, we can assert that object a dies before object b immediately after they are created, while a reachability-based assertion such as *assertDead* in prior work has to be placed in a location where the asserted object is about to become unreachable [1]. Third, our framework can be used to assert arbitrary objects whose lifetimes are correlated due to some high-level semantics (e.g., events), while a reachability-based assertion can work only on objects that have low-level structural relationships (e.g., reachability in the object graph).

Of course, using this assertion framework requires one to understand considerable design and implementation details of the program such as how a data structure is constructed. However, performing this level of diagnosis can give the user a very precise report. Hence, these assertions can be added by developers during coding, which may ease significantly the diagnosis of memory problems when performance degradation is observed. In fact, the transaction-based properties described in the other two tiers are translated by our framework into these low-level assertions at run time. Section 6.2 presents details about this translation.

As the level of abstraction decreases (from tier H to tier L), the diagnosis becomes more focused. Figure 6.1 illustrates the process of memory leak detection using this framework. In our experience, LeakChaser is especially useful to a performance expert who is unfamiliar with the program code, since he or she can follow an iterative process that involves all these levels of diagnosis (i.e., from tier H down to tier L). The programmer starts with the tier H analysis with little program knowledge and insight. By repeating a higher-level analysis a few times (with refined specifications) and inspecting its reports, the programmer gains a deeper understanding of the program as well as more insight into the problem. As a result, she will be able to move on to a lower-level analysis for a more focused diagnosis. Very often, this process ends up narrowing down the information to the exact cause of the leak.

For large-scale applications, LeakChaser allows programmers to specify transactions at *clients* of these applications, without digging into application implementation

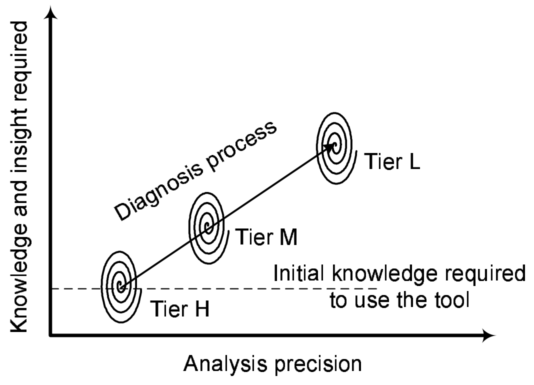

Knowledge and insight required
Diagnosis process
Tier L
Tier M
Tier H
Initial knowledge required to use the tool
Analysis precision

Figure 6.1: Illustration of the diagnosis process. Spirals at each level indicate that a user may need to run each tier multiple times with refined specifications (e.g., smaller transaction regions, more precise shared object specifications, etc.) to gain sufficient knowledge to proceed to a lower tier.

details. For example, to diagnose problems in a large database system, the programmer only needs to create transactions at client programs that perform database queries. We found that this feature of the tool is quite useful in simplifying the diagnostic task: all previous techniques require a programmer to understand a fair amount of low-level details of the system before he can start the diagnosis. This burden is reduced significantly by LeakChaser.

Implementation and experiments We implemented our approach in Jikes RVM 3.1.0 (`http://jikesrvm.org`), a high-performance Java-in-Java virtual machine, and successfully applied it to real leaks in large-scale applications such as Eclipse and MySQL. The implementation techniques are discussed in detail in Section 6.3. Section 6.5 evaluates analysis expenses. We add one extra word in the header of each run-time object and this space is used to store the assertion information for the object. The overall space overhead of the tool is less than 10%, including both extra header space and memory used to store the metadata of our analysis. Using the optimizing compiler and the Immix garbage collector [16], the current implementation imposes an average slowdown of 2.3× for GC only and 1.1× for overall executions for the framework infrastructure (i.e., no assertions added). Additional overhead is incurred for checking and inferring specifications. For example, after adding assertions to SPECjbb2000, 256236 assertions were executed, and overall slowdown was 5.5×. While the overhead is probably high for production runs, we found it acceptable for performance tuning and memory leak diagnosis.

Section 6.4 presents six case studies on real-world memory leak problems. Among these problems, four are true leaks and for the remaining two, programmers experienced high memory footprints but were not sure whether or not there were leaks. Using our tool, we have quickly identified root causes for the true leaks, and found reasons that could explain the high memory consumption for the other two cases. In SPECjbb2000, in addition to the already-known leak, we found memory issues that have not been reported previously and actually cause more severe performance degradation than the already-known leak.

We observed significant performance improvements after fixing these problems. The experimental results strongly indicate that the proposed three-tier diagnosis methodology can be adopted in real-world development and tuning to find and prevent memory leaks, and LeakChaser is useful in helping a programmer quickly identify unnecessary references that lead to leaks and other memory issues.

The contributions of this work are:

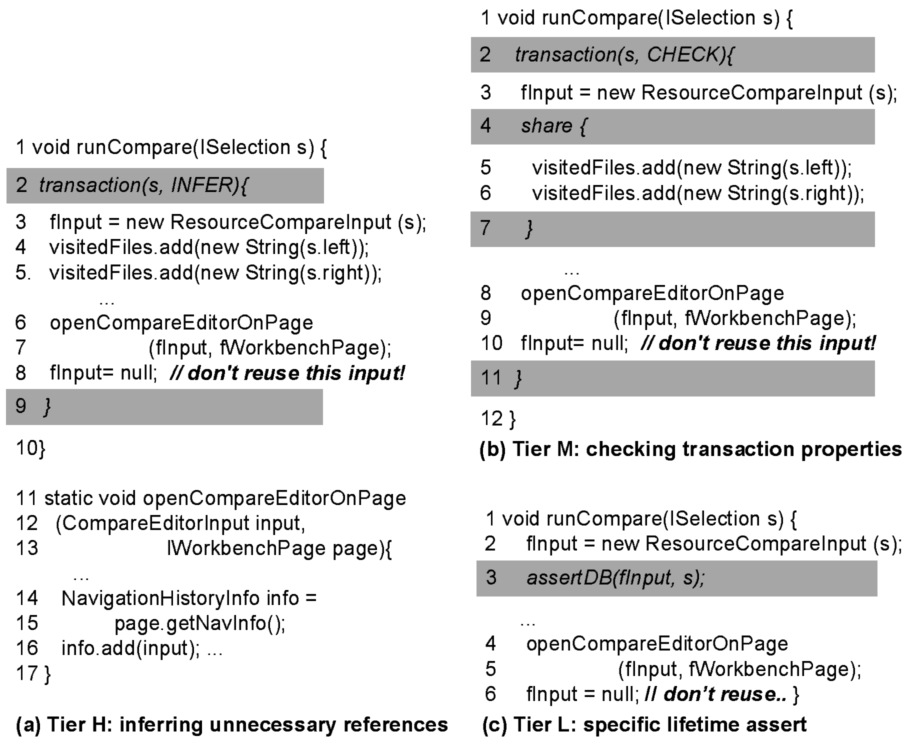
- A three-layer methodology that introduces different levels of abstraction to help both experts and novices understand and diagnose memory leak problems.

- A new heap assertion framework that allows programmers to assert object liveness properties instead of using reachability to approximate liveness.
- An implementation of LeakChaser in Jikes RVM that piggybacks on garbage collection to check assertions.
- Six case studies demonstrating that LeakChaser can help a programmer unfamiliar with the program source code to quickly find root causes of memory leaks.

6.1 Overview

We illustrate our technique using a simplified version of a real-world memory leak (Eclipse bug #115789). Figure 6.2(a) shows the code that contains the leak. This bug can be easily reproduced on Eclipse 3.1.2 when comparing the structures of two large JAR files multiple times using an Eclipse built-in comparison option. Method `runCompare` implements a comparison operation (in plugin `org.eclipse.compare`). It is invoked every time the comparison option is chosen, and this information can be easily obtained from the Eclipse plugin APIs. The method takes a parameter of type `ISelection` that contains information about the two selected files to be compared. A `ResourceCompareInput` object is first created using this parameter (line 3). This object is fairly heavyweight as it caches the complete structures of the two files. The names of these files are then recorded in a list `visitedFiles` (lines 4 and 5), and this list may be used by the workspace GUI upon receiving a user request to view the history.

Next, `openCompareEditorOnPage` is invoked to open a compare editor in the workspace GUI that shows the differences between the two files. The (simplified)



```
1 void runCompare(ISelection s) {
2 transaction(s, INFER){
3   fInput = new ResourceCompareInput (s);
4   visitedFiles.add(new String(s.left));
5.  visitedFiles.add(new String(s.right));
        ...
6   openCompareEditorOnPage
7               (fInput, fWorkbenchPage);
8   fInput= null;  // don't reuse this input!
9 }
10}

11 static void openCompareEditorOnPage
12  (CompareEditorInput input,
13                  IWorkbenchPage page){
      ...
14   NavigationHistoryInfo info =
15          page.getNavInfo();
16   info.add(input); ...
17 }
```

(a) Tier H: inferring unnecessary references

```
1 void runCompare(ISelection s) {
2   transaction(s, CHECK){
3   fInput = new ResourceCompareInput (s);
4   share {
5    visitedFiles.add(new String(s.left));
6    visitedFiles.add(new String(s.right));
7   }
        ...
8   openCompareEditorOnPage
9               (fInput, fWorkbenchPage);
10  fInput= null;  // don't reuse this input!
11 }
12 }
```

(b) Tier M: checking transaction properties

```
1 void runCompare(ISelection s) {
2   fInput = new ResourceCompareInput (s);
3   assertDB(fInput, s);
  ...
4   openCompareEditorOnPage
5               (fInput, fWorkbenchPage);
6   fInput = null; // don't reuse.. }
```

(c) Tier L: specific lifetime assert

Figure 6.2: Overview of the technique, illustrated using a simplified version of Eclipse bug #115789.

method body is shown at lines 11–17. In this method, the new `ResourceCompareInput` object is cached in a `NavigationHistoryInfo` object retrieved from the current workbench page for future save or restore operations (lines 14–16). Caching this heavyweight input object is actually the cause of the leak: the structures of the two files keep being created and referenced. As a result, the memory footprint grows quickly, and Eclipse runs out of memory. Note that in the real Eclipse code, this cache operation (at line 16) and the `runCompare` method are in two different plugins, which makes it particularly hard to diagnose the problem as these plugins are written by different groups of programmers. The two plugins communicate only through public interfaces and it is unclear to programmers of one plugin what side effects the other plugin can have. It is interesting to see that despite the fact that the developers of plugin `org.eclipse.compare` are aware that the input must be cleared after the comparison (e.g., in fact, the comment at line 8 is from the real code), these references are still unnecessarily kept somewhere out of their scope.

While these plugins have been studied in previous work [20, 71], we start with the tier H approach to simulate what a programmer would do at the beginning of a diagnostic task. Hence, our experience with this case, to a large degree, reflects how a programmer unfamiliar with a program can use the tool to diagnose memory leaks. We first need to identify a transaction and let the tool infer unnecessary references for the transaction. This is easy: as the comparison is the regularly occurring event leading to the leak, it is a natural idea to let the transaction cross the entire body of method `runCompare`, as shown in Figure 6.2(a). A transaction creation (at line 2) takes two parameters: a transaction identifier object and a mode in which the transaction works. The identifier object must be unique per transaction and must be created before the transaction starts. Here we choose s to be the identifier object, because s refers to an `ISelection` object that is created per comparison operation before `runCompare` runs. Constant *INFER* informs the tool to run in inference mode. We found that the identifier object is usually easy to find for a transaction: a transaction body often crosses a method that is invoked on an existing object. In many cases, either the receiver object or a parameter object of the method can be selected as the transaction identifier object (as long as it is created per transaction and does not cross multiple transaction invocations).

For each transaction, its *spatial boundary* is specified by a pair of curly brackets (i.e., $\{\ldots\}$) and its *temporal boundary* is defined by the lifetime of its identifier object. To find unnecessary references, we focus on objects created within its spatial boundary, and check their lifetimes against its temporal boundary. Informally, the semantics of a transaction is such that each *transaction-local* object must die[4] before the identifier object, and only *shared* objects can live after the identifier object dies. In inference mode, where a programmer does not explicitly specify local and shared objects, our tool infers shared objects automatically: objects created within the spatial boundary are treated as shared objects if they are still live at the time the identifier object dies. Once an object is marked as shared, the tool starts tracking its *staleness* and records a violation if it is not used for a given period of time, based on the intuition that an object that one transaction intends to share with other transactions should be used outside its creating transaction. Violations are aggregated and eventually reported in the order of their frequencies.

The following example shows a typical violation report that includes information about the transaction where the violating object is created, its creation site, violation type, the number of times this violation occurs, and the (heap) reference paths that lead to this object at the time of the violation. Each line in a reference path shows the information of an object on the path. While only one reference path is shown here (for illustration) for this violation, multiple paths were actually reported by the tool.

```
Transaction specified at:
   CompareAction:runCompare(ISelection), ln 2
Violating objects created at:
   CompareEditorInput:createOutlineContents
```

[4]In this chapter, an object is considered to die immediately after it becomes unreachable in the object graph.

```
                  (widgets.Composite), ln 439
Violation type:
   Objects shared among transactions are not used
Frequency: 4
Reference paths:
    Type: ArrayList, created at: NavigationHistory:
                                <init>(WorkbenchPage), ln 44
--> Type: Object[], created at: ArrayList: <init>(I), ln 119
--> Type: NavigationHistoryEditorInfo, created at:
                 NavigationHistory: createEntry(...), ln 553
--> Type: ResourceCompareInput, created at: CompareAction:
                                runCompare(ISelection), ln 3
```

This reference path makes it easy to explain why this `Resource CompareInput` object is shared: the reference path exists because the object is cached (transitively) by a `NavigationHistory` object. However, this is not the only violation in the tier H report. The two strings created at lines 4 and 5 (together with many other objects) are also in the report as they are not used at all if there is no user request to perform history-related operations. It takes some time to inspect this report, as it contains a total of 36 violations. By ranking violations (based on frequencies and other factors), it is not hard for us to eliminate many of them that are lower on the list and that are obviously not leaks. For example, after inspecting the warnings, we determine that these small strings (and other similar objects) are not the major cause of the leak, because the growth of used memory is so significant that it is unlikely to be due to small objects (especially because their frequencies are not significantly higher).

As we obtain this knowledge (regarding these strings and other irrelevant violating objects), we move down to tier M for a more focused analysis. In Figure 6.2(b), we explicitly mark these strings (and others created by the two `add` calls) as shared (with a *share* region) and let our tool run in checking mode. Objects created in the transaction but not in the share region are marked (implicitly) as transaction-local objects. Our tool then ignores objects marked as shared and reports violations only when transaction-local objects are found live after the transaction identifier object dies. This gives us a much cleaner report (with 4 violations), and of course, the violation shown earlier appears in the report, indicating the `ResourceCompareInput` object is referenced from somewhere else.

After these two rounds of diagnosis, we have gained implementation knowledge (e.g., the general procedure of performing a comparison operation) and some insights into the problem (e.g., this leak might be caused by an unnecessary reference somewhere in `NavigationHistory`). During code inspection, we become interested in the comments in line 8 of the code: this statement has a clear purpose of releasing the input object, but why did we see it is still reachable in both tier M and tier H reports? Having this question in mind, we decide to perform a detailed (tier L) diagnosis using a lifetime assertion (e.g., `assertDB` asserts a “dies-before” relationship), as shown in Figure 6.2(c). This single assertion fails, which confirms our suspicions.

After some code inspection (with the help of the reference path associated with the violation), we found that `NavigationHistory` allows a user to step backward and forward through browsed editor windows. It keeps a list of `NavigationHistoryEntry` objects, each of which points to an `EditorInfo` object that, in turn, points to a `CompareEditorInput` object, the root of a data structure that holds the diff results. `NavigationHistory` uses a count to control the number of `EditorInfo` objects it caches, and removes an `EditorInfo` if the count drops to zero. However, `NavigationHistory` does not correctly decrement this count in some cases, leading to unnecessary references.

This example clearly shows the difficulty of diagnosing real-world memory problems. The root cause of this bug is that the `NavigationHistory` entries are cached and not getting removed due to reference-counting problems. This occurs entirely on the UI side. The developers of the `compare` plugin may have never thought that calling a general interface to open an UI editor can cause a big chunk of memory to be cached. The complexity of the large-scale code base and the limited knowledge of each individual developer strongly call for LeakChaser's step-by-step approach. Such tool support can help a programmer who starts without insights into the program or its leak, to systematically explore the leaky behavior in order to pinpoint its cause.

6.2 Assertions and Transactions

This section presents a formalism to describe our analysis of unnecessary references. The presentation proceeds in three steps. First, we define a simple garbage-collected language `assert` and its abstract syntax. We next give a semantics of this language by focusing on its *traces*. Each trace is a sequence of basic events (e.g., alloc, dealloc, and use) on objects, transaction events (e.g., start and end), and assertions. Finally, we formulate assertion checking and inference of unnecessary references as judgments on traces (i.e., trace validation). Note that in our implementation (discussed in Section 6.3), checking and inference are performed during GC runs. Here the trace collection phase and the trace validation phase are separated for ease of presentation and formal development.

Language `assert` The abstract syntax and the semantic domains for language `assert` are defined in Figure 6.3. The program has a fixed set of global reference-typed variables. An allocation site has the form $a = \text{new ref}^o$, where o stands for an allocation site ID defined at compile time.

There are two types of assertions: `assertDB` and `assertDBA`. `assertDB`(a, b) asserts that object (pointed to by) a must die before (or together with) object (pointed

$$
\begin{array}{llll}
\text{Variables} & & a,\, b & \in \ \mathsf{V} \\
\text{Allocation sites} & & o & \in \ \mathsf{O} \\
\text{Instance fields} & & f & \in \ \mathsf{F} \\
\text{Labels} & & l & \in \ \mathsf{N} \\
\text{Assertions} & e & ::= & \text{assertDB}(a,\, b) \mid \text{assertDBA}(a,\, b) \\
\text{Transactions} & t & ::= & \text{transaction}(a,\, m)\{\ tb\ \} \\
\text{Trans bodies} & tb & ::= & s\ ;\ tb \mid \text{share}\ \{\ s\ \}\ tb \mid \epsilon \\
\text{Trans modes} & m & ::= & \text{CHECK} \mid \text{INFER} \\
\text{Statements} & s & ::= & a = b \mid a = \text{new ref}^{o} \mid a = \text{null} \mid \text{while} \ldots \\
 & & & \mid a = b.f \mid a.f = b \mid e \mid \text{gc} \mid \text{if} \ldots \mid s\ ;\ s \\
\text{Program} & p & ::= & s\ ;\ p \mid t\ ;\ p \mid \epsilon
\end{array}
$$

(a)

$$
\begin{array}{llll}
\text{Labeled object} & \hat{o} & ::= & o^{l} \ \in\ \Phi \\
\text{Environment} & \rho & \in & \mathsf{V} \rightarrow \Phi \cup \{\perp\} \\
\text{Heap} & \sigma & \in & \Phi \times \mathsf{F} \rightarrow \Phi \cup \{\perp\} \\
\text{Operation} & \tau & ::= & \langle \hat{o},\, \text{A} \mid \text{D} \mid \text{U} \rangle^{o} \mid \langle \hat{o},\, m,\, \text{S} \mid \text{E} \rangle^{t} \\
 & & & \mid \langle \text{S} \mid \text{E} \rangle^{s} \mid \langle \hat{o}_a,\, \hat{o}_b,\, \text{DBA} \mid \text{DB} \rangle^{a} \\
\text{Traces} & \alpha & ::= & \tau\ ,\ \alpha \mid \epsilon
\end{array}
$$

(b)

Figure 6.3: A simple `assert` language: (a) abstract syntax (b) semantic domains.

to by) b. Another assertion `assertDBA`(a, b) specifies that object a must die before a new object is created by object b's allocation site (DBA is short for "Dies Before Allocation"). This assertion is useful to enforce a "replaces" relationship between two objects. For example, it can be used to enforce that an old (invalid) screen configuration is appropriately released before a new screen configuration is created upon repainting of an interface in a GUI program. As another example, in Figure 6.2(c), instead of using `assertDB`, we can also write `assertDBA(fInput, fInput)` to assert that the current `ResourceCompareInput` object must die before the next `ResourceCompareInput` object (created by the same allocation site) is allocated. While our framework includes a few other assertions, they are not discussed because they can be implemented using these two basic assertions.

To ease the formal development, a GC run can only be triggered by a `gc` statement, which traverses the object graph to reclaim unreachable objects.

Note that we do not allow the nesting of transactions. While this is easy to implement in our framework, we have not found it helpful for pinpointing memory leak causes.

Each run-time object is labeled with its allocation site o and an integer l denoting the index of this object among all created by the allocation site. Environment and heap are defined in standard ways. A trace is a sequence of operations. There are four types of operations, each of which is a tuple annotated with a type symbol: o for object operation, t for transaction operation, s for share region operation, and a for assert operation. Each *object operation* is an object and event pair, where an event can be either A (i.e., Alloc), D (i.e., Dealloc), or U (i.e., Use). Each *transaction operation* is a triple containing its identifier object, the mode (i.e., CHECK or INFER), and whether this operation corresponds to the start of the transaction (S) or its end (E). A *share region operation* has only a type that indicates whether it is the start or the end of the region. Each *assert operation* contains two input objects and an assertion type (DBA for `assertDBA` and DB for `assertDB`). While the language does not explicitly consider threads, our analysis is thread-safe as each transaction creation is a thread-local event. Different transaction instances created by different threads can exist simultaneously for the same transaction declaration. Trace collection and

$$\frac{\begin{array}{c}\alpha' = \alpha \circ \langle \hat{o}, \mathrm{A}\rangle^{o} \\ \hat{o}.\mathit{allocsite} = o \qquad \hat{o}.l = \mathrm{newObjIndex}(o)\end{array}}{a = \mathrm{new\ ref}^{o}, \rho, \sigma, \alpha \Downarrow \rho[a \mapsto \hat{o}], \sigma, \alpha'} \quad (\textsc{New})$$

$$a = b.f, \rho, \sigma, \alpha \Downarrow \rho[a \mapsto \sigma(\rho(b).f)], \sigma, \alpha \circ \langle \rho(b), \mathrm{U}\rangle^{o} \quad (\textsc{Load})$$

$$a.f = b, \rho, \sigma, \alpha \Downarrow \rho, \sigma[\rho(a).f \mapsto \rho(b)], \alpha \circ \langle \rho(a), \mathrm{U}\rangle^{o} \quad (\textsc{Store})$$

$$\frac{s, \rho, \sigma, \alpha \circ \langle \rho(a), m, \mathrm{S}\rangle^{t} \Downarrow \rho', \sigma', \alpha' \qquad \alpha'' = \alpha' \circ \langle \rho(a), m, \mathrm{E}\rangle^{t}}{\mathrm{transaction}(a, m)\{s\}, \rho, \sigma, \alpha \Downarrow \rho', \sigma', \alpha''} \quad (\textsc{Tran})$$

$$\frac{s, \rho, \sigma, \alpha \circ \langle \mathrm{S}\rangle^{s} \Downarrow \rho', \sigma', \alpha' \qquad \alpha'' = \alpha' \circ \langle \mathrm{E}\rangle^{s}}{\mathrm{share}\{s\}, \rho, \sigma, \alpha \Downarrow \rho', \sigma', \alpha''} \quad (\textsc{ShareReg})$$

$$\mathrm{assertDB}(a, b), \rho, \sigma, \alpha \Downarrow \rho, \sigma, \alpha \circ \langle \rho(a), \rho(b), \mathrm{DB}\rangle^{a} \quad (\textsc{AssertDB})$$

$$\mathrm{assertDBA}(a, b), \rho, \sigma, \alpha \Downarrow \rho, \sigma, \alpha \circ \langle \rho(a), \rho(b), \mathrm{DBA}\rangle^{a} \quad (\textsc{AssertDBA})$$

$$\frac{\mathit{tr} = \circ\{\langle \hat{o}, \mathrm{D}\rangle^{o} | \hat{o} \in \sigma \wedge \hat{o} \notin \mathrm{reachable}(\rho, \sigma)\} \qquad \forall \langle \hat{o}, \mathrm{D}\rangle^{o} \in \mathit{tr} : \hat{o} \notin \sigma' \qquad \alpha' = \alpha \circ \mathit{tr}}{\mathrm{gc}, \rho, \sigma, \alpha \Downarrow \rho, \sigma', \alpha'} \quad (\mathrm{GC})$$

$$\frac{s_1, \rho, \sigma, \alpha \Downarrow \rho', \sigma', \alpha' \qquad s_2, \rho', \sigma', \alpha' \Downarrow \rho'', \sigma'', \alpha''}{s_1; s_2, \rho, \sigma, \alpha \Downarrow \rho'', \sigma'', \alpha''} \quad (\textsc{Comp})$$

Figure 6.4: Operational semantics.

validation are also performed on a per-thread basis. A special symbol $\perp$ is added to the heap and the environment to represent a `null` value.

Language semantics An operational semantics is given in Figure 6.4. A judgment of the form s, ρ, σ, $\alpha \Downarrow \rho'$, σ', α' starts with a statement s, which is followed by environment ρ, heap σ, and trace α. The execution of s terminates with a final environment ρ', heap σ', and trace α'. Trace concatenation is denoted by $\circ$. Rules New, AssertDB, AssertDBA, and Comp are defined as expected. In rules Load and Store, trace α is augmented with an object use event. In rule Tran, for each transaction, the two transaction events (i.e., `S` and `E`) are added in the beginning and at the end of the trace for the execution of the sequence of statements in the transaction. The share region events are handled in a similar way (in rule ShareReg). Rule GC removes unreachable objects from the heap, and records deallocation events for them in the trace α.

Trace validation Checking and inference of unnecessary references are formulated as judgments $\omega, \gamma, \pi, \iota \vdash \alpha \rightsquigarrow \omega', \gamma', \pi', \iota'$ on traces. Here α is an execution trace, ω is a stack of transactions and share regions, γ is a "diesBefore" map in which each pair $(\hat{o}_a, \hat{o}_b)$ has been asserted to have a "diesBefore" relationship (i.e., $\hat{o}_a$ must die before $\hat{o}_b$), π is a "diesBeforeAlloc" map which contains pairs $(\hat{o}_a, o)$ where object $\hat{o}_a$ has been asserted to die before allocation site o creates a new object, and ι maps each object that has been marked shared (i.e., in inference mode) to its staleness value (measured in terms of the number of transactions). As we currently do not support nested transactions, ω can contain at most one transaction identifier object (and one $\perp$ symbol indicating the execution is in a share region). Transaction nesting can be implemented easily by allowing ω to contain multiple identifier objects.

The validation rules are given in Figure 6.5. Validity checks are underlined, and the remaining clauses (without underlines) are for environment updates. Rule VALLOC first ensures that if an object $\hat{o}_1$ has been asserted to die before this allocation site creates a new object (i.e., due to an `assertDBA` assertion), $\hat{o}_1$ is not live anymore. A violation is recorded if there is such an object. If ω is not empty (meaning the execution is currently in a transaction) and `top`(ω) is not $\perp$ (meaning we are not in a share region), a pair $(\hat{o}, \text{top}(\omega))$ is added to map γ because, as mentioned earlier, all transaction-local objects are asserted to die before the transaction identifier object, and top(ω) returns the identifier object of the current transaction.

$$\frac{\begin{array}{c}\nexists \hat{o}_1 : (\hat{o}_1, \hat{o}.\mathit{allocsite}) \in \pi \\ \gamma' = \gamma \cup (\hat{o}, \mathrm{top}(\omega)),\ \text{if } \omega \neq \emptyset \wedge \mathrm{top}(\omega) \neq \bot \qquad \gamma' = \gamma,\ \text{otherwise}\end{array}}{\omega, \gamma, \pi, \iota \vdash \langle \hat{o}, \mathrm{A}\rangle^o \rightsquigarrow \omega, \gamma', \pi, \iota} \quad (\text{VA}\textsc{lloc})$$

$$\frac{\begin{array}{c}\gamma' = \gamma \backslash \{(\hat{o}, *)\} \qquad \pi' = \pi \backslash \{(\hat{o}, *)\} \\ \nexists \hat{o}_1 : (\hat{o}_1, \hat{o}) \in \gamma \ \wedge \ \iota' = \iota \backslash \{(\hat{o}, *)\},\ \text{if } \omega = \emptyset \vee \text{MODE} = \text{CHECK} \\ \iota' = (\iota \backslash \{(\hat{o}, *)\}) \cup \{(\hat{o}_1, 0) \mid (\hat{o}_1, \hat{o}) \in \gamma\},\ \text{otherwise}\end{array}}{\omega, \gamma, \pi, \iota \vdash \langle \hat{o}, \mathrm{D}\rangle^o \rightsquigarrow \omega, \gamma', \pi', \iota'} \quad (\text{VD}\textsc{ealloc})$$

$$\frac{\begin{array}{c}\omega' = \omega \circ \hat{o} \qquad \text{MODE} = m \qquad \pi' = \pi \cup (\hat{o}, \hat{o}.\mathit{allocsite}) \\ \iota' = \iota[\forall \hat{p} : \hat{p} \mapsto \iota(\hat{p}) + 1]\end{array}}{\omega, \gamma, \pi, \iota \vdash \langle \hat{o}, m, \mathrm{S}\rangle^t \rightsquigarrow \omega', \gamma, \pi, \iota'} \quad (\text{VT}\textsc{ran}\text{S})$$

$$\frac{\omega = \omega' \circ \hat{o} \qquad \underline{\forall \hat{p} \in \mathrm{dom}(\iota) : \iota(\hat{p}) < \mathrm{T}}}{\omega, \gamma, \pi, \iota \vdash \langle \hat{o}, m, \mathrm{E}\rangle^t \rightsquigarrow \omega', \gamma, \pi, \iota} \quad (\text{VT}\textsc{ran}\text{E})$$

$$\frac{\iota' = \iota \backslash \{(\hat{o}, *)\}}{\omega, \gamma, \pi, \iota \vdash \langle \hat{o}, \mathrm{U}\rangle^o \rightsquigarrow \omega, \gamma, \pi, \iota'} \quad (\text{VU}\textsc{se})$$

$$\omega, \gamma, \pi, \iota \vdash \langle \mathrm{S}\rangle^s \rightsquigarrow \omega \circ \bot, \gamma, \pi, \iota \quad (\text{VS}\textsc{hare}\text{S})$$

$$\frac{\omega = \omega' \circ \bot}{\omega, \gamma, \pi, \iota \vdash \langle \mathrm{E}\rangle^s \rightsquigarrow \omega', \gamma, \pi, \iota} \quad (\text{VS}\textsc{hare}\text{E})$$

$$\omega, \gamma, \pi, \iota \vdash \langle \hat{o}_1, \hat{o}_2, \mathrm{DB}\rangle^a \rightsquigarrow \omega, \gamma \cup \{(\hat{o}_1, \hat{o}_2)\}, \pi, \iota \quad (\text{VDB})$$

$$\omega, \gamma, \pi, \iota \vdash \langle \hat{o}_1, \hat{o}_2, \mathrm{DBA}\rangle^a \rightsquigarrow \omega, \gamma, \pi \cup \{(\hat{o}_1, \hat{o}_2.\mathit{allocsite})\}, \iota \quad (\text{VDBA})$$

$$\frac{\omega, \gamma, \pi, \iota \vdash \tau \rightsquigarrow \omega', \gamma', \pi', \iota' \qquad \omega', \gamma', \pi', \iota' \vdash \alpha \rightsquigarrow \omega'', \gamma'', \pi'', \iota''}{\omega, \gamma, \pi, \iota \vdash \tau, \alpha \rightsquigarrow \omega'', \gamma'', \pi'', \iota''} \quad (\text{VT}\textsc{race})$$

where

ω	$\in$	*Transaction stack*:	$\mathsf{N} \rightarrow \Phi \cup \{\bot\}$
γ	$\in$	*"diesBefore" map*:	$\Phi \rightarrow \Phi$
π	$\in$	*"diesBeforeAlloc" map*:	$\Phi \rightarrow \mathsf{O}$
ι	$\in$	*Staleness map for shared objects*:	$\Phi \rightarrow \mathsf{N}$
MODE:		Mode of transaction (CHECK or INFER)	
T:		Threshold staleness value	

Figure 6.5: Checking and inferring of unnecessary references.

VDealloc removes all entries $\langle \hat{o}, * \rangle$ in the three maps upon the deallocation of $\hat{o}$. If there is no running transaction, or the running transaction is in CHECK mode, this rule checks if there exists $\langle \hat{o}_1, \hat{o} \rangle \in \gamma$. If this is the case, a violation is reported, because $\hat{o}_1$ is still live at the time $\hat{o}$ dies. If the running transaction is in INFER mode, such $\hat{o}_1$ is marked as a shared object and the tool starts to track its staleness: a pair $(\hat{o}_1, 0)$ is added to map ι.

Rule VTRANS first pushes the transaction identifier object onto stack ω. It then adds a "diesBeforeAlloc" assertion on the identifier object itself: adding a pair $(\hat{o}, \hat{o}.\mathit{allocsite})$ indicates that this current instance $\hat{o}$ is asserted to die before allocation site $\hat{o}.\mathit{allocsite}$ creates a new object. As the lifetimes of transaction identifier objects are used to specify temporal boundaries of transactions, they are not allowed to overlap. Lifetime overlapping can lead to ambiguity of transaction behaviors. For each object inferred to be a shared object (i.e., $\hat{p} \in \text{dom}(\iota)$), its staleness value is incremented. An object's staleness is defined as the number of transactions since it has been marked as shared. At the end of each transaction (rule VTRANE), this staleness value is checked against a user-defined threshold T. A violation is reported if a shared object's staleness exceeds this threshold. This rule also pops stack ω. This stack becomes empty after this rule, indicating that no transaction is currently running.

VUSE removes from ι an object marked as shared: its staleness is no longer tracked because the object is used. Rules VSHARES/VSHAREE push/pop $\perp$ onto/from stack ω. Having $\perp$ on top of ω means the execution is in a share region of a transaction. Rule VTRACE specifies the composition of two different traces.

Our system allows one to specify transactions and assertions simultaneously in a program. Transaction properties are translated into basic assertions, which are checked together with assertions specified by programmers. For example, upon the allocation of each object $\hat{o}$ in a transaction (shown in VALLOC), $\hat{o}$ and the transaction identifier object (i.e., top(ω)) are added into map γ, and this relationship is checked in exactly the same way as a normal `assertDB` assertion (shown in VDB).

6.3 Implementation

We have implemented LeakChaser in Jikes RVM 3.1.0, a high-performance Java virtual machine [70]. LeakChaser is publicly available on the Jikes RVM Research Archive.[5]

Metadata and instrumentation LeakChaser adds one word to the header of each object that tracks the allocation site information of the object. There are two dynamic compilers in Jikes that transform Java bytecode into native code. The baseline compiler compiles each method when it first executes. When a method becomes hot (i.e., executed frequently), the optimizing compiler recompiles it at increasing levels of optimizations. Our approach changes both compilers to add the necessary instrumentation. LeakChaser adds instrumentation at each allocation site in order to store an identifier for the source code location (class, method, and line number) into the allocated object's extra header word.

Garbage collection LeakChaser performs assertion checks during garbage collection runs. It uses a table structure (discussed shortly) to represent the assertions, and scans this table to perform checks at the end of each GC run.

To implement quick assertion checks, we create an assertion table, shown in Figure 6.6, to record asserted "diesBefore" and "diesBeforeAlloc" relationships during the execution. Once a pair of objects is asserted, we create an assertion table entry for each of them, and then update the LeakChaser-reserved header word in each object

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

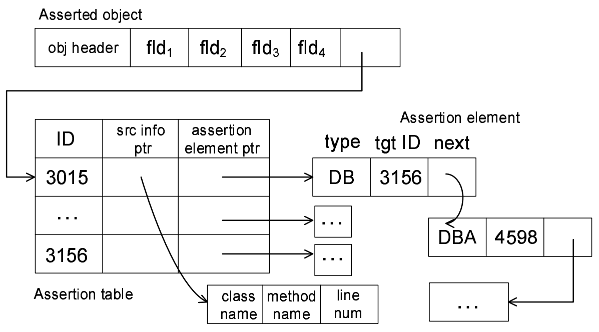


Figure 6.6: Assertion table implementation that allows quick assertion checks.

with the address of the object's corresponding assertion table entry. The pointer to the source code information of the object initially stored in this space is moved to a field of this table entry.

For each table entry, we let its "assertion element pointer" field point to a linked list of assertion elements, each of which represents an object that has been asserted together with this object. For example, for an assertion `assertDB`(a, b), we create an assertion element (`DB`, id_b, null) and store its address in a's assertion table entry. Once a new assertion `assertDBA`(a, c) is executed, a new assertion element (`DBA`, id_c, null) is created and appended to the list (i.e., now the previous element's next field points to this new element). In this way, all objects that have been asserted through `assert...`(a,...) are in an assertion chain that is going to be checked when a's entry is traversed. We do not need to create an assertion element representing a and associate it with b and c's table entries, as this unnecessarily duplicates information.

The current implementation of LeakChaser supports all non-generational garbage collectors (e.g., MarkSweep, MarkCompact, and Immix). At the end of each GC run, the assertion table is scanned twice: the first scan marks entries that correspond to objects that are unreachable in this GC (i.e., dead objects), and the second scan performs violation detection. Hence, in the second scan, all table entries represent live objects. In order to slow down the growth of the assertion table, table entries corresponding to unreachable objects are reclaimed and reassigned later to newly asserted objects. Our current implementation does not work correctly in a generational garbage collector, as a nursery GC scans only part of the heap and thus may cause LeakChaser to report either false positives or false negatives.

For each entry, its assertion element chain is traversed. For an element whose type is `DB`, if the object represented by this element has been reclaimed, we report a violation. It is much more difficult to check a `DBA` assertion. We create a global array (per thread) and each entry in this array records the status of an allocation site. Once an `assertDBA`(a, b) assertion is executed, the array entry corresponding to b's allocation site is marked as "ASSERTED". When this allocation site creates a new object, the status of its array entry is changed to "ALLOCATED". During the scanning of the assertion table, for an assertion element (`DBA`, i, *), a violation is reported if the entry of this global array corresponding to the allocation site of the object whose assertion table entry id is i is "ALLOCATED", because it creates a new object before the asserted object dies. No false information can result from assertion failures reported by the tool because the assertions directly specify the liveness of objects (instead of reachability on the object graph) and report violations only when definitive evidence is observed.

In this chapter, we focus on the real-world utility of LeakChaser, rather than its performance. Hence, this assertion table is scanned during every GC. Future work could incorporate sampling to reduce overhead (i.e., scan the assertion table less frequently). To report the reference paths that lead to a violating object, we modify the worklist-based algorithm used by the tracing collector in a way so that when a reachable object is added into the worklist, its reference path (from which it is reached) is also added (to another worklist). The length of this path can be determined by the user. We develop a technique that aggregates violations of objects that are created by the same allocation sites and whose reference paths match, so that a violation is reported only after its (aggregated) frequency exceeds a user-defined threshold value. LeakChaser filters out violations that are associated with VM objects. Future work could consider more powerful aggregation and ranking functions, such as a combination of frequency, staleness, and the amount of memory leaked. In order to make LeakChaser work for a generational GC, future work could perform the two assertion table scans separately: the first scan (that marks dead objects) would be performed at the end of every GC, while the second scan (that checks assertions and detects violations) would be performed only at the end of each full-heap GC.

6.4 Case Studies

We have evaluated LeakChaser on six real, reported memory leaks. While our ultimate goal is to evaluate with enterprise-level applications such as application servers and programs running on top of them, Jikes RVM fails to run some larger server applications (such as `trade` in the latest DaCapo benchmark set [15]).

For these six cases, we also applied Sleigh [20], a publicly available research memory leak detector for Java. Sleigh finds leaks by tracking the staleness of arbitrary objects and reporting allocation and last-use sites for stale objects.

For each case, we compared our report with the report generated by Sleigh, and found that first, for the same amount of running time, information reported by our (even tier H) approach is much more relevant than that reported by Sleigh. Sleigh requires much more time to collect information and generate relatively precise reports because it is designed for production runs and uses only one bit per object to encode information statistically, whereas LeakChaser tracks much more information for debugging and tuning (at a higher cost). We did not run Sleigh for as much time as in the prior work [20], and thus Sleigh results are different from those reported earlier. Second, our iterative technique produces more precise information at each tier, which eventually guides us to the root causes of leaks.

Experience summary We found that it is quite easy to specify transactions in large programs even for users who have never studied the programs before. From our experience using LeakChaser, we generalize an approach that can be employed by performance experts to select transactions. All large-scale applications we have studied can be classified into two major categories: event-based systems (e.g., web servers and GUI applications) and transaction-based systems (e.g., enterprise Java applications and databases). An event-based system often uses a loop to deal with different events received, dispatching them to their corresponding handlers. For such a system, our transaction can cross the body of the loop; that is, the handling of each event is treated as a transaction. Each event object can be used as the identifier object for the transaction. It is much easier to determine a transaction for a transaction-based system, as each "system transaction" in the original program can be naturally treated as a transaction in LeakChaser.

Case	*#Tran*	*#TO*	*#LO*	*#SO*	*#V*	*#F*
Diff	8	2048148	1707244	340904	36	14
Jbb	4346	256236	186752	69484	14	4
Editor	12	512471	506620	5851	2	13
WTP	20	2774556	2767237	7319	27	39
MySQL	10000	319529	170902	148627	11	0
Mckoi	100	2689366	2243888	445478	10	193

Table 6.1: Tier H statistics for transactions for the case studies. Shown are the number of transaction runs (#Tran), the total number of objects tracked (#TO), the number of transaction-local objects (#LO), the number of shared objects inferred (#SO), the number of violations reported after filtering (#V), and the number of violations filtered by our filtering system (#F).

A particularly useful feature of LeakChaser is that it allows programmers to quickly specify transactions at a *client* that interacts with a complex system, without digging into the system to understand its implementation details. For example, we diagnose Eclipse framework bugs by specifying transactions in plugins (i.e., clients) that trigger these bugs. Likewise, for databases such as `MySQL` and `Mckoi`, we only need to create transactions at client programs that perform database queries. This feature allows performance experts like us (who are unfamiliar with these databases) to quickly get started with LeakChaser. If we could not put transactions around clients, it would be hard to even find a starting point in these systems that have thousands of classes and millions of lines of code.

Table 6.1 shows transaction and assertion statistics for all six cases. Despite the large number of assertions checked for each benchmark, LeakChaser reports a small number of warnings. We show statistics for tier H only, as it is the coarsest-grained approach and thus runs the most assertions and reports the most violations.

Diff *(Eclipse bug #115789)* We quickly found this bug using our three-tier approach and the detailed diagnosis process discussed in Section 6.1.

We have tried Sleigh on this program and the top four last-use sites (i.e., where objects are last used) reported by Sleigh are in class `java.util.HashMap`, in methods `put` and `addEntry`.

In this case, the sites leading to these `HashMap` operations point back to method `createContainer` in class `org.eclipse.compare.ZipFileStructureCreator`, which indicates that the structures of input files are cached but not used. At this point, it is completely up to the programmer to find out why these structures become stale and which objects reference them. Recall that even a violation reported by our tier H approach shows the violating object is referenced by a `NavigationHistory` object, which is actually the root cause of the leak (shown in in Section 6.1). According to [20], Sleigh could have reported much more precise information if the program was run for significantly more time. For example, LeakChaser reported the root cause after only 8 structure diffs were performed, while Sleigh may need more than 1000 diffs to report relatively precise information regarding where the stale objects are created. In addition, as we observed in our experiments, reporting reference paths can be more helpful for finding leak root causes than reporting program locations, which are usually far from where the true problem occurs.

Jbb SPECjbb2000 simulates an online trading system. It contains a known memory leak that has been studied many times [1, 20, 71, 158]. This bug is caused by unnecessary caching of `Order` objects. To investigate the usefulness of LeakChaser in helping programmers unfamiliar with the code, we attempted to simulate what these programmers would do. First, we needed to understand the basic modules and functions of the program so that we can identify regularly occurring events. This is not hard at all, as SPECjbb is a transaction-based system where a `TransactionManager` class runs different transactions types, and the transaction-creating method is only a few calls away from method `main`. This method contains a loop that retrieves a command from an input map per iteration, and creates and runs a transaction whose type corresponds to the command received. This was the knowledge we obtained quickly before trying the tier H approach.

Problem 1: unused objects We added a transaction that encloses the main Jbb transaction-creating method. Each Jbb transaction object is used as the transaction identifier object of its corresponding transaction. We ran the program on our modified VM and let LeakChaser infer unnecessary references. The tool reports 14 violations (detailed statistics are shown in Table 6.1). Most of the violating objects are `String` and `History` objects cached in orders, and the rest are `Order` objects transitively referenced by `District` objects. We inspected the source code and found that these `String` objects are created to represent district information or names of order lines, and the `History` objects are used to represent histories of orders made by each company. These objects are indeed never used in the program. We modified the program to eliminate these unused strings and `History` objects. The throughput improvements are shown in Figure 6.7(a). We did not expect to find this problem, which had not been reported before.

Problem 2: mutual references It was unclear to us what to do about the `Order` objects reported above: while many are not used again, some are retrieved by subsequent transactions. We further inspected the code in order to run a more focused diagnosis. The reference paths in our tier H report showed that the unused `Order` objects are referenced by `Customer` objects and then by `Company` objects. Following this clue, we inspected all classes related to `Company` and `Customer`. One clear piece of information that we could take advantage of were the ownership relationships among `Company` and `Customer` objects and objects referenced by them. For example, each customer has an order array that stores orders made by that customer. It is clearly problematic if a customer object dies while an order made from this customer is still alive. Using this information, we wrote 72 `diesBefore` assertions to assert such relationships.

We easily added these assertions in constructors, wherever an owned object is assigned to an owner object. Running this version of the program resulted in 4 violations, all of which were related to orders. By inspecting reference paths associated with these violations, we found the following three important problems. (1) `Order` and `Customer` are mutually referenced, which explains the persistent increase in `Order` objects, even though customers frequently drop orders. (2) `Customer` objects are not released even after their container arrays in `Company` die. They are not released because `Company` holds unnecessary references to these objects. (3) `Order` and `Heap` are mutually referenced, which prevents `Heap` objects from being garbage collected. Fixing all of these problems (including the unused `String` and `History` objects) led to both the elimination of the quick growth of memory consumption and an overall 12.7% throughput improvement (from 13,644 to 15,372 ops/sec). Previous work has not reported all of these problems that we found using LeakChaser.

An important last-use site reported by Sleigh is in method `getObject` of a BTree data structure that is used to retrieve `Order` objects from customers. The associated calling contexts tell us that this call is made transitively by `processLine`. While

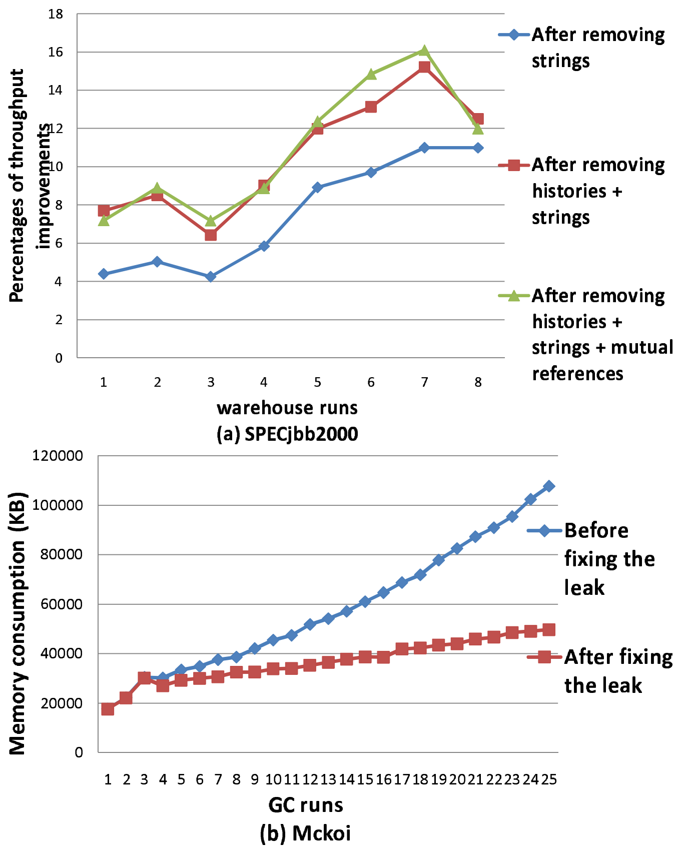


Figure 6.7: (a) Throughput improvements after fixing the memory problems detected in SPECjbb; (b) Comparison between the memory footprints of Mckoi before and after fixing the leak.

this is indeed a method that needs to be fixed, the key to understanding the problem is learning about the mutual references among `Order` and other classes. Without such reference information, it remains a daunting task to identify the root cause and develop a fix.

Editor *(Eclipse bug #139465)* This reported memory leak in Eclipse 3.2 occurs when opening a `.outline` file using a customized editor and making selections on the "outline" page. Objects created while displaying the file keep accumulating, and Eclipse eventually crashes. This bug is still open because users can rarely reproduce the leak. While we could not observe the reported symptoms, we still ran the test case with LeakChaser. We wrote a plugin that repeats multiple times the process of opening the editor, making selections, and closing the editor. We added a transaction in this plugin that crosses each such process (which manipulates the editor), and used the editor object as the transaction identifier.

The tier H approach reported only two violations even when we set a very small number (2) as the staleness threshold (see T in Figure 6.5). For each violation, we could not find any object on its associated reference paths that is related to the editor of interest. Hence, we quickly concluded that there were no unnecessary references for this case in Eclipse 3.2 on the Linux platform where we ran the experiment: had there existed a problem with respect to a transaction, it should have been reported by the tier H approach, as the approach captures references to all objects that are shared but not used. For this case, Sleigh reported four last-use sites, and we found it difficult to verify whether or not these sites are relevant. The key is to understand how these objects are reachable in the heap, instead of where they are used in the program.

WTP *(Eclipse bug #155898)* According to the bug report for this memory leak in the Eclipse Web Tool Platform (WTP), the memory footprint grows quickly when copying and pasting large text blocks in the JSP editor associated with the WTP framework. This bug is still open because it cannot always be reproduced. One developer suspected that "the bug might have already been fixed by the addition of other features in a later version of WTP" one year after it was reported. We reproduced the problem and saw the memory footprint growth by writing a plugin that automatically copies and pastes text several times. Similarly to the previous case, we added a transaction in the plugin and let it cross each iteration that involves a pair of copy-and-paste operations. Our tier H approach reported a total of 26 violations, among which 11 seemed clearly irrelevant to the problem (e.g., regarding Eclipse's Java Development Tools and other plugins).

To confirm this observation, we used this plugin to perform the same copy-and-paste operations in a regular text editor in Eclipse, which does not have memory problems. These 11 violations also appeared in the generated report. Thus, they were safely discarded. As this bug is caused by copying and pasting text, we focused on string-related violations. Only one violation among the remaining 15 was about `String` objects, shown as follows:

```
Transaction specified at:
  Quick_copy_pastePlugin:mouseUp(MouseEvent), ln 258
Violating objects created at:
  StructuredTextUndoManager:createNewTextCommand(String, String), ln 302
Violation type:
  objects shared among transactions are not used
Frequency: 28
Reference paths:
    Type: StructuredTextViewerUndoManager, created at:
               StructuredTextViewerConfiguration
               : getUndoManager(ISourceViewer), ln 469
--> Type: StructuredTextUndoManager, created at:
     BasicStructuredDocument: getUndoManager(), ln 1823
--> Type: BasicCommandStack, created at:
        StructuredTextUndoManager : <init>(), ln 160
--> Type: ArrayList, created at: BasicCommandStack: <init>(), ln 67
--> Type: Object[], created at: ArrayList: ensureCapacity(I), ln 176
```

This violation clearly shows that the strings are (transitively) referenced by a text undo manager. This information quickly directed us to classes responsible for undo operations. The cause of the problem was clear to us almost immediately after inspecting how an undo operation is performed. In this JSP editor, all commands (and their related data) are cached in a command stack, in case an undo is requested in the future. There is a tradeoff between "undoability" (i.e., how many commands are cached) and performance, especially when there is a large amount of data being cached with each command. In this version of WTP (WTP 1.5 with Eclipse 3.2.0), this command stack can grow to be very deep (it is not cleared even when a save operation is performed), and thus, many strings can be cached, leading to significant performance degradation. A later version has limited the depth of this stack (for other purposes), implicitly fixing this bug. For this case, we understood the problem even without moving to tier M. We did not manage to run Sleigh for this case, as Sleigh was developed on top of an older version of Jikes RVM (2.4.2), which cannot run Eclipse 3.2.

MySQL This case is a simplified version of a JDBC application that exhausts memory if the application keeps using the same connection but different SQL statements to access the database. Prior work reproduced this leak to evaluate tolerating leaks [23], but no prior work reports the leak cause. The leak occurs because the JDBC library caches already-executed SQL statements in a container unless the statements are explicitly closed. We created a transaction that crosses the creation and execution of each `PreparedStatement` (in a loop). We executed 100 iterations, and the tier H approach reported 10 violations, all of which were related to `PreparedStatement`, regarding either the statement object itself, or objects reachable from the statement.

From the associated reference paths, it took only a few minutes for us to identify the container that caches the statements: a `HashMap` created at line 1486 (in the constructor) of class `com.mysql.jdbc.Connection`. For this case, Sleigh reported warnings after the program ran for 303 iterations, significantly longer than for LeakChaser. Sleigh reported a few last-use sites where `PrepareStatement` objects are last touched. We did not find the `HashMap` information in Sleigh's report, which is the key to tracking down this leak bug.

Mckoi Mckoi (`http://www.mckoi.com`) is an open-source SQL database system written in Java. It contains a leak that previous work reproduced to evaluate leak survival techniques [22, 23]. However, none of the existing work has investigated the root cause of the leak. We reproduced the leak by writing a client that repeatedly (100 times) establishes a connection, executes a few SQL queries, and closes the connection. We started with our tier H approach and created a transaction that crosses each iteration of this process in the client. Our tool reported 10 warnings, all regarding objects that are reachable from a thread object of type `DatabaseDispatcher`. In all the warnings reported, this thread object references a `DatabaseSystem` object, which transitively caches many other never-used objects. By inspecting only the constructors of `DatabaseSystem` and `DatabaseDispatcher`, we found that they are mutually referenced. The creation of each `DatabaseSystem` object explicitly creates a `DatabaseDispatcher` object and runs this thread in the background. There are two major problems:

(1) `DatabaseDispatcher` runs in a `while(true)` loop, which means that no `DatabaseSystem` object can ever be garbage collected even though its `dispose` method is invoked. To address this problem, we broke this reference cycle when `dispose` is invoked on `DatabaseSystem`. Next, in `DatabaseDispatcher`, we modified the `while(true)` loop to terminate if its referenced `DatabaseSystem` object becomes null. This modification resulted in a very slight improvement in performance, leading us to believe there must be a bigger problem.

(2) A `DatabaseSystem` object can be created in two situations. (a) A new database needs to be started (e.g., a connection is established); (b) a method `dbExists` (that

checks whether a database instance has already been there) is called. In every iteration, two `DatabaseSystem` objects are created but only one of them gets disposed. The one created by `dbExists` is never reclaimed, because `dispose` is not explicitly invoked on this object, which we quickly discovered by using a call graph generated by an IDE tool. The developer took it for granted that this object would be garbage collected, but it is referenced by a live thread. We added a call that invokes `dispose` at the end of method `dbExists`.

For this case, all warnings generated by Sleigh are about objects cached (transitively) by `DatabaseSystem`, and they are far away from the leaking thread. It would be difficult to understand why these objects are not garbage collected without appropriate reference paths, as reported by LeakChaser. The memory footprint of the database before and after the leak is fixed is shown in Figure 6.7(b). The original version of the program ran out of memory at the 106^{th} iteration, while the modified version ran indefinitely (as far as we could tell).

These six case studies demonstrate that developers do not need to have much implementation knowledge in order to diagnose leaks with LeakChaser. LeakChaser generates more relevant reports than Sleigh: it is easier to find the root cause from reference chains that cache a violating object, than from the allocation or last-use site of the object.

6.5 Overhead

We evaluated the performance of our technique using 19 programs from the SPECjvm98 [139] and DaCapo [15] benchmarks, on a dual-core machine with an

Bench	(a) Overall time			(b) GC time			(c) Space		
	T_1	T_2	OT	G_1	G_2	OG	S_1	S_2	OS
check	0.033	0.068	2.1	0.016	0.033	2.0	11.4	10.3	0.9
compr	9.4	9.5	1.0	0.016	0.033	2.0	17.9	18.0	1.0
jess	2.2	2.4	1.1	0.022	0.049	2.2	48.9	58.4	1.2
db	4.9	5.0	1.0	0.024	0.062	2.6	22.2	24.9	1.1
javac	2.1	2.4	1.1	0.030	0.064	2.1	98.1	99.0	1.0
mpeg	6.2	6.2	1.0	0.020	0.039	2.0	19.4	19.8	1.0
mtrt	1.6	1.6	1.0	0.031	0.081	2.6	46.5	43.6	0.9
jack	1.7	1.8	1.0	0.025	0.050	2.0	79.6	76.8	1.0
antlr	38.7	43.2	1.1	0.032	0.074	2.3	53.5	57.3	1.1
bloat	98.3	105.9	1.1	0.073	0.169	2.3	515.1	520.0	1.0
chart	19.7	21.4	1.1	0.084	0.232	2.8	367.4	479.4	1.3
eclipse	150.8	157.5	1.0	0.207	0.465	2.2	512.4	532.7	1.0
fop	1.4	1.8	1.3	0.064	0.15	2.3	107.6	102.2	0.9
hsqldb	9.3	16.4	1.8	0.331	1.21	3.7	420.5	432.6	1.0
jython	45.6	48.6	1.1	0.077	0.172	2.2	119.7	137.8	1.2
luindex	35.3	37.7	1.1	0.031	0.070	2.3	45.5	49.8	1.1
lusearch	7.8	8.1	1.0	0.038	0.088	2.3	105.0	129.5	1.2
pmd	22.9	23.6	1.0	0.055	0.127	2.3	115.3	157.5	1.4
xalan	32.3	39.5	1.2	0.066	0.145	2.2	214.2	209.0	1.0
GeoMean	-	-	1.1	-	-	2.3	-	-	1.1

Table 6.2: Time and space overheads incurred by our infrastructure for the FastAdaptiveImmix configuration. Shown in column (a) are the original overall execution times (T_1) in seconds, the execution times for our modified JVM (T_2), and the overheads (OT) in times ($\times$). Column (b) reports the GC times and overheads: G_1 and G_2 show the average times for each GC run in the original RVM and our modified RVM, respectively. OG reports the GC overheads as G_2/G_1. Memory consumption and overheads are shown in column (c): S_1 and S_2 are the maximum amounts of memory used during the executions in the original RVM and our RVM, respectively. OS reports the space overheads as S_1/S_2.

Intel Xeon 2.83GHz processor, running Linux 2.6.18. We ran each program using the large workload.

Table 6.2 reports the time and space overheads that our tracking infrastructure incurs on normal executions (without assertions or transactions written). The overhead measurements that we show in this section are obtained based on a (high-performance)

FastAdaptiveImmix configuration that uses the optimizing compiler and the state-of-the-art Immix garbage collection algorithm [16].

The LeakChaser infrastructure slows programs by less than 10% on average using the optimizing compiler and an advanced garbage collector. Much of the overhead comes from extra operations during GC and the barrier inserted at each object reference read to check the use of the object. Column (b) of Table 6.2 reports the detailed GC slowdown caused by our tool, which is 2.3×, averaged across the 19 programs. The space overheads, reported in column (c) of Table 6.2, are less than 10%, primarily due to the extra word added to the header of each object. In some cases, the peak memory consumption for LeakChaser is even lower than that for the original run (i.e., *OS* is smaller than 1), presumably because GC is triggered at a different set of program points that happens to have a lower maximum reachable memory size.

The overhead of assertion checking and inference is reported in Table 6.3. The overall running time measurements are available only for *Jbb*, *MySQL*, and *Mckoi*, as the other three cases are all based on Eclipse IDE operations. Note that in a run with 4,346 transactions and 442,988 objects (shown in Table 6.1), our tool slows the program 6.6× and 5.5× overall for the two configurations. Similarly to Table 6.2, the overhead can be larger if we consider only GC time. For example, for *Diff*, a 2.8× slowdown can be seen for GC time for FastAdaptiveImmix. Note that these overheads have not prevented us from collecting data from any real-world application. In a production setting, run-time overhead can be effectively reduced by sampling (i.e., the current sampling rate is 100%). Future work could also define a tradeoff framework between the quality of the reported information and the frequency of assertion table

Bench	(a) Overall time			(b) GC time			(c) Space		
	T_1	T_2	OT	G_1	G_2	OG	S_1	S_2	OS
Diff	-	-	-	0.27	0.75	2.8	534.2	534.4	1.0
Jbb	50.7*	9.2*	5.5	0.008	0.056	7.0	313.7	315.3	1.0
Editor	-	-	-	0.15	0.41	2.7	300.5	367.2	1.2
WTP	-	-	-	0.22	0.58	2.6	52.7	52.9	1.0
MySQL	6.8	14.0	2.1	0.045	0.108	2.4	17.1	17.7	1.0
Mckoi	165.6	172.2	1.0	0.046	0.154	3.3	129.2	144.2	1.1
GeoMean	-	-	2.3	-	-	3.2	-	-	1.0

Table 6.3: Overheads for the cases that we have studied under FastAdaptiveImmix. * indicates that we measure throughput (#operations per second) instead of running time. For both MySQL and Mckoi, a fixed number of iterations (100) was run for this measurement.

scanning (similar to QVM [9]), and find an appropriate sampling rate that can enable the reporting of sufficient information at acceptably low cost.

6.6 Summary and Interpretation

This chapter presents another program-semantics-aware memory leak detector that allows programmers with little program knowledge to quickly find the root cause of a memory leak. The most significant advantages of this work over existing Java leak detection tools are that (1) the approach uses lightweight user annotations to improve the relevance of the generated reports, and can provide semantics-related diagnostic information (e.g., which object escapes which transaction), and (2) it is designed in a multi-tier way so that programmers at different levels of skill and code familiarity can use it to identify performance problems. Our experience shows that the three-tier technique is quite effective: the tool can help a programmer identify the root cause of a leak after an iterative process of specifying, inferring, and checking object lifetime properties. For the case studies we conducted, the tool provided more relevant information than an existing memory leak detector Sleigh. For example, using the tool, we quickly found the root causes of memory issues in the analyzed programs, including both known memory leaks and problems that have not been reported before. While LeakChaser incurs relatively large overhead (2.3× on average), we found it acceptable for debugging and performance tuning.

We envision that the assertions and transaction constructs proposed in this chapter may be incorporated into the Java language so that they can be used in a production JVM to help detect memory problems. Similarly to functional specifications that are used widely in program testing and debugging, writing performance specifications can have significant advantages. For example, it could enable *unit performance testing* that can identify performance violations even before degradation is observed. In addition, performance specifications (similar to the ones described in this chapter) may help bridge the gap between performance analysis and the large body of work on model checking and verification, so that one may be able to prove (statically) that a program is "bloat-free" or "bloat-bounded" with respect to the performance specifications provided. One may even be able to use a compiler to automatically synthesize a bloat-free implementation that satisfies the given performance specifications. For future work, it is interesting to investigate what other performance specifications can be useful in exposing run-time performance problems.

CHAPTER 7: Statically Detecting Inefficiently-Used Containers to Avoid Bloat

Dynamic analysis has typically been the technique of choice in detecting performance problems, as finding these problems requires a fair amount of run-time information, such as method execution frequency and object allocation counts, which cannot be obtained at compile time. However, during the studies of our dynamic analysis reports, we found a number of patterns of inefficiencies (i.e., bloat patterns) that occur frequently in the execution of large-scale applications. These patterns often point to problems inherent in the source code of a program, and thus, can be easily recognized at compile time. While the execution of these patterns may or may not exhibit noticeable performance degradation, it is certain that they are harmful and may trigger significant performance issues when the application workload increases or the layers of the application grow to be deep.

In the next two chapters, we present static analysis techniques targeting two specific bloat patterns, (1) inefficient use of containers and (2) loop-invariant data structures. These patterns are among the most-frequently occurring ones that we observed in our studies. Our static analyses are able to pinpoint instances of such patterns in the source code, even when the program is incomplete. Using these static analyses, developers can find and avoid performance problems early during their development. These analyses can also be incorporated into an optimizing compiler so that the compiler may be able to either transform a program automatically to avoid such patterns, or generate performance warnings during compilation.

Inefficient use of containers The inefficient use of containers is an important source of systemic bloat. Programming languages such as Java include a collection framework which provides abstract data types for representing groups of related data objects (e.g., lists, sets, and maps). Based on this collection framework, one can easily construct application-specific container types such as trees and graphs. Java programs make extensive use of containers, both through collection classes and through user-defined container types. Programmers allocate containers in thousands of code locations, using them in a variety of ways, such as storing data, implementing unsupported language features such as returning multiple values, wrapping data in APIs to provide general service for multiple clients, etc.

Containers are easy to misuse. In Chapter 5, we have seen memory leak problems caused by careless use of containers. Another problematic situation is when a container, while not strictly necessary for what it is supposed to accomplish, is used nevertheless. Although most such containers are eventually garbage-collected and allocating them may not lead to `OutOfMemory` errors, they can have conspicuous impact on performance. For example, as illustrated in Chapter 3, finding and specializing an inefficiently-used HashMap dramatically reduced the number of allocated objects in a commercial server application. Identifying containers that consume excessive resources (for what they accomplish) is an important step toward finding potential container-related optimization opportunities.

Motivation and problems The extensive use of containers in Java software makes it impossible to manually inspect the choice and the use of each container in

```
class CUP$LexParse$actions {
    RegExp makeNL(){
        Vector list = new Vector();
        list.addElement(new Interval('\n','\r'));
        list.addElement(new Interval('\u0085','\u0085'));
        list.addElement(new Interval('\u2028','\u2029'));
        RegExp1 c = new RegExp1(sym.CCLASS, list);
        ...
    }
}
                        (a)
```

```
class Httpd extends HttpConnection{
  Reply recvReply(Request request){
    if (request.getPath().equals("/admin/enable")){
      Hashtable attrs = cgi(request);
      String config = (String) attrs.get("config");
      String filter = (String) attrs.get("filter");
      ...
    }else if(request.getPath().equals("/admin/createConfig")){
      Hashtable attrs = cgi(request);
      String config = (String) attrs.get("config");
      ...
    } ...
  }
  Hashtable cgi(Request request){
    Hashtable attrs = new Hashtable(13);
    String query = request.getQueryString();
    String data = request.getData();
    if (query != null){
      StringTokenizer st = new StringTokenizer(decode(query), "&");
      while (st.hasMoreTokens()){
         String token = st.nextToken();
         String key = token.substring(0, token.indexOf('='));
         String value = token.substring(token.indexOf('=') + 1);
         attrs.put(key, value);
      }
    } ...
    return attrs;
  }
}
                         (b)
```

Figure 7.1: (a) An example of an *underutilized container*, extracted from program *JFlex*, a scanner generator for Java. `Vector` *list* is used only to pass the three `Interval` objects to the constructor of `RegExpr1`; (b) An example of an *overpopulated container*, extracted from program *Muffin*, a WWW filtering system written in Java. Given a request, method `cgi` always decodes the entire request string into `Hashtable` *attrs*. However, this `HashMap` is later subjected to at most two lookup operations.

the program. A dynamic bloat detector finds performance problems using a "from-symptom-to-cause" approach: it starts by observing suspicious behaviors and then attempts to locate the cause from the observed symptom. One fundamental problem for such approaches is the selection of appropriate symptoms. It can be extremely difficult to define symptoms that can precisely capture the target bloat—in many cases, the suspicious behaviors that the tool intends to capture are not unique characteristics of the problematic program entities. Very often, they can also be observed from entities that function appropriately, leading to false positives and imprecise reports. The first research problem investigated in this chapter is *whether it is possible to alleviate this limitation of purely dynamic bloat analyses by using a static or a hybrid technique.* A static (or a hybrid static/dynamic) technique may be able to reduce false positives by exploiting certain program properties inherent in the source code.

The second major problem addressed in the chapter is related to understanding the semantics of containers. Of existing dynamic bloat detection techniques, tools from [158] and [122] were designed specifically for finding container-related problems. Both tools require the user to provide annotations for each container type, which are subsequently used to understand the container semantics and relate container behavior to the profiling frameworks. However, it may be a heavy burden for the programmer to complete these annotations. In addition, when the interface of a container changes, its annotations have to be revised as well. Creating such annotations is also impractical when the container types come from large libraries and frameworks developed by others (e.g., containers of transaction data in enterprise applications). To reduce such an annotation burden and make the container analysis more general, this chapter explores the possibility of *extracting the container semantics automatically from the container implementation, with minimal need for user interaction.*

Targeted container inefficiencies We aim to detect containers that are inefficiently used in two different ways.[6]

Underutilized container. A container is *underutilized* if it holds a very small number of elements during its lifetime. It is wasteful to use an underutilized container to hold data. First, a container is usually created with a default number of slots (e.g., 16), and a big portion of the memory space is wasted if only a few slots are occupied. If the size of the container is fixed (e.g., 1), a specialized container type such as `Collections.singletonSet` can be employed to replace the original general type (e.g., `HashSet`). Second, the functionality associated with the container type may be much more general than what is actually needed. For example, the process of retrieving an element from a `HashSet` involves dozens of calls. If there is a small number of objects in the `HashSet`, it may be possible for a performance expert to replace this `HashSet` by introducing extra local variables, parameters, or an array.

Figure 7.1(a) illustrates an example of this type of problem, which was reported by our tool. The example is extracted from *JFlex*, a scanner generator from our benchmark set. Method `makeNL` creates a `Vector` object that by default allocates a 10-element array. The only purpose of this object is to pass the three `Interval` objects into the constructor of `RegExpr1`. This `Vector` object is allocated more than 10,000 times during the execution of a large test case. In fact, there are many locations in the code where `Vector` objects are created solely for this purpose. Creating specialized

[6]We thank Gary Sevitsky for suggesting the terms "underutilized" and "overpopulated" to describe these patterns of inefficient use.

constructors of `RegExpr1` that allow the direct passing of `Interval` objects can avoid the allocation/deallocation of thousands of objects.

Overpopulated container. This problem occurs if there is a container that, while holding many objects, is looked up only a few times. Due to unnecessary data elements, memory is wasted and it takes longer to perform a container operation. In this case, a programmer may be able to inspect the code to find which objects will definitely *not* be retrieved from the container, and then find a way to avoid adding these objects or even creating them (if they are never used). Figure 7.1(b) shows an example of this problem, which was found by our analysis in *Muffin*, a WWW filtering system from our benchmark set. Many strings are generated and added into a `Hashtable`, but only the entries with keys `"config"` and `"filter"` are eventually requested. Instead of decoding and bookkeeping the entire request string, a specialized version of method `cgi` could declare an additional string parameter representing the requested key, and return the corresponding value immediately when the given key is found.

Base static analysis: extracting container semantics There are three major technical challenges in automatically extracting semantics for different container types and implementations. The first challenge is to establish a unified model for different container types. For example, consider two concrete container classes in Java, `Hashtable` and `LinkedList`. The unified model has to capture the common behaviors that characterize their "container" property while ignoring the differences in their specific implementations and usage domains. To address this challenge, we propose to treat all container classes as an abstract data type with two basic operations: *ADD* and *GET*. Consider `Hashtable` and `LinkedList` again: despite their many differences, we are interested only in the process by which objects are added to (*ADD*) and retrieved from (*GET*) these containers. Details of their implementations and usage (e.g., whether they store key-value pairs or individual objects) are abstracted away.

The second challenge is to select program entities that correspond to these abstract operations. Research from [122, 158] focuses on methods. For example, methods `put` and `get` implement the semantics of *ADD* and *GET*, respectively, for class `Hashtable`. While identifying abstract operations at the method level is a straightforward idea, it is impossible to perform without user annotations because different container types use different methods for these operations. To automate this process, we propose to operate at the statement level. The key insight is that the core behavior of each operation can be implemented by a single statement. The statements that implement the *ADD* and *GET* operations are usually heap stores and loads, respectively. Such statements will be referred to *semantics-achieving* statements. For example, for class `ArrayList`, the statement achieving the functionality of *ADD* is a heap store `array[i] = o` in method `ArrayList.add`, where `array` refers to the backing array of the list and `o` is a formal parameter referring to the object[7] to be added. Identifying semantics-achieving statements bridges the gap between the low-level code analysis and the high-level container semantics.

The third challenge is to develop precise and efficient algorithms to discover container structures. The identification of semantics-achieving statements for a particular container object requires reasoning about the container data structure, in order to detect the objects that are added to the container from the client code (i.e., *element*

[7]For the rest of this chapter, we will use "object" to denote the static abstraction (i.e., allocation site) of a set of run-time objects created by the allocation site.

objects) as well as the helper objects that are created by the container (i.e., *inside objects*). There are usually multiple layers in the data structure of a container type, and a naive approach based on points-to analysis may not be able to distinguish among elements added to different container objects that have the same type. To obtain precise information about the use of a container, it is crucial to prune, context-sensitively, nodes and edges irrelevant to the container in the points-to graph. While there exists a body of precise reasoning techniques such as shape analysis (e.g., [29, 33, 121]) and decision procedures for verification of pointer-based data structures (e.g., [81, 89]), these analyses tend to be expensive and do not scale well to large applications.

Our analysis attempts to refine the object sub-graph rooted at each container object by taking advantage of the CFL-reachability formulation of pointer aliasing. The key observation is that if an object o can be reached from a container object c through (direct or transitive) field dereferences, there must exist a chain of stores of the form $a_0.f_0 = o; a_1.f_1 = b_0; a_2.f_2 = b_1; \ldots ; a_n.f_n = b_{n-1}; b_n = c$, such that the two reference variables in each pair (a_i, b_i) for $0 \leq i \leq n$ are aliases. Because aliasing relationships can be computed by solving CFL-reachability on a flow graph [133], the goal of our analysis is to find valid paths (in terms of both heap accesses and method calls) on the flow graph that contain such chains of stores. We consider all objects o that have such paths reaching the container object and that are not inside objects created by the container. Among those, element objects are the ones that have a chain of stores $a_0.f_0 = o; a_1.f_1 = b_0; \ldots$ such that all a_i and b_i along the chain point to inside objects of the container. We have successfully applied this demand-driven analysis to large Java applications, including the `eclipse` framework (and its plugins). The description of the analysis can be found in Section 7.1.

Static inference and dynamic profiling of execution frequencies If the *ADD* operations of a container are executed a very small number of times, the container may suffer from an underutilization problem. If the frequency of its *GET* operations is much smaller than the frequency of its *ADD* operations, the container may be overpopulated. The next step of the analysis is to compare the frequencies of these operations, using the semantics-achieving statements (annotated with the relevant calling contexts) identified by the base analysis.

A natural choice for comparing the frequencies of the semantics-achieving statements is to instrument these statements and to develop a dynamic analysis by profiling the *observed* frequencies. However, the usefulness of this approach may be limited because it does not directly point to the underlying cause of the problem. Furthermore, the generated results depend completely on the specific inputs and runs being observed: containers whose behaviors are suspicious in one particular run may behave appropriately in other runs, making it hard to identify problematic containers.

An alternative is to design a static analysis that detects performance problems by looking for certain source code properties that can *approximate* the relationship between execution frequencies, regardless of inputs and runs. There exist a number of analyses that can be employed to infer such a relationship. For example, semantics-achieving statements are often nested in loops. Various techniques such as interval analysis [140] and symbolic bound analysis [55, 56] may be used to discover the loop bounds. However, such techniques are often ineffective in handling dynamic heap data structures, and it is difficult to scale them to large programs.

We take a much simpler approach where data flow does not need to be considered: relative frequencies are inferred based on the nesting of the loops where the semantics-achieving statements are located. Despite this simplicity, the inferred relationships are execution independent and, in our experience, lead to low false positive rates when used to find optimizable containers.

We have implemented both the dynamic frequency profiler (Section 7.2.1) and the static inference analysis (Section 7.2.2). Detailed comparison between them is provided in Section 7.2.2 and demonstrated experimentally in Section 7.3.

Features of the base analysis The base analysis needs to be sufficiently precise, as both the static inference algorithm and the dynamic frequency profiler rely on it to find problematic containers. Our algorithmic design is focused on three important features of the analysis. First, since we are interested only in containers, the algorithm is *demand-driven*, so that it can perform only the work necessary to answer queries about the usage of containers. Second, because a container type can be instantiated many times in the program, failure to distinguish elements added into different container objects of the same type could result in a large number of false positives. To avoid this, if the analysis cannot find a highly-precise solution under a client-defined time budget, it does *not* report any ADD and GET operations (instead of reporting them based on over-conservative approximations). In a practical tool that identifies potential bloat, the precise identification of inefficiently-used containers (i.e., reducing the false positive rate) is much more important than reporting all potentially problematic ones with many false warnings (i.e., reducing the false negative rate). This choice aims at higher programmer productivity and real-world usefulness. Finally, the analysis is *client-driven*, as the amount of information it produces can be controlled by the client-defined time budget.

Evaluation Section 7.3 presents experimental results showing that the static tool successfully finds inefficient uses of containers. It generates a total of 295 warnings for the 21 Java programs in our benchmark set. For each benchmark, we randomly picked 20 warnings for manual inspection. Among those, we found a very small number of false positives (e.g., 4 for the largest benchmark `eclipse`). Further experiments showed that (1) most of the statically reported containers indeed exhibit problematic behaviors at run time, and (2) the inefficient uses of these containers are much easier to understand than the uses of containers reported by the dynamic analysis.

The static inference approach is useful for detecting container problems during coding, before performance tuning has started. It is a good programming practice to fix (static) performance warnings early, in order to avoid potential performance problems before they grow and become observable. It has already been recognized [94] that bloat can easily accumulate when insufficient attention is paid to performance during development. Once coding is complete and performance tuning starts, information about run-time frequency of container allocation can focus the programmer's attention on statically-identified containers that are most likely to provide optimization payoffs.[8]

Using this approach, we studied the warnings for the DaCapo `bloat` and `chart` benchmarks, and easily identified fixes that reduced object creation rates by 30% for `bloat` and 5% for `chart`, leading to execution time reduction of 24.5% for `bloat`

[8]Allocation frequencies can even be collected *before* the static analysis, allowing the demand-driven static algorithm to focus on hot containers.

```
class ContainerClient{
 static void main(String[] args){
   ContainerClient client = new ContainerClient();
   Container c = new Container();
   for(int i = 0; i < 1000; i++){
     client.addElement(c, new Integer(i));
   }
   client.foo(c);
   client.bar();
 }
 void foo(Container n){
   Integer i = (Integer)n.get(10);
   Container d = new Container();
   addElement(d, new String("first"));
   addElement(d, new String("second"));
   String s = (String)d.get(0);
 }
 void bar(){
   for(int j = 0; j < 5; j++){
     Container a = new Container();
     for(int i = 0; i < 10; i++)
        addElement(a, new Double(i));
     for(int i = 0; i < a.size(); i++){
        Double b = (Double)a.get(i);
         ...
     }
    }
 }
 void addElement(Container c, Object e){
   c.add(e);
 }
}
```

```

class Container{
 Object[] arr;
 int pos = 0;
 Container(){
   t = new Object[1000]; this.arr = t;
 }
 void add(Object e){
   t = this.arr; t[pos++] = e;
 }
 Object get(int index){
   t = this.arr; ret = t[index]; return ret;
 }
}
```

Figure 7.2: Running example.

and 3.5% for `chart`. These promising initial findings suggest that our tools could be useful in practice to find and exploit opportunities for performance gains.

7.1 Formulation of Container Operations

This section starts with an outline of the CFL-reachability formulation of context-sensitive points-to/alias analysis for Java [133]. We formulate the base analysis for identification of semantics-achieving statements as a new CFL-reachability problem, and then present algorithms to solve this problem.

7.1.1 CFL-Reachability Formulation of Points-to Analysis

A variety of program analyses can be stated as CFL-reachability problems [115]. CFL-reachability is an extension of standard graph reachability that allows for filtering of uninteresting paths. Given a directed graph with labeled edges, a relation R over graph nodes can be formulated as a CFL-reachability problem by defining a context-free grammar such that a pair of nodes $(n, n') \in R$ if and only if there exists a path from n to n' for which the sequence of edge labels along the path is a word in the language L defined by the grammar. Such a path will be referred to as an L path. If there exists an L path from n to n', then n' is L-reachable from n (denoted by $n\ L\ n'$). For any non-terminal S in L's grammar, S paths and $n\ S\ n'$ are defined similarly.

Existing work on points-to analysis for Java [133, 135] employs this formulation to model (1) context sensitivity via method entries and exits, and (2) heap accesses via object field reads and writes. A demand-driven analysis is formulated as a single-source L-reachability problem which determines all nodes n' such that $n\ L\ n'$ for a given source node n. The analysis can be expressed by CFL-reachability for language $L_{\mathsf{F}} \cap R_{\mathsf{C}}$. Language L_{F}, where F stands for "flows-to", ensures precise handling of field accesses. Regular language R_{C} ensures a degree of calling context sensitivity. Both languages encode balanced-parentheses properties.

L_{F}-reachability is performed on a graph representation G of a Java program (sometimes referred to as a *flow graph*), such that if a heap object represented by the abstract location o can flow to variable v during the execution of the program, there exists an L_{F} path in G from o to v. The flow graph is constructed by creating edges for the following canonical statements: an edge $o \xrightarrow{\mathsf{new}} x$ is created for an allocation x

$=$ new O; an edge $y \xrightarrow{\mathsf{assign}} x$ is created for an assignment $x = y$; edges $y \xrightarrow{\mathsf{store(f)}} x$ and $y \xrightarrow{\mathsf{load(f)}} x$ are created for a field write $x.f = y$ and a field read $x = y.f$, respectively. Parameter passing is represented as assignments from actuals to formals; method return values are treated similarly. Writes and reads of array elements are handled by collapsing all elements into an artificial field *arr_elm*.

Language L_{F} Consider a simplified flow graph G with only new and assign edges. In this case the language is regular and its grammar can be written simply as $\mathit{flowsTo} \rightarrow \mathsf{new}\,(\,\mathsf{assign}\,)^*$, which shows the transitive flow due to `assign` edges. Clearly, o *flowsTo* v in G means that o belongs to the points-to set of v.

For field accesses, inverse edges are introduced to allow a CFL-reachability formulation. For each graph edge $x \rightarrow y$ labeled with t, an edge $y \rightarrow x$ labeled with $\bar{t}$ is introduced. For any path p, an inverse path $\bar{p}$ can be constructed by reversing the order of edges in p and replacing each edge with its inverse. In the grammar this is captured by a new non-terminal $\overline{\mathit{flowsTo}}$ used to represent the inverse paths for *flowsTo* paths. For example, if there exists a *flowsTo* path from object o to variable v, there also exists a $\overline{\mathit{flowsTo}}$ path from v to o.

May-alias relationships can be modeled by defining a non-terminal *alias* such that $\mathit{alias} \rightarrow \overline{\mathit{flowsTo}}\,\mathit{flowsTo}$. Two variables a and b may alias if there exists an object o such that o can flow to both a and b. The field-sensitive points-to relationships can be modeled by $\mathit{flowsTo} \rightarrow \mathsf{new}\,(\,\mathsf{assign}\,|\,\mathsf{store(f)}\ \mathit{alias}\ \mathsf{load(f)}\,)^*$. This production checks for balanced pairs of store(f) and load(f) operations, taking into account the potential aliasing between the variables through which the store and the load occur.

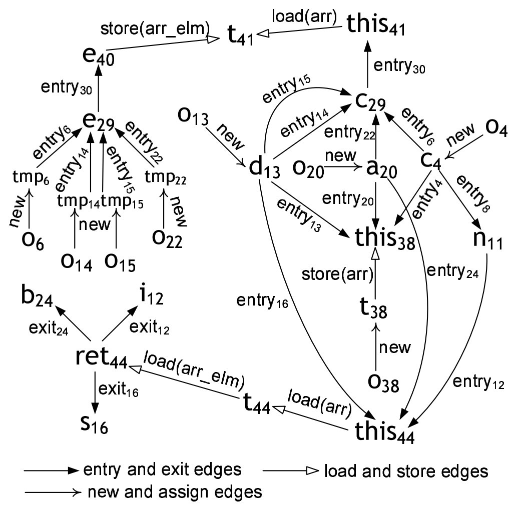


Figure 7.3: Flow graph for the running example.

Language R_{C} The context sensitivity of the analysis ensures that method entries and exits are balanced parentheses: $C \rightarrow \mathsf{entry(i)}\ C\ \mathsf{exit(i)} \,|\, C\ C \,|\, \epsilon$. Here $\mathsf{entry(i)}$ and $\mathsf{exit(i)}$ correspond to the i-th call site in the program. This production describes only a subset of the language, where all parentheses are fully balanced. Since a realizable path does not need to start and end in the same method, the full definition of R_{C} also allows a prefix with unbalanced closed parentheses and a suffix with unbalanced open parentheses [133]. In the absence of recursion, the balanced-parentheses language is a finite regular language (thus the notation R_{C} instead of L_{C}); approximations are introduced as necessary to handle recursive calls. Context sensitivity is achieved by considering entries and exits along a L_{F} path and ensuring that the resulting string is in R_{C}. For the purposes of this context-sensitivity check, an $\overline{\mathsf{entry}}$ edge is treated as an exit edge and vice versa.

7.1.2 Example

The code in Figure 7.2 shows an example used for illustration of the static analysis throughout the chapter. The example is based on a common usage scenario of Java containers. A simple implementation of a data structure `Container` is instantiated three times (at lines 4, 13 and 20) by a client `ContainerClient`. In the example code, statement $t[pos{+}{+}] = e$ (at line 41) is the one that achieves the functionality of *ADD*, and statement $ret = t[index]$ (at line 44) is the one achieving the functionality of *GET*. Of course, these statements do not make much sense by themselves. For each semantics-achieving statement, we also need to identify the calling contexts that are relevant to the container of interest. The (statement,contexts) pair is used later to find underutilized and overpopulated containers.

Through this example, we show how the CFL-reachability formulation of points-to analysis works. We will use t_i to denote the variable t whose first occurrence is at line i, and o_i to denote the abstract object for the allocation site at line i. For example, e_{40} and o_4 represent variable e declared at line 40 and the `Container` object created at line 4. As another example, o_{14} represents the `String` object created at line 14. Node tmp_i denotes a temporary variable created artificially to connect an object and an actual parameter. For example, tmp_6 is used to link the `Integer` object created at line 6 and the actual parameter of the call to `addElement`.

The program representation for the example is shown in Figure 7.3; for simplicity, the inverse edges are not shown. Each entry and exit edge is also treated as an assign edge for language L_{F} to represent parameter passing and method returns. The analysis can conclude that the points-to set of i_{12} has a single object o_6, because there exists a *flowsTo* path between them. To see this, first note that $this_{41}$ *alias* $this_{38}$

because object o_4 can flow to both $this_{41}$ and $this_{38}$. Similarly, $this_{44}$ *alias* $this_{38}$ and $this_{41}$ *alias* $this_{44}$ can be derived. Second, t_{41} and t_{44} are aliases because object o_{38} can flow to both t_{41} and t_{44}. For example, o_{38} *flowsTo* t_{41} can be derived as follows:

$$o_{38}\ \mathsf{new}\ t_{38}\ \mathsf{store(arr)}\ this_{38}\ \mathit{alias}\ this_{41}\ \mathsf{load(arr)}\ t_{41} \Rightarrow \quad o_{38}\ \mathit{flowsTo}\ t_{41}$$

Finally, o_6 *flowsTo* i_{12} can be derived as follows:

$$o_6\ \mathsf{new}\ tmp_6\ \mathsf{store(arr\_elm)}\ t_{41}\ \mathit{alias}\ t_{44}\ \mathsf{load(arr\_elm)}\ i_{12} \Rightarrow \quad o_6\ \mathit{flowsTo}\ i_{12}$$

In addition, this *flowsTo* path is a realizable interprocedural path because it contains matched pairs of entry and exit edges. The chain of entry and exit edges along the path is $\mathsf{entry_6} \rightarrow \mathsf{entry_{30}} \rightarrow \overline{\mathsf{entry_{30}}} \rightarrow \overline{\mathsf{entry_6}} \rightarrow \mathsf{entry_4} \rightarrow \overline{\mathsf{entry_4}} \rightarrow \mathsf{entry_8} \rightarrow \mathsf{entry_{12}} \rightarrow \mathsf{exit_{12}}$, which does not have any unmatched pair of method entry and return. (Recall that an $\overline{\mathsf{entry}}$ edge is treated as an exit edge.)

7.1.3 Formulation of Container Operations

This section presents our formulation of container operations based on the CFL-reachability formulation of points-to analysis. We first discuss a formalization of container and container operations that we are interested in. This is very similar to the container definition in Chapter 5.

Definition 7.1.1 ***(Container)****. A* container type Γ *is an abstract data type with a set Σ of element objects, and two basic operations ADD and GET that operate on Σ. A* container object γ^n *is an instantiation of Γ with n elements forming an element set Σ_γ. ADD maps a pair of a container object and an element object to a container object. GET maps a container object to one of its elements. The effects of the operations are as follows:*

- $ADD(\gamma_{pre}^{n}, o) : \gamma_{post}^{m} \quad \equiv \quad o \notin \Sigma_{\gamma_{pre}} \wedge o \in \Sigma_{\gamma_{post}} \wedge m{=}n{+}1 \wedge \forall p : p \in \Sigma_{\gamma_{pre}} \rightarrow p \in \Sigma_{\gamma_{post}}$
- $GET(\gamma^{n}) : o \quad \equiv \quad o \in \Sigma_{\gamma}$

Following the common Java practice, the definition does not allow a container to have primitive-typed elements.[9] Furthermore, for the purposes of our tool, operations that reduce the size of a container are ignored. Despite their simplicity, these two abstract container operations capture many common usage scenarios for Java containers.

To identify semantics-achieving statements for a container, we first introduce relation *reachFrom* (short for "reachable from"). For each abstract object o, set $\{o' \mid o' \textit{ reachFrom } o\}$ consists of the object itself and other abstract objects whose run-time instances could potentially be reached from an instance of o through one or more level(s) of field deference(s). The semantic domains that will be used are defined in a standard way as follows:

$\mathsf{Obj}^{\natural}$:	Domain of abstract objects, as represented by object allocation sites
$\mathsf{C}^{\natural} \subset \mathsf{Obj}^{\natural}$:	Domain of container objects
$\mathsf{V}^{\natural}$:	Domain of variable identifiers
$\mathsf{F}^{\natural}$:	Domain of instance field identifiers
$\mathsf{Call}^{\natural}$:	Domain of method entry and exit edges

[9]For brevity, we may use "container" instead of "container object".

Definition 7.1.2 ***(Relation*** *reachFrom).* *reachFrom* $\subseteq \mathsf{Obj}^\natural \times \mathsf{Obj}^\natural$ *is defined by the following production*

$$\mathit{reachFrom} \;\rightarrow\; \mathit{flowsTo}\ \mathsf{store(f)}\ \overline{\mathit{flowsTo}}\ \mathit{reachFrom} \mid \epsilon$$

In addition, the string consisting of entry *and* exit *edge labels on a reachFrom path has to be accepted by language* R_{C}*.*

For example, there exists a *reachFrom* path from o_6 to o_4. To see this, first note that o_{38} *reachFrom* o_4 holds, because of o_{38} *flowsTo* t_{38} $\mathsf{store(arr)}$ this_{38} $\overline{\mathit{flowsTo}}$ o_4. Second, there exists a *reachFrom* path from o_6 to o_{38}, because of o_6 *flowsTo* e_{40} $\mathsf{store(arr\_elm)}$ t_{41} $\overline{\mathit{flowsTo}}$ o_{38}. Finally, o_6 *reachFrom* o_4 due to the transitive property of the relation. In addition, this entire *reachFrom* path does not contain any unmatched pair of method entry and exit.

Note that an object subgraph reachable from a container (i.e., containing only nodes relevant to the container) can be computed by searching for the *reachFrom* paths ending at the container object. Nodes irrelevant to the container can be filtered out by the context-sensitivity check of language R_C. Finding *reachFrom* paths is a single-target CFL-reachability problem with $O(n^3k^3)$ complexity, where n is the number of nodes in the flow graph and k is the size of language L_{F}. However, checking context sensitivity is exponential, as the size of language R_C is exponential in the size of the program (due to the exponential number of call chains). To ensure both high precision and scalability, a time constraint (discussed later) is imposed on the analysis to inspect each container in the program. If no valid *reachFrom* paths are found within the given time budget, the analysis gives up on the container and moves on to check the next one.

Based on *reachFrom*, we can distinguish *element objects* and *inside objects* from the set of all objects reachable from the container object.

Definition 7.1.3 ***(Element object and inside object).*** *An object* $\mathtt{i} \in \mathsf{Obj}^\natural$ *is an inside object with respect to a container object* $\mathtt{c}$ *(where* $\mathtt{i}$ *is different from* $\mathtt{c}$*) if (1)* $(\mathtt{i}, \mathtt{c}) \in$ *reachFrom, and (2)* $\mathtt{i}$ *is created in the container class, its (direct or transitive) superclass, or any other class specified by the user.*

An object $\mathtt{e} \in \mathsf{Obj}^\natural$ *is an element object with respect to* $\mathtt{c}$ *if (1)* $(\mathtt{e}, \mathtt{c}) \in$ *reachFrom, (2)* $\mathtt{e}$ *is neither* $\mathtt{c}$*, nor an inside object of* $\mathtt{c}$*, and (3) for some reachFrom path from* $\mathtt{e}$ *to* $\mathtt{c}$*, all object nodes along the path (except for* $\mathtt{e}$ *and* $\mathtt{c}$*) are inside objects with respect to* $\mathtt{c}$*.*

In our example, o_4, o_{13} and o_{20} are container objects. Object o_6 is an element object for o_4; o_{14} and o_{15} are element objects for o_{13}; and o_{22} is an element object for o_{20}. Object o_{38} is an inside object for all three containers. In the rest of the discussion we will use I^c and E^c to denote the domains of inside objects and element objects with respect to a container $c \in \mathsf{C}^\natural$.

The definition of inside objects provides the flexibility to use a programmer-defined list that separates the client classes from the classes that are involved in the implementation of the container functionality. At present, the tool does not require such a list, as it targets only containers from the Java collections framework. In this case, it is sufficient to distinguish a client object from a Java collection internal object by checking if the object is created within the `java.util` package. (Note that the tool does not check the efficient use of containers within the JDK library code.) However, the class list will be useful if the tool is extended to inspect user-defined containers, because such containers may use an object created in a non-container class as an internal object. Such a (non-container) utility class should be explicitly listed as such.

The *semantics-achieving statements* are the loads/stores that read/write element objects from/to inside objects of a container.

Definition 7.1.4 ***(Semantics-achieving statements).*** *A statement that achieves the functionality of ADD with respect to a container object* $\mathtt{c}$ *is a store of the form* $\mathtt{a.f} = \mathtt{b}$ *(where* $\mathtt{a}, \mathtt{b} \in \mathsf{V}^\natural$, $\mathtt{f} \in \mathsf{F}^\natural$*), such that there exists an addTo path from an object* $\mathtt{o_b}$ *that* $\mathtt{b}$ *points to, to the container object* $\mathtt{c}$*. This addTo path has the following components: (1) a flowsTo path between* $\mathtt{o_b}$ *and* $\mathtt{b}$*, (2) an edge* $\mathtt{b} \xrightarrow{\mathsf{store(f)}} \mathtt{a}$ *representing the store, (3) a* $\overline{\textit{flowsTo}}$ *path between* $\mathtt{a}$ *and an object* $\mathtt{o_a}$*, and (4) a reachFrom path from* $\mathtt{o_a}$ *to* $\mathtt{c}$*. Using* $\oplus$ *to denote path concatenation, the path is*

$$\begin{aligned} \textit{addTo}(\mathtt{o_b}, \mathtt{c}) \;\triangleq\; & \textit{flowsTo}(\mathtt{o_b}, \mathtt{b}) \oplus \mathtt{b} \xrightarrow{\mathsf{store(f)}} \mathtt{a} \oplus \overline{\textit{flowsTo}}(\mathtt{a}, \mathtt{o_a}) \\ & \oplus \textit{reachFrom}(\mathtt{o_a}, \mathtt{c}) \end{aligned}$$

where $\mathtt{o_a}, \mathtt{o_b} \in \mathsf{Obj}^\natural$, $\mathtt{o_a} \in \mathsf{I^c}$*, and* $\mathtt{o_b} \in \mathsf{E^c}$*.*

A statement that achieves the functionality of GET with respect to a container object $\mathtt{c}$ *is a load of the form* $\mathtt{b} = \mathtt{a.f}$ *(where* $\mathtt{a}, \mathtt{b} \in \mathsf{V}^\natural$, $\mathtt{f} \in \mathsf{F}^\natural$*), such that there exists a getFrom path from an object* $\mathtt{o_b}$ *that* $\mathtt{b}$ *points to, to the container object* $\mathtt{c}$ *where*

$$\begin{aligned} \textit{getFrom}(\mathtt{o_b}, \mathtt{c}) \triangleq\; & \textit{flowsTo}(\mathtt{o_b}, \mathtt{b}) \oplus \mathtt{b} \xrightarrow{\overline{\mathsf{load(f)}}} \mathtt{a} \oplus \overline{\textit{flowsTo}}(\mathtt{a}, \mathtt{o_a}) \\ & \oplus \textit{reachFrom}(\mathtt{o_a}, \mathtt{c}) \end{aligned}$$

where $\mathtt{o_a}, \mathtt{o_b} \in \mathsf{Obj}^\natural$, $\mathtt{o_a} \in \mathsf{I^c}$*, and* $\mathtt{o_b} \in \mathsf{E^c}$*.*

In addition, the string consisting of entry *and* exit *labels on an addTo or a getFrom path has to be accepted by language* R_{C}*.*

The goal of the analysis is to identify semantics-achieving statements by finding all *addTo* and *getFrom* paths for each container. An *addTo* path is a *reachFrom* path from o_b to c, which models the process of element object o_b being added to the container. Hence, the computation of *addTo* paths can be performed along with the computation of *reachFrom* paths. However, the computation of a *getFrom* path requires a priori knowledge of *reachFrom* paths and cannot be performed until all *reachFrom* paths, element objects, and inside objects are identified. In the running example, there exists an *addTo* path from o_6 to o_4, and the semantics-achieving statement on this path is the store $t[pos{+}{+}] = e$. There also exists a *getFrom* path from o_6 to o_4, because of o_6 *flowsTo* *ret* and t_{44} $\overline{\textit{flowsTo}}$ o_{38} *reachFrom* o_4. The semantics-achieving statement on this path is the load $ret = t[index]$ at line 44.

The identification of semantics-achieving statements is not sufficient to understand the usage of a particular container object, as different container objects can have the same semantics-achieving statements. For example, all of the *addTo* (or *getFrom*) paths between o_6 and o_4, o_{14} and o_{13}, and o_{22} and o_{20} have $t[pos{+}{+}] = e$ at line 41 (or $ret = t[index]$ at line 44) as their semantics-achieving statement. These statements are executed from multiple calling contexts and it is crucial to identify the contexts that correspond to the container object to be inspected.

Definition 7.1.5 ***(Relevant context).*** *For each addTo or getFrom path* `p` *that ends at container object* `c`*, let* `r` *be the prefix of* `p` *that appears before the semantics-achieving statement on* `p`*. In other words,* `r` *is the flowsTo path before the corresponding store/load statement that achieves the ADD/GET functionality. Let* $\mathtt{l} \in \mathsf{Call}^{\natural\star}$ *be the chain of* entry *and* exit *edges (some of which may be inverted) along* `r`*. The relevant context for the semantics-achieving statement on* `p` *is a sub-chain of* `l` *that contains only the unbalanced* entry *and* $\overline{\mathsf{exit}}$ *edges.*

The chain of entry and exit edges before the semantics-achieving statement on p represents the method invocations that cause the element object o_b to flow to variable b. This chain models the process of the element object being added to/retrieved from the container. The remaining entry and exit edges on p are irrelevant for this adding/retrieving process, because they represent calls that cause the inside objects (rather than the element objects) to flow into the container. We do not need to consider balanced entry/exit edges, as they represent completed invocations along the data flow. An example will be given shortly to illustrate this modeling. Note that there could be multiple relevant contexts for a semantics-achieving statement, because a container can have multiple element objects and each element objects can be added to (and retrieved from) the container through multiple calls.

The chain of unbalanced entry and $\overline{\mathsf{exit}}$ edges, together with the semantics-achieving statement, can be used to represent a specific (ADD or GET) operation executed on a specific container object.

For example, the chain of entry and exit edges on the *addTo* path from o_6 to o_4 before semantics-achieving statement $t[pos{+}{+}] = e$ is $\mathsf{entry}_6 \rightarrow \mathsf{entry}_{30}$, which is the relevant context for the store operation with respect to container o_4. As another example, the chain on the *getFrom* path from o_6 to o_4 before the load operation $ret = t[index]$ is $\mathsf{entry}_6 \rightarrow \mathsf{entry}_{30} \rightarrow \overline{\mathsf{entry}_{30}} \rightarrow \overline{\mathsf{entry}_6} \rightarrow \mathsf{entry}_4 \rightarrow \overline{\mathsf{entry}_4} \rightarrow \mathsf{entry}_8 \rightarrow \mathsf{entry}_{12}$. Hence, the relevant context for this *getFrom* path is $\mathsf{entry}_8 \rightarrow \mathsf{entry}_{12}$, which captures the fact that element object o_6 is retrieved from container o_4. Later we will consider the relationship between the execution frequencies of statements $t[pos{+}{+}] = e$ and $ret = t[index]$ only under their respective relevant contexts $\mathsf{entry}_6 \rightarrow \mathsf{entry}_{30}$ and $\mathsf{entry}_8 \rightarrow \mathsf{entry}_{12}$.

Algorithm 2: Solving single source *addTo*-reachability.

```
Input: Flow graph, container $c$, context-insensitive points-to solution $pts$
Output: Map $solution$: pairs (heap store achieving $ADD$, relevant contexts)
1  Map⟨Statement, Set⟨Stack⟩⟩ $solution \leftarrow \emptyset$
2  Set⟨AllocNode⟩ $reachFrom \leftarrow \{c\}$                 // reachable objects
3  Set⟨AllocNode⟩ $elemObj \leftarrow \emptyset$                 // element objects
4  List⟨AllocNode⟩ $objectList \leftarrow \{c\}$                       // worklist
5  List⟨Set⟨Stack⟩⟩ $contextSetList \leftarrow \{\{$EMPTY_STACK$\}\}$
6  while $objectList \neq \emptyset$ do
7      remove an allocation node $o$ from the head of $objectList$
8      remove a set $contexts$ of context stacks from the head of $contextSetList$
9      Set⟨Stack⟩ $baseContexts \leftarrow \emptyset$
10     foreach store $a.f = b$ such that $o \in pts(a)$ do
11         foreach context stack $s \in contexts$ do
12             $baseContexts \leftarrow baseContexts \cup$ COMPUTE$FlowsTo(o, a, s)$
13         Set⟨Stack⟩ $rhsContexts \leftarrow \emptyset$
14         foreach allocation node $o_b \in pts(b)$ do
15             foreach context stack $s \in baseContexts$ do
16                 $rhsContexts \leftarrow rhsContexts \cup$ COMPUTE$\overline{FlowsTo}(b, o_b, s)$
17         if $rhsContexts \neq \emptyset$ then
18             $reachFrom \leftarrow reachFrom \cup \{o_b\}$
19             if ($o = c$ OR $o$ is an inside object) AND ($o_b$ is NOT an inside object) then
20                 $elemObj \leftarrow elemObj \cup \{o_b\}$          // An element object is found
21                 $solution \leftarrow solution \cup (a.f = b, rhsContexts)$
22             else
23                 $objectList \leftarrow append(objectList, o_b)$
24                 $contextSetList \leftarrow append(contextSetList, rhsContexts)$
25 return $solution$
```

7.1.4 Analysis Algorithms

The algorithms for solving *addTo*- and *getFrom*-reachability are shown in Algorithm 2 and Algorithm 3, respectively.

Both algorithms rely on an initial context-insensitive points-to set to find candidates for semantics-achieving statements. Algorithm 2 iteratively computes a set of reachable objects. The i-th element of list *contextSetList* keeps a set of relevant contexts for the i-th object in worklist *objectList*. Each context is represented by a stack, which contains exactly the chain of unbalanced entry and $\overline{\text{exit}}$ edges of a *flowsTo* path. Initially, *objectList* contains the container object c and *contextSetList*

Algorithm 3: Solving single source *getFrom*-reachability.

```
Input: Flow graph, container $c$, context-insensitive points-to solution $pts$, relation $reachFrom$, set $elemObj$
Output: Map $solution$: pairs (heap load achieving $GET$, relevant contexts)
1  Map⟨Statement, Set⟨Stack⟩⟩ $solution \leftarrow \emptyset$
2  foreach $allocation\ node\ o \in elemObj$ do
3      foreach $load\ b = a.f,\ such\ that\ o \in pts(b)$ do
4          Set⟨Stack⟩ $lhsContexts \leftarrow \emptyset$
5          $lhsContexts \leftarrow$ COMPUTE$FlowsTo(o, b,$ EMPTY_STACK$)$
6          Set⟨AllocNode⟩ $ins \leftarrow pts(a) \cap reachFrom$
7          Set⟨Stack⟩ $baseContexts \leftarrow \emptyset$
8          foreach $AllocNode\ o_a \in ins$ do // A candidate load
9              foreach $context\ stack\ s \in lhsContexts$ do
10                 $baseContexts \leftarrow baseContexts \cup$ COMPUTE$\overline{FlowsTo}(a, o_a, s)$
11         if $baseContexts \neq \emptyset$ then
12             $solution \leftarrow solution \cup (b = a.f, lhsContexts)$
13 return $solution$
```

contains an empty stack. Map *solution* contains pairs of semantics-achieving statement and relevant contexts, which will be returned after the function finishes. Function COMPUTE*FlowsTo* (o, a, s) at line 12 attempts to find *flowsTo* paths from an object o to a variable a, under calling context s that leads to the method creating o. Due to space limitations this function is not shown; conceptually, it is similar to the FINDPOINTSTO algorithm described in [133]. The function returns a set of contexts (i.e., stacks) that are chains of unbalanced edges on the identified *flowsTo* paths from o to a. An empty set returned means that there does not exist any valid *flowsTo* path between them.

Similarly, function COMPUTE$\overline{FlowsTo}$ (b, o_b, s) at line 16 attempts to find a $\overline{flowsTo}$ path from variable b to object o_b, under calling context s that leads to the method declaring b. The purpose of this function is to connect the chain of entry and exit edges on the *flowsTo* path from o to a with the chain of entry and exit edges on the $\overline{flowsTo}$ path from b to o_b, and to check if the combined chain corresponds to a realizable call path. If the combined chain is a realizable path (i.e., $rhsContexts \neq \emptyset$

at line 17), o_b is added to set *reachFrom* of reachable objects. Furthermore, if o_b is found to be an element object (line 20 and line 21), it is included in set *elemObj*, which will be used later by Algorithm 3. At this time, it is clear that store $a.f = b$ is a semantics-achieving statement, and its relevant contexts are contained in set *rhsContexts*. If o_b is not an element object, we append o_b to the worklist (and also append *rhsContexts* to *contextSetList*) for further processing. Some subsequent iteration of the *while* loop will use this context set to compute *flowsTo* path from object o_d to variable d for a new store $c.f = d$ (line 12), etc. In this case, remembering and eventually using *rhsContexts* ensures that no unrealizable paths can be produced during the discovery of the container's data structure.

Note that we omit a check for recursive data structures in the algorithm. In fact, an object is not added into the worklist, if it has been visited earlier during the processing of reachable objects. In other words, the back reference edges between inside objects in the object graph are ignored, because they have nothing to do with the element objects. It is also possible for an element object to have a back reference edge going to an inside object or the container object (although this is not likely to happen in practice). This back edge is also ignored, because we are interested in the process where the element object is added to the container, rather than in the shape of the data structure of the container object.

Algorithm 3 inspects each element object o computed in Algorithm 2 (line 2) and attempts to find load statements of the form $b = a.f$ such that b could point to o. As before, the algorithm starts from a context-insensitive solution, and then checks if there exists a *flowsTo* path from o to b (line 5). If such *flowsTo* paths are found, it uses the set of contexts returned (i.e., chains of unbalanced edges extracted from these paths) to compute $\overline{\mathit{flowsTo}}$ paths from a to object o_a that a may potentially point to. Note that we use an intersection between $\mathit{pts}(a)$ and *reachFrom* (line 6) to filter out irrelevant objects that are not reachable from container c. If a $\overline{\mathit{flowsTo}}$ path can be found (line 11-line 12), this load is a semantics-achieving statement and the relevant contexts for it are contained in the set *lhsContexts*.

Precision improvement Not all load statements identified in *getFrom* paths correspond to *GET* operations. For example, methods `equals` and `remove` in many container classes need to load element objects for comparison (rather than returning them to the client). To avoid the imprecise results generated in these situations, we employ a heuristic when selecting statements that implement *GET* operations. A load $b = a.f$ is selected if (1) it is on a valid *getFrom* path, and (2) the points-to set of b and the points-to set of the return variable of the method where the load is located have a non-empty intersection. This heuristic is based on the common usage of Java containers: only methods that can return element objects can be used to retrieve objects by a client.

It would be interesting to investigate other heuristics in future refinements of the analysis. In situations where individual statements are not precise enough to capture the semantics of *ADD* and *GET*, it may be possible to use such heuristics to find coarser-grained program entities (e.g., methods) that correspond to these abstract operations. For example, a splay tree may perform both stores and loads for a single *GET* operation. At present, it is unclear how to generalize the analysis to model such cases.

7.2 Execution Frequency Comparison

This section describes the dynamic profiling algorithm and the static inference algorithm to compare the execution frequencies of semantics-achieving statements.

7.2.1 Dynamic Frequency Profiling

During the execution, the profiling framework needs to record, for each container object, its *ADD* and *GET* frequencies. A key challenge is how to instrument the program in a way so that the frequencies of semantics-achieving statements can be associated with their corresponding container objects. In many cases, the container object is not visible in the method containing its semantics-achieving statements. For example, the store that implements *ADD* for `HashMap` is located in the constructor of class `HashMap.Entry` where the root `HashMap` object cannot be referenced. In this and similar cases, it is unclear where the instrumentation code should be placed to access the container object.

We use relevant contexts to determine the instrumentation points. Given a pair $(s, e_0, e_1, \ldots, e_n)$ of a semantics-achieving statement s and its context, we check whether the receiver of each call site e_i can be the container object c. This check is performed in a bottom-up manner (i.e., from n down to 0). The instrumentation code is inserted before the first call site $e_i : a.f()$ found during the check such that the points-to set of a includes c. For example, one instrumentation site for `HashMap` is placed before the call to `addEntry` in method `put`, because the receiver variable of the call site can point to the `HashMap` object:

```
class HashMap{ ...
   void put(K key, V value){ ...
      // increment the ADD frequency for "this" container
      recordADD(this);
      this.addEntry(..., key, value, ...);
   }
   void addEntry(..., K key, V value, ...){ ...
     table[...] = new Map.Entry(..., key, value, ...);
   }
}
class Entry{
   Entry(..., K key, V value, ...){
      // these are stores achieving ADD
      this.key = key;
      this.value = value;
   }
}
```

7.2.2 Static Inference of Potentially-Smaller Relationships

This subsection describes the static inference algorithm that detects inefficiencies by inferring potentially-smaller/larger relationships for the execution frequencies of two (semantics-achieving statement, contexts) pairs. These relationships are computed by traversing an interprocedural *inequality graph*, which models the interprocedural nesting among the loops containing the semantics-achieving statements.

Definition 7.2.1 ***(Inequality graph).*** *An inequality graph $IG = (\mathcal{N}, \mathcal{E})$ has node set $\mathcal{N} \subseteq \mathcal{L} \cup \mathcal{M}$, where $\mathcal{L}$ is the domain of loop head nodes, and $\mathcal{M}$ is the domain of method entry nodes. The edge set is $\mathcal{E} \subseteq \mathcal{C} \cup \mathcal{I}$, where $\mathcal{C}$ represents call edges and $\mathcal{I}$ represents inequality edges.*

Inequality Graph Construction

For two statements s_1 and s_2 that are located in loops l_1 and l_2 respectively, we say that the execution frequency of s_1 is *potentially-smaller* than the execution frequency of s_2 if l_2 is nested within l_1. Such a relationship does not necessarily reflect the real run-time execution frequencies (thus the use of "potentially"). For example, if s_2 is guarded by a non-loop predicate inside l_2, it is possible that it is executed less frequently than s_1 because the path containing s_2 can be skipped many times inside the body of l_2. One example of this situation comes from a common container implementation scenario. When the client attempts to add an object into the container, the implementation first checks if the object is already in the container, and stores it only if the container does not have it already. In this situation, the semantics-achieving statement is under the non-loop predicate that checks whether the element object is already in the container.

Despite this potential imprecision, this modeling of execution frequency, to a large degree, captures the high-level programmer's intent. For example, in many cases the programmer just wants to add objects using the nested loops without even caring about whether they have been added before. Even though the statically-inferred potentially-smaller relationship may not hold for some particular runs of a program, the problems found using this relationship may reflect inefficient uses of containers in general. In addition, the loop nesting relationship itself may clearly suggest a fix if a problem really exists. For example, the problem may be solved simply by moving some operations out of a loop. We have found this approach based on loop nesting to work well in practice.

The algorithm for constructing the inequality graph is shown in Algorithm 4. For each method m in the call graph, intraprocedural inequality edges are first added (line 4-line 11). For each loop head, we find the loop in which it is nested (line 6). If it is not nested in any loop (line 7-line 8), we create an inequality edge between the entry node of the method and the loop head node. Otherwise, the edge is created between the head of the surrounding loop and the node (line 9-line 11).

For each caller of m, a call edge is added to connect the two methods (line 14-25). Specifically, the loop where the call site for m is located (or the entry node of the caller) is found (line 17), and a call edge is created to link the head of the loop (or

Algorithm 4: Algorithm for constructing the inequality graph.

```
    Input: Call graph CG
    Output: Inequality graph IG
1   foreach method m in the call graph do
2       EntryNode entry ← GetEntryNode(m)
3       CFG cfg ← BuildCFG(m)
4       foreach loop $l \in cfg$ do                                    // add inequality edges
5           LoopHeadNode head ← GetLoopHeadNode(l)
6           Loop $l'$ ← FindSurroundingLoop(head, cfg)
7           if $l'$ = null then
8               CreateInequalityEdge($entry \xrightarrow{\leq} head$)
9           else
10              LoopHeadNode $head'$ ← GetLoopHeadNode($l'$)
11              CreateInequalityEdge($head' \xrightarrow{\leq} head$ )
12      foreach incoming call graph edge e do                          // add call edges
13          Method caller ← Source(e)
14          CFG cfg ← GetCFG(caller)
15          Loop l = FindSurroundingLoop(e.callsite, cfg)
16          if l = null then
17              EntryNode $entry'$ ← GetEntryNode(caller)
18              CreateCallEdge($entry' \xrightarrow{call(e)} entry$)
19          else
20              LoopHeadNode head ← GetLoopHeadNode(l)
21              CreateCallEdge($head \xrightarrow{call(e)} entry$)
```

the entry node of the caller) and m's entry node. Call edges are useful in filtering out irrelevant calling contexts during the traversal of the inequality graph. Figure 7.4 shows an inequality graph for the running example. Here we use e_i to denote the entry node for the method declared at line i, and l_i to denote the loop head node located at line i. Each call edge is annotated with call_i, which represents the call site at line i. Each inequality edge is annotated with $\leq_i$, where i is a globally-named index. Inverse edges are allowed for call edges: if there is a call edge $\mathsf{call_e}$ between nodes m and n, an edge $\overline{\mathsf{call_e}}$ exists between n and m. Unlike in the flow graph, there are no exit edges in the inequality graph because it is not necessary to model any data flow.

An inequality edge is used to represent only potentially-smaller relationships. The potentially-larger relationships could potentially be represented by inverse edges; however, we do not allow the use of such inverse edges because a path in the graph must represent only one of these two relationships (i.e., either smaller or larger, but not both).

Definition 7.2.2 ***(Valid potentially-smaller path).*** *Given two inequality graph nodes* `m` *and* `n`*, a path* `p` *from* `m` *to* `n` *is a valid potentially-smaller path if the chain of call edges (including inverse edges) on* `p` *forms a realizable interprocedural path (i.e., the sequence of edge labels on the chain forms a string in language* R_{C}*). Path* `p` *is a strictly-smaller path if (1)* `p` *is a valid potentially-smaller path and (2)* `p` *contains at least one inequality edge.*

One can easily define a grammar with starting non-terminal *potentially-smaller* to capture the above definition of validity. Similarly to the *flowsTo* computation, finding a valid *potentially-smaller* path in the inequality graph requires a context-sensitivity check of call and $\overline{\mathsf{call}}$ edges.

Inefficiency Detection as Source-Sink Problems

Detecting underutilized containers To find an underutilized container, we need to compare the execution frequencies of the container allocation site c and each store s that implements the functionality of ADD under relevant context r with respect to c. In the following definition, lh(`s`) denotes the loop head (if `s` is within a loop) or the entry node of the method (if `s` is not in a loop) for statement `s`.

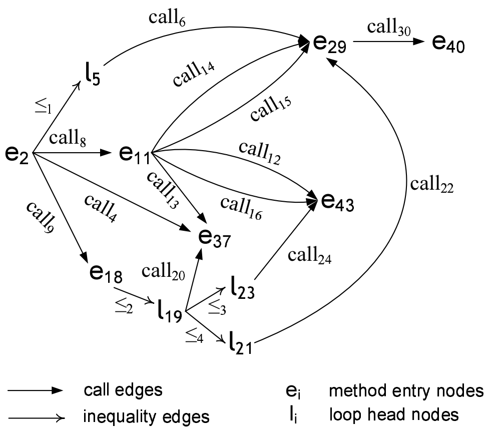


Figure 7.4: Inequality graph for the running example.

Definition 7.2.3 ***(Underutilized container).*** *Given a container allocation site* `c` *and a set of statement-contexts pairs that implement ADD operations for* `c`*, an underutilized container problem occurs for* `c` *if there does* ***not*** *exist a pair* (`store`, `contexts`) *for which (1) a strictly-smaller path* `p` *exists from lh*(`c`) *to lh*(`store`)*, and (2) there exists a context* `t` $\in$ `contexts` *such that* `p` *ends with the chain of call edges represented by* `t`*.*

Informally, an underutilized container problem is reported if there does not exist an *ADD* operation such that the loop where it is located is nested within the loop where the container allocation site is located.

Consider again the running example. Recall that the statement-contexts pair that achieves *ADD* operation for container o_4 is ($t[pos{+}{+}] = e$, $\{\mathsf{entry}_6 \rightarrow \mathsf{entry}_{30}\}$). There exists a strictly-smaller path $\leq_1\rightarrow \mathsf{call}_6 \rightarrow \mathsf{call}_{30}$ from node e_2 to e_{40}, which are the method entry nodes for the allocation site of o_4 (line 4) and for $t[pos{+}{+}] = e$ (line 41), respectively. In addition, this path contains the call chain $\mathsf{call}_6 \rightarrow \mathsf{call}_{30}$, which is exactly the context contained in set $\{\mathsf{entry}_6 \rightarrow \mathsf{entry}_{30}\}$ in the pair (comparing only labels on the call edges and the entry edges). Hence, no underutilized container problem will be reported for container o_4.

As another example, the statement-contexts pair for ADD on o_{13} is ($t[pos{+}{+}] = e$, $\{\mathsf{entry}_{14} \rightarrow \mathsf{entry}_{30}, \mathsf{entry}_{15} \rightarrow \mathsf{entry}_{30}\}$). The tool reports that o_{13} exhibits an underutilized container problem, because no strictly-smaller paths from the entry node e_{11} of its allocation site (line 13) to the entry node e_{40} of $t[pos{+}{+}] = e$ can be found, under relevant calling context $\mathsf{entry}_{14} \rightarrow \mathsf{entry}_{30}$ or $\mathsf{entry}_{15} \rightarrow \mathsf{entry}_{30}$. The problem does not occur for container o_{20}, since there is a strictly-smaller path $\leq_4\rightarrow$ $\mathsf{call}_{22} \rightarrow \mathsf{call}_{30}$ from the allocation site at line 20 to the store operation $t[pos{+}{+}] = e$ that implements ADD, under the relevant calling context $\mathsf{entry}_{22} \rightarrow \mathsf{entry}_{30}$.

Note that although the number of elements added to container o_{13} (i.e., 2) is in fact larger than the number of times its allocation site can be executed (i.e.,1), it is not a false positive to report it as an underutilized container. This is because the creation of the container (which could cost hundreds of run-time instructions) can be easily avoided by introducing extra variables for storing the data. It could be the case that, while the ADD operations for a container are in the same loop as the allocation site of the container, it may not be easy to perform an optimization because there could be a large number of distinct ADD operations (e.g., the programmer intends to add many elements without using loops). However, we have found that this situation rarely occurs in real-world programs. Once an underutilized container problem is reported, there is usually an obvious container that holds a very small number of elements, and a specialization can be easily created.

Detecting overpopulated containers To find an overpopulated container, it is necessary to compare the number of *GET* operations against the number of *ADD* operations for the container.

Definition 7.2.4 ***(Overpopulated container).*** *Given a container allocation site* $\mathtt{c}$*, a set* $\mathtt{S}_1$ *of statement-contexts pairs that implement ADD for* $\mathtt{c}$*, and a set* $\mathtt{S}_2$ *of statement-contexts pairs that implement GET for* $\mathtt{c}$*,* $\mathtt{c}$ *is an overpopulated container if for any pair* $(\mathtt{store}, \mathtt{contexts}_1) \in \mathtt{S}_1$ *and any pair* $(\mathtt{load}, \mathtt{contexts}_2) \in \mathtt{S}_2$*, (1) there exists a valid potentially-smaller path* $\mathtt{p}$ *from* $lh(load)$ *to* $lh(store)$*, and (2) there exist a context* $\mathtt{t}_1 \in \mathtt{contexts}_1$ *and a context* $\mathtt{t}_2 \in \mathtt{contexts}_2$ *such that* $\mathtt{p}$ *starts with the chain of (inverse) call edges represented by* $\overline{\mathtt{t}_2}$ *and ends with the chain of call edges represented by* $\mathtt{t}_1$*.*

Informally, an overpopulated container is reported if for every pair of *GET* and *ADD* operations, a potentially-smaller relationship can be inferred between them.

In the running example, an overpopulated container problem is detected for container o_4. Recall that the statement-contexts pair that implements *ADD* is ($t[pos{+}{+}]$ $= e$, $\{\mathsf{entry}_6 \rightarrow \mathsf{entry}_{30}\}$), and the statement-contexts pair that implements *GET* is ($ret = t[index]$, $\{\mathsf{entry}_8 \rightarrow \mathsf{entry}_{12}\}$). There exists an *potentially-smaller* path between these two statements: $\overline{\mathsf{call}_{12}} \rightarrow \overline{\mathsf{call}_8} \rightarrow \leq_1 \rightarrow \mathsf{call}_6 \rightarrow \mathsf{call}_{30}$. This path contains the call edges $\mathsf{call}_6 \rightarrow \mathsf{call}_{30}$ (i.e. t_1) and $\overline{\mathsf{call}_8} \rightarrow \overline{\mathsf{call}_{12}}$ (i.e., $\overline{t_2}$). Container o_{13} is also overpopulated, because there exists a valid *potentially-smaller* path from the *GET* to any of the two *ADD* operations.

Container o_{20} is not overpopulated, since no *potentially-smaller* path can be found from its *GET* operation (i.e., the statement-contexts pair ($ret = t[index]$, $\{\mathsf{entry}_{24}\}$)) to its *ADD* operation (i.e., pair ($t[pos{+}{+}] = e$, $\{\mathsf{entry}_{22} \rightarrow \mathsf{entry}_{30}\}$)).

Analysis Algorithm

In general, both the proof and the disproof of a certain path under certain calling contexts requires a traversal of the inequality graph. The traversal has to follow the call edges represented by a start context and an end context, which are the relevant contexts associated with semantics-achieving statements. The start context is an empty stack if the statement is the allocation site of the container. A standard worklist-based algorithm is used in the expected way to perform a breadth-first traversal of the graph. The traversal terminates immediately if call edges in the path are detected to form a cycle, because there is no way to reason about the number of *ADD* and *GET* operations for a container if calls connecting these operations are involved in recursion.

Trade-off between the analysis scalability and the amount of information produced Because the inequality graph does not contain any data flow information, the graph traversal algorithm can follow arbitrary $\textsf{call}$ and $\overline{\textsf{call}}$ edges when selecting the path. The start and end contexts are useful when the algorithm attempts to decide which $\textsf{call}/\overline{\textsf{call}}$ edge to follow. Suppose the algorithm is inspecting method m. The algorithm decides to leave m through a $\textsf{call}$ edge if (1) this edge is on the end context, or (2) it can lead to the end context. On the other hand, it follows a $\overline{\textsf{call}}$ edge if (1) the edge is on the start context, or (2) there does not exist any $\textsf{call}$ edge going out of m that is on the end context or can lead to the end context. A call edge is "leading to" a context when the method that the call edge goes to can (directly or transitively) invoke the source method of the first call edge on the context. In addition, we keep track of the set of methods that the related *addTo* and *getFrom* paths have passed. The traversal of the inequality graph never enters a method if this method is not in this set.

While the start and end contexts are useful, the worst-case time complexity of a naive graph traversal algorithm is still exponential, as the number of distinct calling contexts is exponential in the size of the program. This motivates the need to define a trade-off framework to handle the analysis scalability and the amount of information produced.

One factor considered by this framework is the number of unbalanced $\overline{\mathsf{call}}/\mathsf{call}$ edges in a valid *potentially-smaller* path. These numbers represent the length of the method sequences on the call stack that the path crosses, and they implicitly determine the running time of the algorithm. A path crossing too many calls is usually an indicator of an unrealizable interprocedural path (e.g., due to spurious call graph edges). Furthermore, even though an inefficiently-used container can be found by traversing a long interprocedural path on the inequality graph, it may be hard to optimize it as the data it carries might be needed by many places in the program. In this framework, the number of unbalanced $\overline{\mathsf{call}}/\mathsf{call}$ edges allowed in a path can be pre-set as a threshold. While this introduces unsoundness, it improves the scalability and presents to the user a set of containers that are potentially easy to specialize.

Another factor (orthogonal to the number of unbalanced $\overline{\mathsf{call}}/\ \mathsf{call}$ edges) that is taken into account is the time used to inspect each container. If a `TimeOutException` is caught during the inspection of a container, the analysis moves on to the next container, without generating any warnings about this current one. We experimented with different time thresholds, and some of these results are described in Section 7.3.

7.2.3 Comparison between Static Inference and Dynamic Profiling

Compared with all existing bloat detection techniques based on dynamic analysis [94,96,105,122,156], a major weakness of a static analysis approach is its inability to estimate precisely the execution frequencies of various statements. However, it has the following three advantages over the dynamic approaches. First, the static analysis can be used as a coding assistance tool to find container-related problems during development, before testing and tuning have begun. It is desirable to avoid inefficient operations early, even before meaningful run-time executions are possible. Second, a problem detected by the static analysis usually indicates a programmer intent (or mistake) that is inherent in the program, while the results from a dynamic analysis depend heavily on the specific run-time execution being observed. Finally, the process of locating the underlying cause from the dynamically-observed symptoms is either completely manual, or involves ad hoc techniques that do not quickly lead a tool user to the problematic code. For example, a profiler can find a container exhibiting few lookup operations, but it is hard for it to effectively explain this behavior to the programmer. The static tool explicitly reports the loops that cause the generation of the warnings, thus reducing the effort to "connect the dots" from the manifestation of the problem to the core cause.

7.3 Empirical Evaluation

We have implemented the static and dynamic analyses based on the Soot program analysis framework [132, 147], and evaluated their effectiveness on the set of 21 Java programs shown in Table 7.1. All experiments used a dual-core machine with an Intel Xeon 2.80GHz processor, running Linux 2.6.9 and Sun JDK 1.5.0 with 4GB of max heap space. The Sridharan-Bodik analysis framework from [133] was adapted to compute CFL-reachability. A parallel version of the analysis was used: 4 threads were ran to simultaneously inspect containers. Note that the total numbers of reachable methods for some programs are significantly larger than the numbers shown in previous work [133] for the same programs. This is because of the use of the JDK 1.5.0 library which is much larger than the JDK 1.3 library used in that previous work. Many of the programs were chosen from the DaCapo [15] benchmark set. The analysis was able to run on all of the DaCapo programs, but we excluded from the table the programs that do not use any Java containers. For `eclipse`, we analyzed the main framework and the following plugins that are necessary for the DaCapo run: org.eclipse.jdt, org.eclipse.core, org.eclipse.text, org.eclipse.osgi, and org.eclipse.debug.

Static analysis Table 7.1 and the first part of Table 7.2 (T_1 and T_2) show the warnings generated by the static tool, the false positives, and the running times for two different configurations: 20 and 40 minutes allowed to inspect each container. If all containers can be completely inspected under the first configuration, the second configuration is not applied, and "-" marks the corresponding column. The table shows the number of underutilized container warnings ($\#UC$), the number of overpopulated container warnings ($\#OC$), the number of containers whose inspection is not completed due to time out ($\#NC$), the number of false positives ($\#FP$), and the total running time in seconds (RT). For programs with many warnings, we randomly picked 20 warnings (including both types of problems) for manual checking; the numbers of false positives found in these samples are reported and marked with $^\sharp$.

Benchmark	*#M(K)*	*#Con*	T_1 = 20 min				
			#UC	*#OC*	*#NC*	*#FP*	*RT(s)*
jack	12.5	34	8	4	2	*	1725
javac	13.4	45	12	10	5	*	5040
soot-c	10.4	15	0	1	3	0	1235
sablecc-j	21.4	29	3	5	0	1	1140
jess	12.8	4	2	2	0	0	790
muffin	21.4	108	4	7	78	0	28213
jb	8.2	9	1	7	0	0	64
polyglot	8.6	18	1	6	1	1	1259
jflex	20.2	44	2	5	17	0	6785
jlex	8.2	16	1	0	0	0	474
java-cup	8.4	10	1	3	0	0	519
antlr	12.9	15	2	2	0	0	584
bloat	10.8	260	18	46	118	$1^\sharp$	34542
chart	17.4	286	15	29	52	$1^\sharp$	26406
xalan	12.8	1	0	1	0	0	222
hsqldb	12.5	1	0	0	0	0	99
luindex	10.7	1	0	1	0	0	69
ps	13.5	42	0	8	0	0	1077
pmd	15.3	39	8	7	0	0	1322
jython	27.5	75	5	26	5	$0^\sharp$	7055
eclipse^1	41.0	1623	18	25	1097	$0^\sharp$	447465
eclipse^2	41.0	1623	47	121	351	$3^\sharp$	32151

Table 7.1: Analysis statistics, part I. *#M* is the number of methods (in thousands) in Soot's Spark context-insensitive call graph [82, 147]. *#Con* is the total number of containers inspected in the application code. There are two rows for `eclipse`: (1) analyzing all plugins together and (2) analyzing them one at a time. Results are shown with $T_1 = 20$ minutes and $T_2 = 40$ minutes (in Table 7.2) limit for the static tool to inspect each container.

The analysis running time shown in the table includes the identification of semantics-achieving statements and the inference of potentially-smaller relationships. They are not listed separately because the running time is dominated by the former.

We have tuned the analysis by adjusting the maximum number of unbalanced $\overline{\mathsf{call}}/\mathsf{call}$ edges traversed. All numbers from 3 to 10 were evaluated. We observed that problems caused by certain container usage patterns are missing in the reports when

Benchmark	$T_2 = 40$ min					*Dynamic vs Static*				
	#UC	*#OC*	*#NC*	*#FP*	*RT(s)*	*#DU*	*#DO*	*#SN*	*#DN*	*#MS*
jack	8	4	0	*	2561	8	4	*	*	*
javac	12	10	5	*	5040	12	10	*	*	*
soot-c	0	1	0	0	1440	0	1	0	0	0
sablecc-j	-	-	-	-	-	3	5	0	4	0
jess	-	-	-	-	-	2	2	0	0	0
muffin	5	16	56	0	66472	5	16	0	18	2
jb	-	-	-	-	-	1	7	0	0	0
polyglot	0	6	1	0	2447	0	6	2	1	1
jflex	2	5	14	0	16092	2	5	1	8	0
jlex	-	-	-	-	-	1	0	0	0	0
java-cup	-	-	-	-	-	1	3	0	0	0
antlr	-	-	-	-	-	2	2	0	2	0
bloat	24	76	17	$2^\sharp$	49844	24	72	2	18	1
chart	21	38	12	$1^\sharp$	35406	21	36	3	16	1
xalan	-	-	-	-	-	0	1	0	0	0
hsqldb	-	-	-	-	-	0	0	0	0	0
luindex	-	-	-	-	-	0	1	0	0	0
ps	-	-	-	-	-	0	8	0	5	0
pmd	-	-	-	-	-	5	7	0	19	0
jython	5	26	1	$0^\sharp$	9745	5	21	1	17	2
eclipse[1]	18	32	956	$0^\sharp$	825897	18	27	0	20	0
eclipse[2]	47	137	104	$4^\sharp$	57897	45	110	5	20	0

Table 7.2: Analysis statistics, part II. The last part of the table compares the static and dynamic analyses.

this parameter is set to a number less than 7. Hence, 7 was chosen as the parameter value for the experiments. This value appears to be an appropriate choice for the set of benchmarks we used, and it may need to be re-adjusted for programs with different container usage patterns.

False positives are determined by manually inspecting each program. Given a warning, we examine the code and check whether the container is appropriately used, and whether there is a better way of using it. This is a subjective choice in which we are trying to simulate what an "intelligent programmer" would do. Although there is no objective perfect answer, such an evaluation provides valuable indication of the real-world usefulness of the tool. For programs such as `jack` and `javac` whose source code is not publicly available, we mark the *#FP* columns with "*", meaning that the warnings are not checked. Overall, the results of study are promising, since the number of false positives is low across the benchmark set.

In general, false negatives can be introduced by the unsoundness of the analysis. For example, a container problem can be missed if the container structure cannot be completely discovered within the allowed time limit, or if the call chain required to expose the problem is too long. The numbers of false negatives were not investigated because they can be reduced by increasing the resource budget (i.e., time limit and maximum call chain length). For example, for most programs in our study, a budget of $T = 40$ min allows the analysis to successfully inspect all containers in the application code (i.e., to achieve $\#NC = 0$).

The tool could not finish the inspection of many container objects in `eclipse`^{1}. This performance is caused in part by the extremely large code base of `eclipse`, and in part by the lack of precise contexts in many *flowsTo* or $\overline{\textit{flowsTo}}$ paths computed by the underlying Sridharan-Bodik framework (because of the use of "match edges" [133]). In our current work we decided to use the framework as-is and to focus on demonstrating that the approach successfully identifies problematic containers. Future work can improve the Sridharan-Bodik machinery to provide more precise context information for our analysis.

Splitting a large code base for scalability Note that both analysis time and the number of timed-out containers decrease substantially when `eclipse` plugins are analyzed individually (row `eclipse`^{2}). This is because the number of calling contexts for each container method is reduced significantly. In general, it is *not* always valid to analyze separately the components of a large program. However, separate analysis can be made more general by first employing a relatively inexpensive escape analysis, which identifies container objects that may be passed across the boundaries of components (or plugins). A container that can never escape the component where it is created can be safely analyzed in the absence of other components. This is similar to the observation that context sensitivity is not necessary for an object that never escapes its creating method. While element objects could still flow in and out of components, there exist techniques to handle incomplete programs, for example, by creating placeholders for missing objects [118]. For a non-escaping container, all semantics-achieving statements under the relevant calling contexts triggered by the creating component would be confined to that component.

Comparison with a dynamic approach The dynamic analysis instruments each semantics-achieving statement, runs the program, and reports containers whose (1) *ADD* frequencies are smaller than 10, and (2) *ADD* frequency/*GET* frequency ratios are greater than 2. The intersections between the sets of statically and dynamically generated warnings are shown in the last part of Table 7.2: *#DU* and *#DO* are the numbers of containers reported by the static analysis (from *#UC* and *#OC*, respectively) that also appear in the dynamic analysis reports. Note that most inefficiently-used containers found by the static analysis are also reported by the dynamic analysis, which shows that the static warnings indeed produce containers that exhibit problematic run-time behavior.

It is not as easy to use the dynamic analysis to find containers that are optimizable across all inputs and runs, compared to using the static analysis. Columns *#SN* and *#DN* show the numbers of containers reported by the static analysis and the dynamic analysis, respectively, for which we did *not* manage to come up with optimization solutions. To determine *#SN*, we examined each container that was already subjected to a manual check for false positives (with $T = 40$ min, and with 20 randomly chosen containers for programs with more than 20 warnings). To determine $\#DN$, we examined all dynamically-reported containers, if there were at most 20 of them; otherwise, we examined the 20 containers with the highest potential for performance improvement: the ten containers with the largest number of *ADD* operations, and the ten containers with the largest ratio of *ADD* to *GET* operations. Among all these examined containers, $\#DN$ is the number of those for which we could not determine an appropriate optimization.

In the course of this experiment, it became clear that the problems reported by the static analysis are easier to fix than those reported by the dynamic analysis. For instance, among the top 20 dynamically-reported containers for `bloat`, a program analysis framework in DaCapo, we eventually came up with optimization solutions for only two, and one of them was also in the static analysis report. The remaining containers are used to hold various kinds of program structures such as CFGs and ASTs. While they are not retrieved frequently in one particular run (with the inputs provided by DaCapo), it is hard to optimize them as their elements may be heavily used when the program is run with other inputs. In contrast, most of the container problems reported by the static analysis are straightforward and the programmer can quickly come up with optimization solutions after she understands the loop nesting relationships that cause the tool to report the warnings.

While static analysis reports can precisely pinpoint the causes of inefficiency problems, they cannot say anything about the severity of the warnings. When the analysis is used during development (i.e., without any test runs) to find container-related problems, this information may not be required because a programmer should, ideally, fix all reported warnings to avoid potential bloat. However, ranking of warnings becomes highly necessary when the tool is used for performance tuning and problem diagnosis, as the reported warnings have different performance impact and importance. For example, when we attempt to find optimization opportunities for `bloat`, it is unclear, among the total of 100 warnings generated by the static analysis, which ones to inspect first. It is impractical to examine and fix all of them, a task that can be very labor-intensive and time-consuming. A natural solution is to rank the statically-reported containers based on their run-time allocation frequencies; this was the approach used in the two case studies described below.

As discussed earlier, the time budget limitations and the constraints on call chain length may cause the static analysis to miss problematic containers that are misused in complex ways. Consider the "top" inefficiently-used containers from the dynamic reports (the same 20 or less containers examined when determining *#DN*). Column *#MS* shows the number of such containers that are missed by the static analysis. While there are only few missed containers, a more comprehensive study of their properties could be performed in future work, in order to identify new usage patterns that may be used to refine our current static analysis.

Performance improvement from specializing containers For `bloat` we found that among the containers that have warnings, the most frequently-allocated container is an `ArrayList` created by method `children` for each expression tree node to provide access to its children nodes. The number of objects added in this container is always less than three, and in many cases, it does not contain any objects. We studied the code and found an even worse problem: even when the children do not change, this container is never cached in the node. Every time method `children` is invoked, a new list is created and the node's children are added. By creating a specialized version that takes advantage of container types such as `Collections.emptyList` and `Collections.singletonList` and that caches the children list in each node, we were able to reduce the number of objects created from 129253586 to 89913518 (30% reduction), and the running time from 147 seconds to 110 seconds (24.5% reduction).

We have also inspected the report generated for `chart`. Many problems reported are centered around method `getChunks` declared in an interface, which returns an `ArrayList`. This interface is implemented by more than 10 classes, each of which has its own implementation of `getChunks`. While many of these concrete classes do not have any chunks associated, they have to create and return an empty `ArrayList`, in order to be consistent with the interface declaration. By further studying the clients that invoke these `getChunks` methods, we found that many of them need only to know the number of chunks (i.e., by invoking `getChunks().size()`). We quickly modified the code to replace the empty `ArrayList` with the specialized `Collections.emptyList`, add a method `getSize` in each of the corresponding classes that calculates the number of chunks without creating a new list, and replace the calls `getChunks().size()` with calls to the new `getSize` method. This process took us less than an hour. The modified version achieved 3.5% running time reduction and 5% reduction of the number of generated objects. Note that these case studies only addressed two obvious problems and did not go into any depth; in general, significant optimizations may be possible if a developer familiar with the code thoroughly analyzes the reported inefficiently-used containers and creates appropriate specialized versions.

7.4 Summary and Interpretation

This chapter presents a practical static analysis technique that can automatically find inefficiently-used containers. The goal of this analysis is to check, for each container, whether it has enough data added and whether it is looked up sufficient number of times. At the heart of these tools is a base static analysis that abstracts container functionality into basic operations *ADD* and *GET*, and detects them by formulating CFL-reachability problems. The identified operations can be used by a static inference engine that infers the relationship between their execution frequencies, and by a dynamic analysis that instruments these statements and finds bloat by profiling their frequencies. Experimental results show that the static analysis can scale to large Java applications and can generate precise warnings about the suspicious usage of containers. Promising initial case studies suggest that the proposed techniques could be useful for identifying container-related optimization opportunities.

This analysis is the first work that attempts to identify performance problem using static approaches, and may thus open up a new research direction where compiler optimizations should target not just low-level redundancies in the program, but also much higher level patterns of inefficiencies. In fact, the success of this technique has demonstrated that it is possible for a static analysis to identify performance problems in large object-oriented programs as long as the analysis is focused on a clear performance-impacting pattern. The static analysis can be made even more useful by combining it with some lightweight run-time information. We believe that various similar analyses can be developed when additional bloat patterns are observed in future. In fact, the next chapter presents a static analysis that targets another frequently-occurring bloat pattern we observed—loop-invariant data structures.

CHAPTER 8: Statically Detecting Loop-Invariant Data Structures to Avoid Bloat

As a culture of object-orientation, Java programmers are taught to freely create objects for whatever tasks they want to achieve, without concern for cost. They often take for granted that the runtime system can optimize away all execution inefficiencies: the Just-In-Time (JIT) compiler can remove whatever redundancy exists in the code, and the Garbage Collector (GC) can quickly reclaim redundant objects created for simple tasks. However, creating an object in Java with a `new` operator, in most cases, is far beyond allocating memory space, and can be much more expensive than a programmer realizes.

For example, object creation may need to execute large volumes of code to construct and initialize a data structure, and this process may even involve many slow I/O operations. One especially important case is when these expensive objects have data that is invariant. Frequently constructing data structures with unchanged data may have significant effect on application running time and scalability. Large performance improvements can often be seen when these data structures are reused rather than recreated.

Loops are places where such data structures can cause significant harm and thus special attention needs to be paid to find and optimize them. We propose static analyses that can find data structures that are created in a loop but are independent of specific iterations. This work is motivated by bloat patterns that are regularly

```
for(int i = 0; i < N; i++){
    SimpleDateFormat sdf = new SimpleDateFormat();
    try{
       Date d = sdf.parse(date[i]);
       ...
    }catch(...) {...}
}
                             (a)

Templates _template = factory.newTemplates(stylesheet);
while(...){
    XMLFile file = getNewInputFile();
    XMLTransformer transformer = _template.newTransformer();
    transformer.transform(file);
    ...
}
                             (b)
```

Figure 8.1: Real-world examples of heavy-weight creation of loop-independent data structures. (a) A `SimpleDateFormat` object is created inside the loop to parse the given `Date` objects; (b) An `XMLTransformer` object is created within the loop to transform the input XML files.

seen in large-scale applications. Figure 8.1 shows two examples extracted from the real-world programs that we have studied. The code pattern in part (a) has appeared a great number of times in applications that were written by IBM's customers and tuned by a group from IBM Research. The programmer may have never realized that creating one `SimpleDateFormat` object requires to load many resource bundles to get the current date, compile the default date pattern string, and load the time zone to create a calendar. The process involves many expensive operations such as object clones, hash table lookups, etc. Part (b) illustrates a problem detected by our tool in `DaCapo/xalan`. An `XMLTransformer` object is created in a loop to transform the input XML file. While the input file is updated per loop iteration, the transformer object is loop-invariant. A great amount of effort is needed to create a transformer and significant performance improvement can be achieved after hoisting the creation of this transformer. Details of this example can be found in Section 8.5.

Technical challenges While loop optimizations have been extensively studied and used in modern optimizing compilers [4], they are mostly intraprocedural and deal only with instructions that operate on scalar variables and simple data structures (e.g., arrays and linked lists). They are far from reaching our goal of finding large optimization opportunities in programs that make extensive use of object-oriented data structures. Techniques such as loop-invariant code motion target instructions whose input variables are not defined in the loop. Such techniques are usually ineffective at handling instructions involving objects: for an object created in the loop, even though one of its fields used in an instruction is not defined, it is not safe to move this instruction out of the loop, as other fields of the object may be modified elsewhere in the loop. In an object-oriented program, data abstractions are much more complex and data in different locations are tightly coupled based on logical object models.

Focusing on logical data structures In this work, we focus on the data side of the hoisting problem, that is, to find logical data structures that are loop-invariant, regardless of whether or not it is possible to hoist the actual code statements that access these data structures. If a logical data structure is loop-invariant, the programmer should modify its creating and accessing code statements in order to move it out of the loop. There are two important aspects in determining whether a logical data structure is hoistable. First, it is critical to understand *how this data structure is built up.* For example, all objects in a hoistable data structure have to be allocated together in one iteration of the loop. In addition, any object in a hoistable data structure must be owned only by this data structure, and it cannot escape to other data structures. These properties can be verified by checking *points-to relationships* among objects. Second, it is important to understand *where this data* *structure gets its values from.* For example, all values contained in (heap locations of) a hoistable data structure must *not* be computed from any loop-iteration-specific value. This aspect of the problem is naturally related to the *data dependence* problem, and thus, such (value origin) properties can be verified by checking *data-dependence* relationships.

These two kinds of relationships are formalized as two (points-to and dependence) effects by a type and effect system presented in Section 8.2. As the identification of loop-invariant data structures requires to reason about whether objects connected by these relationships are always created in the same iteration of a loop, our analysis computes, for each loop object, a *loop iteration count abstraction* that indicates whether or not an instance of the object created in one iteration of the loop can be carried over to the next iteration. A more detailed description of this abstraction can be found in Section 8.1. Section 8.2 presents a formalism that computes such abstractions.

Manual tuning with the help of hoistability measurement Given logical loop-invariant data structures identified by our analysis, the second challenge lies in how to perform the actual hoisting. While it is attractive to design a transformation technique that automatically moves invariant data structures out of loops, we found that there is little hope that a completely automated approach can effectively hoist these data structures in practice. This is first because of the over-conservative nature of any transformation technique, which may prevent the hoisting of many loop-invariant data structures due to their complex usage in large-scale applications. The chance of developing an effective transformation technique becomes even smaller in the presence of the many Java dynamic features such as dynamic class loading and reflection. Second, effectively optimizing real-world data structures requires developer insight. For example, a data structure with 100 fields cannot be transformed if it has even a single non-loop-invariant field. In fact, by manually inspecting and perhaps modifying the data model, it is highly likely that the data structure can be made hoistable (i.e., by introducing a separate object to store that loop-dependent field).

Our work advocates a semi-automated approach that is intended to identify larger optimization opportunities by bringing developer insight into the optimization process. Instead of eagerly looking only for *completely-hoistable* logical data structures, we also identify *partially-hoistable* logical data structures, by computing a *hoistability measurement* for each logical data structure, and rank all such data structures based on these measurements to help manual tuning. The higher measurement a data structure has, the more likely it is that it can be manually hoisted.

One additional advantage of manual tuning using hoistability measurements is that these metrics can be easily modified to incorporate dynamic information obtained from a profile. Section 8.3 presents one such modification that includes loop frequencies in the metrics so that the "loop hotness" factor is taken into account when the rank of a data structure is computed. To optimize a non-hoistable data structure, the programmer can either split the data model (e.g., to separate the loop-invariant fields and non-invariant fields) and/or restructure the statements that access it (e.g., to eliminate dependences between hoistable and non-hoistable statements). In addition, highly-ranked data structures can often be indicators of other loop-related inefficiencies, such as inappropriate implementation choices. These problems may also be revealed during the inspection of the reported data structures.

Evaluation We evaluated our technique using a set of 19 Java programs. With the help of hoistability measurements, we found optimization opportunities in most of these programs. We discuss the performance gains we have achieved for five representative programs: `ps`, `xalan`, `bloat`, `soot-c`, and `sablecc-j`. For example, we found a performance problem in `DaCapo/xalan`; removing it can improve the benchmark performance by 10.1%. As another example, we found a bottleneck in the core components of `ps`. After the optimization, the running time was reduced by 82.1%. Detailed description of the empirical evaluation can be found in Section 8.5. These results indicate that the proposed technique can be useful both in the coding phase (for finding small performance issues before they accumulate) and in the tuning phase (for identifying performance bottlenecks).

8.1 Overview

Figure 8.2 shows a simple running example. This example contains 10 allocation sites (including string literals), and all of them are located in loops (i.e., either directly in a loop or in a method invoked in a loop). Our analysis inspects each object[10] located in a loop and discovers its structure (including objects that are context-sensitively reachable from it) using a context-free language (CFL)-reachability formulation proposed in Chapter 7 [159]. Using this formulation, the analysis inspects individual objects in a loop, identifies their structures, and computes their hoistability, all without requiring a pre-computed whole-program points-to solution.

Points-to relationships Figure 8.3(a) illustrates three data structures (a.1), (a.2), and (a.3) that are rooted at objects in the two loops shown in Figure 8.2. Each

[10] "Object" will be used in the rest of this chapter to denote a static abstraction (i.e., an allocation site), while "instance" denotes a run-time object.

object is given a name o_i, where i is the number of the line in the code where the object is created. Each edge in a data structure represents a points-to relationship, and is annotated with a field name and a pair of integers $(i,\ j)$. Field `elm` is a special field used to represent array elements. Integers in this pair are the loop *iteration count abstractions* (ICAs) for the two objects connected by this edge, and they can be used to determine whether these objects are created in the same iteration of the loop. Following the iteration abstraction [100] and the recency abstraction [13], an ICA can be one of three (abstract) values: 0, 1, or $\top$. Note that the use of ICA is not a contribution of this dissertation: the major contribution lies in the formulation of loop-invariant data structure detection into computing ICA-annotated points-to and dependence relationships, and in the development of quantitative measurements that use these relationships to help programmers identify hoistable data structures.

For a particular loop l, an object whose ICA is 0 with respect to l must be created outside l. The ICA for an object being either 1 or $\top$ (with respect to l) means the object must be created inside l. In particular, let us consider a run-time iteration p of l and a run-time instance r created by allocation site o_r such that r is live during the execution of p. If the ICA for o_r is 1, r is guaranteed to be created during iteration p. In other words, the ICA for an object being 1 indicates that its instances must be "fresh“ across iterations, that is, in any iteration where an instance of it is live, this instance must be created in that iteration (i.e., it must *not* be carried over from a previous iteration). An object that has a $\top$ in its ICA is created in a (previous) unknown iteration. For example, the ICA for o_{23} is 1, as it creates a fresh object in each iteration of the loop at line 20. The ICA for o_{22} is $\top$, as the instance it creates in one iteration can be carried over to the next iteration.

```
1 class List{
2   Object[] arr; int index = 0;
3  List() { arr = new Object[1000];}
4   void add(Object o){ if(index < 1000) arr[index++] = o;}
5   Object get(i){ return arr[i]; }
6 }
7 class Pair{
8   Object f; Object g;
9   Pair(Object o1, Object o2){ this.f = o1; this.g = o2;}
10}
11
12 class Client{
13   static void main(String[] args){
14     for(int i = 0; i < 500; i++){
15        List l = new List();
16        Pair p = new Pair("hello", "world");
17        l.add(p);
18     }
19     Integer b = null;
20     for(int j = 0; j < 400; j++){
21        Integer a = new Integer(j);
22        if(j == 20) b = new Integer(10);
23        Pair q = new Pair(a, b);
24        Pair r = new Pair("good", a);
25        ... //use q and r
26     }
27  }
```

Figure 8.2: Running example.

Hence, for a points-to edge annotated with $(i,\, j)$, $i = j = 1$ guarantees that the two objects connected by the edge must be created in the same iteration of the loop, while either i or j being $\top$ indicates that the two objects may be created in different iterations. A data structure is obviously not hoistable if it contains objects that are created in different iterations.

Dependence relationships Figure 8.3(b) illustrates the dependence (def-use) chains that start at memory locations in each data structure shown in Figure 8.3(a). An edge $o_1.f \leftarrow o_2.g$ indicates that a run-time value contained in a heap location abstracted by $o_2.g$ is required in computing a value written into (a heap location abstracted by) $o_1.f$. Note that $o_{21}.\mathit{value}$ is a field of class `Integer` that stores the int value embedded in an `Integer` object. A stack location is also considered in a

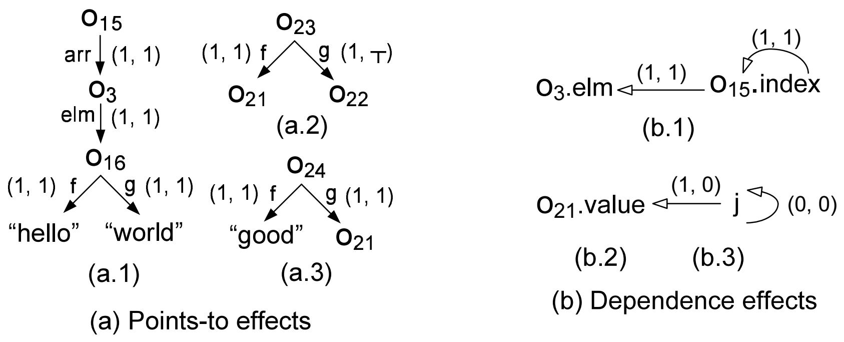


Figure 8.3: Data structures identified for the running example and their effect annotations. (a) Points-to effects among objects; (b) Dependence effects among memory locations.

dependence chain, if this location (variable) has a primitive type and is in the method that contains the loop of interest (e.g., variable j in method `main`). Such a variable needs to be taken into account as it may contain iteration-specific values. Variables that are not in the loop-containing method are abstracted away from dependence chains, as they can get iteration-specific values only from heap locations or variables in the loop-containing method (e.g., variable o at line 4). Reference-typed variables (e.g., a and b) do not need to be considered as well (even though they are in the loop-containing method), because they can get loop-specific values only from heap locations, and thus, it is necessary to track only heap locations.

Note that each dependence edge $o_1.f \leftarrow o_2.g$ is also annotated with a pair of ICAs (for o_1 and o_2), which is used to determine whether o_1 and o_2 are always created in the same loop iteration. If a node in a dependence edge is a stack variable, such as j, its ICA is determined by whether or not it is declared in the loop. For example, j's ICA is 0, because it is declared before the loop starts. Its ICA would have been 1 if it were declared in the loop. The ICA for a stack variable can never be $\top$, as each variable declared in a loop must be initialized (i.e., get a new value) per iteration.

Hoistable logical data structures As discussed earlier, the analysis identifies (completely or partially) hoistable logical data structures from a *purely data perspective*, regardless of the actual code and control flow. This can be done by reasoning about these two kinds of annotated relationships. A hoistable data structure has the following important properties.

(1) *(Disjoint).* Its run-time instances have to be disjoint. No object is allowed to appear in multiple instances of one single logical data structure. This property can be verified by checking whether all points-to edges in a data structure are annotated with (1, 1). A data structure is not hoistable if any of its nodes has a non-1 ICA. This guarantees that any instance of a hoistable data structure does not have objects created outside the loop or in different iterations. For example, (a.2) is not hoistable, as edge $o_{23} \xrightarrow{g} o_{22}$ may connect objects created in different iterations.

(2) *(Loop-invariant).* Fields of objects in a hoistable data structure have to be loop-invariant. No data in any run-time instance of the data structure can be dependent on specific loop iterations. We check this property by formulating it as a data-dependence problem. A sufficient condition for the statement "object o is loop-invariant" is that, for each field of o, (2.1) no edge on a dependence chain (e.g., shown in Figure 8.2(b)) that starts from the field can have $\top$ in its annotated ICA pair, and (2.2) for each memory location node $o.f$ (i.e., heap location) or j (i.e., stack location) on the chain, if the ICA for o or j is 0, this node must not be involved in a *dependence cycle*.

(2.1) enforces that all (stack and heap) locations from which a hoistable data structure instance (allocated in one iteration) gets its data are either created in this same iteration, or exist before the loop starts. No data in this instance can be obtained from an object created in a different iteration. In addition, as stated in (2.2), if one such location already exists before the loop starts (e.g., variable j), this node must not be in a dependence cycle. Otherwise, its value may be updated by each iteration and any data structure that is dependent on this value is not hoistable. For example, in Figure 8.3(b), $o_{21}.value$ depends on variable j, whose ICA is 0. Because j is in a dependence cycle, $o_{21}.value$ may have iteration-specific values, and thus, any data structure that contains o_{21} is not hoistable (e.g., structures (a.2) and (a.3) in Figure 8.3(a)). Note that field $o_3.elm$ is loop-invariant: while it depends on field $o_{15}.index$, which is involved in a cycle, o_{15}'s ICA is 1 and thus, it is impossible for an iteration-specific value to propagate to this field.

Note that these two properties are sufficient (but not necessary) conditions for hoistable logical data structures. For example, the first condition (i.e., disjointness) is an over-conservative approximation of the shape of a hoistable data structure—it is perfectly possible for a hoistable data structure to contain objects that are created outside the loop (i.e., their ICAs are 0) and but not mutated in the loop. We choose not to consider such objects in our hoistable data structure definition primarily for scalability purposes—these objects (created outside the loop) often have extremely long dependence chains (as objects that they reference can come from arbitrary places) and thus tracking these chains (for checking the second condition) can reduce the scalability of our analysis by orders of magnitude. On the other hand, dependence chains for objects that are created in the loop and do not escape the loop (i.e., all their ICAs are 1) are generally much shorter and the dependence analysis is much more scalable when considering only these chains.

Computing hoistability measurements After inspecting these two conditions, it is clear that only data structure (a.1) is a completely hoistable logical data structure. However, there might still exist optimization opportunities with the other two (partially hoistable) data structures. For example, if we can move object o_{22} out of data structure (a.2), it may still be possible to hoist (a.2). In order to help programmers discover such hidden optimization opportunities, hoistability measurements are proposed to quantify the likelihood of manually hoisting data structures out of loops.

For example, for each data structures shown in Figure 8.3, we compute two separate hoistability measurements based on the two orthogonal (points-to and dependence) effects mentioned above: structure-based hoistability (SH) that considers how many objects in the data structure must be allocated in the same loop iteration (i.e., that comply with condition 1), and dependence-based hoistability (DH) that considers how many fields in the data structure must contain loop-invariant data (i.e., that comply with condition 2). Eventually, these three data structures are ranked based on the two measurements and are then presented to the user for further inspection. Detailed description of hoistability measurements can be found in Section 8.3.

8.2 Loop-Invariant Logical Data Structures

This section formalizes the notion of loop-invariant logical data structure, and in this context, formally defines our analysis that identifies hoistable data structures.

Variables	a, b	$\in$	V
Allocation sites	o	$\in$	O
Instance fields	f	$\in$	F
Labels	l	$\in$	L
Statements	e	$\in$	E

$$\begin{aligned} e ::= \;& a = b \mid a = \mathit{new}\ \mathit{ref}^{o} \mid a = b.f \mid a.f = b \mid a = \mathsf{null} \mid \\ & e\ ;\ e \mid \text{if } (*) \text{ then } e \text{ else } e \mid \text{while}^{l}\ (*) \text{ do } e \end{aligned}$$

(a)

Iteration count	i	$::=$	$0 \mid 1 \mid 2 \mid \ldots$	$\in \mathsf{N}$
Iteration map	ν	$\in$	$\mathsf{L} \rightarrow \mathsf{N}$	
Loop status	π	$::=$	$\langle l\ ,\ i \rangle$	$l \in \mathsf{L} \cup \{0\}$
Labeled object	$\hat{o}$	$::=$	o^{π}	$\in \Phi$
Heap	σ	$\in$	$\Phi \times \mathsf{F} \rightarrow \Phi \cup \{\perp\}$	
Environment	ρ	$\in$	$\mathsf{V} \rightarrow \Phi \cup \{\perp\}$	
Data origin	μ	$\in$	$\mathsf{V} \rightarrow 2^{\Phi \times \mathsf{F}}$	
Heap points-to effect	H	$::=$	$\emptyset \mid H \cup \{\hat{o}_1 \rhd^{f} \hat{o}_2\}$	
Heap data dep. effect	$\varOmega$	$::=$	$\emptyset \mid \varOmega \cup \{\hat{o}_1.f \prec \hat{o}_2.g\}$	

(b)

Figure 8.4: A simple `while` language: (a) abstract syntax; (b) semantic domains.

The presentation proceeds in three steps. First, we define a simple imperative language and present its abstract syntax and operational semantics, which we will use to formalize our analysis algorithms. Second, we present a type and effect system that abstracts concrete objects and effects. Finally, the analysis that detects hoistable data structures is described based on the abstract heap effects generated by the type and effect system.

8.2.1 Language, Semantics, and Effect System

Language The abstract syntax and the semantic domains for the simple `while` language that we use are defined in Figure 8.4. A program in this language has a fixed set of global variables with reference types. While primitive-typed variables are considered in our analyses, they are excluded from this language for the simplicity of presentation. Each allocation site is labeled with an ID o. Each loop is annotated with an natural number label l ($l > 0$), which will be used as the ID of the loop.

We develop a concrete operational semantics for the language in order to detect hoistable data structures. A loop iteration count i records the number of iterations that a loop has executed. A global loop iteration map ν maps each loop (label) to its current iteration count. Each object instance is represented as its allocation site o annotated with a pair $\langle l, i \rangle$, where l is the label of the loop in which o is located (always > 0), and i is the count of the iteration of l that creates this instance. If an object is not located in any loop, the loop status π for its instances is always $\langle 0, 0 \rangle$. For simplicity, we assume our loops are not nested. While nested loops can be handled easily in our framework (e.g., by creating and associating with each object an *iteration count map* that records an iteration count for each loop in which the object is located), we found that it is not useful in hoisting data structures for real-world Java programs: it is extremely rare that a data structure can be hoisted out of multiple loops.

A heap σ records object reference relationships, and an environment ρ maps variables to objects in the heap. They are defined in standard ways. A data origin map μ records, for each stack variable v, a set of *heap locations* such that values in these locations are required (i.e., either as a pointer for dereferencing, or being copied through a sequence of intermediate stack locations) to obtain a value written to v. This map tracks dependences between variables and their relevant heap locations, and will be used to compute *dependence effects* as described shortly. For example, after the execution of a sequence of statements $c = d.f; b = c; a = b$, we have $\mu(a) = \mu(b) = \mu(c) = \{o_d.f\} \cup \mu(d)$, where o_d represents the object that d points to. $\mu(d)$ is included here because d is required as a reference to an object from which the value is obtained. Dependences via the intermediate stack locations (e.g., b and c) are abstracted away as we are interested only in fields of objects that form data structures.

Note that μ records only one-hop heap location dependence—if the value in $d.f$ is obtained from another heap location, $\mu(a)$, $\mu(b)$ and $\mu(c)$ remain the same. Multi-hop heap location dependences can be obtained by computing the transitive closure of μ.

As discussed earlier in Section 8.1, we use *points-to* and *dependence* relationships to reason about (1) how data structures are built up and (2) where they get values from, respectively. These relationships are modeled by the following two kinds of effects in our system. A heap *points-to effect* $\hat{o}_1 \rhd^f \hat{o}_2 \in H$ is generated if, at a certain point, object $\hat{o}_2$ becomes reachable from $\hat{o}_1$ through field f. A *data dependence effect* tracks the flow of data. One such effect $\hat{o}_1.f \prec \hat{o}_2.g \in \Omega$ indicates that $\hat{o_2}.g$ is required to compute a value written into $\hat{o_1}.f$. This effect captures a transitive data dependence relationship between two heap locations, abstracting away a possible sequence of dependences via intermediate stack variables. Data dependence effects can be computed efficiently by using the data origin map μ.

Note that in this language, it is safe for a dependence chain to *not* include any stack variable (like j in Figure 8.3 (b)). This is because the language supports only reference-typed variables, which can never form a dependence cycle themselves (without a heap location involved). While we choose not to include primitive types in this

$$a = \text{null}, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma, \rho[a \mapsto \bot], \mu[a \mapsto \emptyset], \emptyset, \emptyset \quad (\textsc{Assign-Null})$$

$$\frac{\rho' = \rho[a \mapsto \hat{o}] \qquad \sigma' = \sigma[\forall f.(\hat{o}.f \mapsto \bot)] \qquad \hat{o}.o = \mathit{alloc} \qquad \hat{o}.\pi = \langle l\ ,\ \nu(l)\rangle}{a = \text{new ref}^{\mathit{alloc}}, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma', \rho', \mu[a \mapsto \emptyset], \emptyset, \emptyset} \quad (\textsc{New})$$

$$a = b, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma, \rho[a \mapsto \rho(b)], \mu[a \mapsto \mu(b)], \emptyset, \emptyset \quad (\textsc{Assign})$$

$$\frac{\rho(b) = \hat{o} \qquad \mu' = \mu[a \mapsto \mu(b) \cup \{\hat{o}.f\}]}{a = b.f, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma, \rho[a \mapsto \sigma(\hat{o}.f)], \mu', \emptyset, \emptyset} \quad (\textsc{Load})$$

$$\frac{\begin{array}{c}\rho(a) = \hat{o}_1 \qquad \rho(b) = \hat{o}_2 \\ H = (\hat{o}_2 = \text{null}\ ?\ \emptyset : \{\hat{o}_1 \rhd^f \hat{o}_2\}) \qquad \mathit{\Omega} = \bigcup\{\hat{o_1}.f \prec \hat{o}_i.g_i \mid \hat{o}_i.g_i \in \mu(a) \cup \mu(b)\}\end{array}}{a.f = b, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma[\hat{o}_1.f \mapsto \hat{o}_2], \rho, \mu, H, \mathit{\Omega}} \quad (\textsc{Store})$$

$$\frac{e_1, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H_1, \mathit{\Omega}_1 \qquad e_2, \nu', \sigma', \rho', \mu' \Downarrow \nu'', \sigma'', \rho'', \mu'', H_2, \mathit{\Omega}_2}{e_1; e_2, \nu, \sigma, \rho, \mu \Downarrow \nu'', \sigma'', \rho'', \mu'', H_1 \cup H_2, \mathit{\Omega}_1 \cup \mathit{\Omega}_2} \quad (\textsc{Comp})$$

$$\frac{e_1, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \mathit{\Omega}}{\text{if } (*) \text{ then } e_1 \text{ else } e_2, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \mathit{\Omega}} \quad (\textsc{If-Else-1})$$

$$\frac{e_2, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \mathit{\Omega}}{\text{if } (*) \text{ then } e_1 \text{ else } e_2, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \mathit{\Omega}} \quad (\textsc{If-Else-2})$$

$$\frac{e, \nu[j \mapsto \nu(j)+1], \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H_1, \mathit{\Omega}_1 \qquad \text{while}^j\ (*)\ \text{do}\ e, \nu', \sigma', \rho', \mu' \Downarrow \nu'', \sigma'', \rho'', \mu'', H_2, \mathit{\Omega}_2}{\text{while}^j\ (*)\ \text{do}\ e, \nu, \sigma, \rho, \mu \Downarrow \nu'', \sigma'', \rho'', \mu'', H_1 \cup H_2, \mathit{\Omega}_1 \cup \mathit{\Omega}_2} \quad (\text{W})$$

Figure 8.5: Concrete instrumented semantics.

language (for the simplicity of presentation), our implementation handles both primitive and reference types. It is also important to note that the effects shown in Figure 8.4 are *concrete effects*. We will present an approach to project them into *abstract effects*, which are essentially the annotated edges shown in Figure 8.2.

Concrete instrumented semantics Figure 8.5 presents a big-step operational semantics for our language. A judgment of the form

$$e,\ \nu,\ \sigma,\ \rho,\ \mu \Downarrow \nu',\ \sigma',\ \rho',\ \mu',\ H,\ \mathit{\Omega}$$

starts with a statement e, which is followed by loop iteration map ν, heap σ, environment ρ, and value origin map μ. The execution of e terminates with a final iteration vector ν', heap σ', environment ρ', origin map μ', heap points-to effect set H, and heap data dependence effect set $\mathit{\Omega}$.

Rules Comp, If-Else1, If-Else2, and W are defined in expected ways. In rule Assign-Null and New, the data origin for a (in μ) is assigned $\emptyset$, as the value is freshly generated and does not depend on any heap value. In rule New, for a labeled object $\hat{o}$, $\hat{o}.o$ and $\hat{o}.\pi$ denote the allocation site and the loop status pair for $\hat{o}$, respectively. In $\hat{o}.\pi$, the loop l where the allocation site o is located is determined statically, and is associated with each instance created by o. The iteration count for l is retrieved from the global iteration count map ν. Rule Assign propagates both the object reference and the data origin of this reference value from b to a. In rule Load, the data origin map μ for variable a is updated in a way so that both $\hat{o}.f$ and the origin of the value in b are recorded as a's origin. Hence, dependences via both the value copy (from $b.f$ to a) and the pointer dereference (i.e., dereferencing b) are captured.

To handle a store $a.f = b$ where b's value is written to the heap, a points-to effect $\hat{o}_1 \rhd^f \hat{o}_2$ is first generated. The rule next generates a set of heap dependence effects $\{\hat{o}_1.f \prec \hat{o}_i.g_i \mid \hat{o}_i.g_i \in \mu(b) \cup \mu(a)\}$, by consulting the data origin map μ. Each dependence effect states that a value read from field g_i of object $\hat{o}_i$ has been used to produce a value written to $\hat{o}_1.f$ during the execution.

▶ **Example** For illustration, consider the following example:

$a = \texttt{new ref}^{o_1}$; $e = \texttt{new ref}^{o_2}$; $a.f = e$; $j = a.f$;

$\texttt{while}^1$ (j) $\texttt{do}\{$ $b = a.f$; $d = b$; $c = \texttt{new ref}^{o_3}$; $c.g = d$;$\}$

At the end of the first iteration of the loop, the semantic domains contain the following values:

$\nu = [1 \mapsto 1]$,
$\sigma = [\hat{o_1}.f \mapsto \hat{o_2},\ \hat{o_3}.g \mapsto \hat{o_2}]$,
$\rho = [a \mapsto \hat{o_1},\ b \mapsto \hat{o_2},\ c \mapsto \hat{o_3},\ d \mapsto \hat{o_2},\ e \mapsto \hat{o_2},\ j \mapsto \hat{o_2}]$,
$\mu = [a \mapsto \emptyset,\ b \mapsto \{\hat{o_1}.f\},\ c \mapsto \emptyset,\ d \mapsto \{\hat{o_1}.f\},\ e \mapsto \emptyset,\ j \mapsto \{\hat{o_1}.f\}]$,

$H = \{\hat{o_1} \rhd^f \hat{o_2},\, \hat{o_3} \rhd^g \hat{o_2}\}$,
$\Omega = \{\hat{o_3}.g \prec \hat{o_1}.f\}$. ◀

Before proceeding to the type and effect system, we need to show that our operational semantics adequately captures the heap points-to and dependence effects.

Lemma 8.2.1 *(*Operational semantics adequacy*) In the operational semantics shown in Figure 8.5, H captures all possible run-time points-to relationships; Ω records, for each heap location $o.f \in \textsc{Dom}(\sigma)$, all possible (one-hop) heap locations that have been used to (directly or transitively) produce values written into $o.f$.*

Proof sketch. It is trivial to show that after the program terminates, H contains all possible heap points-to effects: a points-to relationship can be established only when a store is executed, and it is recorded by rule STORE. To demonstrate Ω captures all (one-hop) dependences among heap locations, we first have to show that data origin map μ captures all the heap locations that each variable a depends on at run time. This can be seen by induction on the derivation. The key is to show that (1) if the value of variable a is computed directly from a heap value, this dependence must be captured in μ at a load that reads that heap value, and (2) the dependence is appropriately propagated via a copy assignment (by rule ASSIGN) and via another load that dereferences a (by rule LOAD). Next, since Ω is updated only at stores, it is necessary to focus on the execution of a store $a.f = b$ to show the adequacy of Ω: all values necessary to execute this store are in (stack) locations a and b, and the set of all heap locations that are used to produce these values is already included in $\mu(a) \cup \mu(b)$ at this point (from the above reasoning about μ). Hence, all of the heap locations on which $o_a.f$ is data dependent are recorded in Ω . □

Iteration count abstr.	$\tilde{i}$	$::=$	$0 \mid 1 \mid \top$	$\in \mathbb{N}$
Loop status abstr.	$\tilde{\pi}$	$::=$	$\langle l, \tilde{i} \rangle$	$l \in \mathsf{L} \cup \{0\}$
Type	$\tilde{\tau}$	$::=$	$o^{\tilde{\pi}} \quad \mid \top$	$\in \mathbb{T}$
Type environment	Γ	$\in$	$\mathsf{V} \rightarrow \mathbb{T} \cup \{\perp\}$	
Data origin abstr.	$\tilde{\mu}$	$\in$	$\mathsf{V} \rightarrow 2^{\mathbb{T} \times \mathsf{F}}$	
P.T. effect abstr.	$\tilde{H}$	$::=$	$\emptyset \mid \tilde{H} \cup \{\tilde{\tau}_1 \trianglerighteq \tilde{\tau}_2\}$	
Dep. effect abstr.	$\tilde{\Omega}$	$::=$	$\emptyset \mid \tilde{\Omega} \cup \{\tilde{\tau}_1.f \preceq \tilde{\tau}_2.g\ \}$	

Figure 8.6: Syntax of types and abstract effects.

Abstract semantics The concrete semantics uses an unbounded number of objects and unbounded loop iteration counts. We next develop a type and effect system that describes an abstract semantics, which conservatively approximates the concrete semantics with a bounded set of objects and bounded loop iteration counts. The syntax of types and abstract effects are illustrated in Figure 8.6. The abstraction of each concrete domain (e.g., π) shown in Figure 8.4 is represented by its corresponding tilded symbol (e.g, $\tilde{\pi}$). Environment ρ is abstracted by the type environment, denoted by Γ. A type $\tilde{\tau}$ abstracts a labeled object instance $\hat{o}$ by projecting its concrete iteration count $\hat{o}.\pi.i$ to an iteration count abstraction (ICA) $\tilde{\tau}.\tilde{\pi}.\tilde{i}$, which can have three abstract values: 0, 1, and $\top$. The meaning of these values was explained in Section 8.1. Using this type and effect system, we can identify data structures whose objects are guaranteed to be created in the same iteration by reasoning about object ICAs. Note that each abstract effect in $\tilde{H}$ and in $\tilde{\Omega}$ corresponds to an edge in Figure 8.3 (a) and in Figure 8.3 (b), respectively.

Figure 8.7 shows the type rules, which are parallel with the operational semantics in Figure 8.5. Auxiliary operations used in the type rules are shown in Figure 8.8.

$$\Gamma, \tilde{\mu} \vdash a = \text{null} : \Gamma'[a \mapsto \bot], \tilde{\mu}[a \mapsto \emptyset], \emptyset, \emptyset \quad (\text{TAssign-Null})$$

$$\frac{\Gamma' = \Gamma[a \mapsto \tilde{\tau}] \qquad \tilde{\tau}.o = \mathit{alloc} \qquad \tilde{\tau}.\tilde{\pi} = \langle l\ ,\ 1 \rangle}{\Gamma, \tilde{\mu} \vdash a = \text{new ref}^{\mathit{alloc}} : \Gamma', \tilde{\mu}[a \mapsto \emptyset], \emptyset, \emptyset} \quad (\text{TNew})$$

$$\Gamma, \tilde{\mu} \vdash a = b : \Gamma[a \mapsto \Gamma(b)], \tilde{\mu}[a \mapsto \tilde{\mu}(b)], \emptyset, \emptyset \quad (\text{TAssign})$$

$$\frac{\tilde{\tau} = \Gamma(b) \qquad \tilde{\mu}' = \tilde{\mu}[a \mapsto \tilde{\mu}(b) \cup \{\tilde{\tau}.f\}]}{\Gamma, \tilde{\mu} \vdash a = b.f : \Gamma[a \mapsto \top], \tilde{\mu}', \emptyset, \emptyset} \quad (\text{TLoad})$$

$$\frac{\tilde{\Omega} = \bigcup\{\tilde{\tau_1}.f \preceq \tilde{\tau_i}.g_i \mid \tilde{\tau_i}.g_i \in \tilde{\mu}(b) \cup \tilde{\mu}(a)\} \qquad \begin{array}{c}\tilde{\tau_1} = \Gamma(a) \\ \tilde{\tau_2} = \Gamma(b)\end{array} \qquad \tilde{H} = \{\tilde{\tau_1} \trianglerighteq^f \tilde{\tau_2}\} \text{ if } \tilde{\tau_1} \neq \bot \text{ and } \tilde{\tau_2} \neq \bot,\ \emptyset \text{ otherwise}}{\Gamma, \tilde{\mu} \vdash a.f = b : \Gamma, \tilde{\mu}, \tilde{H}, \tilde{\Omega}}$$
$$(\text{TStore})$$

$$\frac{\Gamma, \tilde{\mu} \vdash e_1 : \Gamma', \tilde{\mu}', \tilde{H}_1, \tilde{\Omega}_1 \qquad \Gamma', \tilde{\mu}' \vdash e_2 : \Gamma'', \tilde{\mu}'', \tilde{H}_2, \tilde{\Omega}_2}{\Gamma, \tilde{\mu} \vdash e_1; e_2 : \Gamma'', \tilde{\mu}'', \tilde{H}_1 \cup \tilde{H}_2, \tilde{\Omega}_1 \cup \tilde{\Omega}_2} \quad (\text{TComp})$$

$$\frac{\Gamma, \tilde{\mu} \vdash e_1 : \Gamma', \tilde{\mu}', \tilde{H}_1, \tilde{\Omega}_1 \qquad \Gamma, \tilde{\mu} \vdash e_2 : \Gamma'', \tilde{\mu}'', \tilde{H}_2, \tilde{\Omega}_2}{\Gamma, \tilde{\mu} \vdash \text{if } (*) \text{ then } e_1 \text{ else } e_2 : \Gamma' \uplus \Gamma'', \tilde{\mu}[\forall v.(v \mapsto \tilde{\mu}'(v) \cup \tilde{\mu}''(v))], \tilde{H}_1 \cup \tilde{H}_2, \tilde{\Omega}_1 \cup \tilde{\Omega}_2} \quad (\text{TIf-Else})$$

$$\frac{\Gamma[\forall v.(v \mapsto (\Gamma(v).o)^{(\Gamma(v).\tilde{\pi}^j \oplus 1)})], \tilde{\mu} \vdash e : \Gamma, \tilde{\mu}, \tilde{H}, \tilde{\Omega}}{\Gamma, \tilde{\mu} \vdash \text{while}^j\ (*) \text{ do } e : \Gamma, \tilde{\mu}, \tilde{H}, \tilde{\Omega}} \quad (\text{TW})$$

Figure 8.7: Typing.

Some abstract semantic domains in Figure 8.6 are extended with $\top$ and/or $\bot$ elements, as necessary.

Since the type and effect system abstracts the concrete semantics in Figure 8.5, most of the rules in Figure 8.7 are similar to their corresponding operational semantics rules. Here we explain only a few of them that differ significantly from their concrete counterparts. In rule TNew, the ICA for a newly created object is always 1, and this value will be changed later (by rule TW) if this object is carried over to the next iteration. (For objects created outside of loops, the ICA is 0; for brevity, this variation of TNew is not shown in Figure 8.7.) Rule TLoad types variable a with an unknown type $\top$. This handling is over-conservative for the purpose of soundness. Our implementation improves this by consulting a points-to graph that is computed on demand.

[Join of Γ]

(1) $\Gamma_1 \uplus \Gamma_2 = \Gamma_3,\ \text{where } \forall a \in \textsc{Dom}(\Gamma_3),\ \Gamma_3(a) = \begin{cases} \Gamma_1(a) & \text{if } a \in \textsc{Dom}(\Gamma_1) \text{ and } a \notin \textsc{Dom}(\Gamma_2) \\ \Gamma_2(a) & \text{if } a \in \textsc{Dom}(\Gamma_2) \text{ and } a \notin \textsc{Dom}(\Gamma_1) \\ \Gamma_1(a) \uplus \Gamma_2(a) & \text{if } a \in \textsc{Dom}(\Gamma_1) \cap \textsc{Dom}(\Gamma_2) \end{cases}$

(2) $\tilde{\tau_1} \uplus \tilde{\tau_2} = \begin{cases} \tilde{\tau_1} & \text{if } \tilde{\tau_2} = \bot \\ \tilde{\tau_2} & \text{if } \tilde{\tau_1} = \bot \\ (\tilde{\tau_1}.o)^{\tilde{\tau_1}.\tilde{\pi} \uplus \tilde{\tau_2}.\tilde{\pi}} & \text{if } \tilde{\tau_1}.o = \tilde{\tau_2}.o \\ \top & \text{otherwise} \end{cases}$

(3) $\tilde{\pi_1} \uplus \tilde{\pi_2} = \langle \tilde{\pi_1}.l\ ,\ \tilde{\pi_1}.\tilde{i} \uplus \tilde{\pi_2}.\tilde{i} \rangle$

(4) $\tilde{i_1} \uplus \tilde{i_2} = \begin{cases} \tilde{i_1} & \text{if } \tilde{i_1} = \tilde{i_2} \\ \top & \text{otherwise} \end{cases}$

[Operator ⊕]

(5.1) $\tilde{\pi}^j \oplus 1 = \begin{cases} \langle \tilde{\pi}.l\ ,\ \tilde{\pi}.\tilde{i} \oplus 1 \rangle & \text{if } \tilde{\pi}.l = j \\ \tilde{\pi} & \text{otherwise} \end{cases}$

(5.2) $\tilde{i} \oplus 1 = \begin{cases} 1 & \text{if } \tilde{i} = 0 \\ \top & \text{otherwise} \end{cases}$

Figure 8.8: Auxiliary operations.

Type environment join ($\uplus$) needs to be performed in order to handle different control flow paths of an `if-else` statement (in Rule TIf-Else). Joining two environments (rules 1-4 in Figure 8.8) needs to consider both allocation sites and abstract loop iteration counts contained in types. If two types $\tilde{\tau_1}$ and $\tilde{\tau_2}$ do not have the same allocation sites o (rule 2), performing join on them yields $\top$. Otherwise, their loop status abstractions $\tilde{\tau_1}.\tilde{\pi}$ and $\tilde{\tau_2}.\tilde{\pi}$ are forced to join (rule 3). Loop labels ($\tilde{\pi}.l$) in the two status pairs have to be the same because they are associated with the same allocation site. Joining ICAs $\tilde{i_1}$ and $\tilde{i_2}$ is shown in rule 4: if $\tilde{i_1} \neq \tilde{i_2}$, the result is $\top$, meaning that nothing is known about the iteration where the object is created. A finite-height type lattice can be defined based on the operations in Figure 8.8, with $\top$ and $\bot$ as the maximum and minimum types in the lattice. Types with different allocation sites are not comparable.

In the beginning of each loop iteration (shown in rule TW), the ICA of each type (whose allocation site is under loop j) in the type environment is incremented by using operator $\oplus$, which is defined in Figure 8.8 (rule 5). The goal of this is to "clear the loop status" of the objects that are carried over from the last iteration, so that these (old) objects and the fresh objects created in the current iteration can be distinguished. Note that a fixed point is computed for the handling of loops (as discussed further in Section 8.4): while each iteration of the loop may yield a different solution, the fixed-point solution must not be smaller than this solution.

Next, we explain how to detect data structures whose objects are guaranteed to be allocated in the same loop iteration, using the type and effect system.

Lemma 8.2.2 *(*Connected objects created in the same iteration*). For each heap points-to effect $\tilde{\tau}_1 \trianglerighteq^f \tilde{\tau}_2 \in \tilde{H}$, if $\tilde{\tau}_1.\tilde{\pi}.\tilde{\tilde{i}} = \tilde{\tau}_2.\tilde{\pi}.\tilde{\tilde{i}} = 1$ for a particular loop j (i.e., $\tilde{\tau}_1.\tilde{\pi}.l = \tilde{\tau}_2.\tilde{\pi}.l = j$), in each iteration of j where an instance of $\tilde{\tau}_1.o$ and an instance of $\tilde{\tau}_2.o$ are connected by a store operation, these instances must be allocated in this same iteration.*

Proof sketch. Consider a specific iteration k of j. If both objects are allocated in this iteration, their corresponding abstract iteration counts $\tilde{\pi}_1.\tilde{\tilde{i}}$ and $\tilde{\pi}_2.\tilde{\tilde{i}}$ are both updated to 1 upon their creation (rule TNew). In the very beginning of the next iteration $k + 1$, $\tilde{\tau}_1.\tilde{\pi}.\tilde{\tilde{i}}$ and $\tilde{\tau}_2.\tilde{\pi}.\tilde{\tilde{i}}$ will be incremented to $\top$ (rule TW) because these objects are carried over from the last iteration. If in this iteration, both of their allocation sites are executed again, the two ICAs (for the two new instances) are set back to 1 (rule TNew). This process (of setting the ICAs to $\top$ and then 1) is repeated if these allocations are executed during every iteration of j until j terminates. However, if one of the allocation sites (say o_1) is not executed in iteration $k+1$, its corresponding

$$(1)\ \tilde{\tau} \sqsubset \hat{o} \quad \Leftrightarrow \quad \hat{o} = \bot \vee (\hat{o}.o = \tilde{\tau}.o \wedge \tilde{\tau}.\tilde{\pi} \sqsubset \hat{o}.\pi)$$

$$(2)\ \tilde{\pi} \sqsubset \pi \quad \Leftrightarrow \quad \tilde{\pi}.l = \pi.l \wedge \tilde{\pi}.\tilde{i} \sqsubset \pi.i$$

$$(3)\ \tilde{i} \sqsubset i \quad \Leftrightarrow \quad i = \tilde{i} = 0 \vee (i > 0 \wedge \tilde{i} = 1) \vee \tilde{i} = \top$$

$$(4)\ \tilde{H} \sqsubset H \quad \Leftrightarrow \quad \forall(\hat{o_1} \rhd^f \hat{o_2}) \in H : \exists(\tilde{\tau}_1 \unrhd^f \tilde{\tau}_2) \in \tilde{H} : (\tilde{\tau}_1 \sqsubset \hat{o_1}) \wedge (\tilde{\tau}_2 \sqsubset \hat{o_2})$$
$$\wedge((\tilde{\tau}_1.\tilde{\pi}.l = \tilde{\tau}_2.\tilde{\pi}.l \wedge \tilde{\tau}_1.\tilde{\pi}.\tilde{i} = \tilde{\tau}_2.\tilde{\pi}.\tilde{i} = 1) \Rightarrow \hat{o_1}.\pi.i = \hat{o_2}.\pi.i)$$

$$(5)\ \tilde{\Omega} \sqsubset \Omega \quad \Leftrightarrow \quad \forall(\hat{o_1}.f \prec \hat{o_2}.g) \in \Omega : \exists(\tilde{\tau}_1.f \preceq \tilde{\tau}_2.g) \in \tilde{\Omega} : (\tilde{\tau}_1 \sqsubset \hat{o_1}) \wedge (\tilde{\tau}_2 \sqsubset \hat{o_2})$$
$$\wedge((\tilde{\tau}_1.\tilde{\pi}.l = \tilde{\tau}_2.\tilde{\pi}.l \wedge \tilde{\tau}_1.\tilde{\pi}.\tilde{i} = \tilde{\tau}_2.\tilde{\pi}.\tilde{i} = 1) \Rightarrow \hat{o_1}.\pi.i = \hat{o_2}.\pi.i)$$

$$(6)\ (\Gamma, \tilde{\mu}) \sqsubset (\rho, \mu) \quad \Leftrightarrow \quad (\forall a \in \textsc{Dom}(\rho) : \Gamma(a) \sqsubset \rho(a)) \wedge (\forall a \in \textsc{Dom}(\mu) : \forall \hat{o}.f \in \mu(a) : \exists \tilde{\tau}.f \in \tilde{\mu}(a) : \tilde{\tau} \sqsubset \hat{o})$$

Figure 8.9: Abstraction relation $\sqsubset$.

ICA $\tilde{\tau}_1.\tilde{\pi}.\tilde{i}$ will keep the value $\top$. Hence, at the end of iteration $k+1$, $\tilde{\tau}_1.\tilde{\pi}.\tilde{i} = \top$ and $\tilde{\tau}_2.\tilde{\pi}.\tilde{i} = 1$. Because the final solution Γ is a fixed point and $\top$ is greater than any other abstract value, $\top$ will be recorded in Γ for $\tilde{\tau}_1.\tilde{\pi}.\tilde{i}$ even though o_1 may allocate instances again later in the loop.

Note that $\tilde{\tau}_1.\tilde{\pi}.\tilde{i} = \tilde{\tau}_2.\tilde{\pi}.\tilde{i} = 1$ does not necessarily indicate that $\tilde{\tau}_1.o$ and $\tilde{\tau}_2.o$ are executed in *every iteration* of loop j. Their ICAs are 1 as long as their instances cannot escape from the iteration where they are created to the next iteration of the loop. This feature allows the analysis to report potentially-hoistable data structures even though their construction code (i.e., stores that connect objects in them) is guarded by conditionals.

Similarly, given a dependence effect $\tilde{\tau}_1.f \preceq \tilde{\tau}_2.g$, if $\tilde{\tau}_1.\tilde{\pi}.\tilde{i} = \tilde{\tau}_2.\tilde{\pi}.\tilde{i} = 1$ for a loop j, we can safely conclude that this whole dependence (i.e., computation) chain from $\tilde{\tau}_1.f$ to $\tilde{\tau}_2.g$ occurs in the same iteration of j, because there are only stack variables between the two end heap locations of the chain. A detailed description of how the type and effect system abstracts the concrete operational semantics can be found in Figure 8.9. The abstraction relation is denoted by $\sqsubset$.

Given the operational semantics and the type rules, we can prove that the heap points-to and dependence effect abstractions are sound. We show that for each type derivation, the points-to and dependence effect abstractions conservatively approximate the actual points-to and dependence effects generated by the corresponding operational semantics derivation.

Theorem 8.2.3 *(*Effect abstraction soundness*). The effect abstractions are sound if*

$$(e, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \varOmega) \wedge$$
$$(\Gamma, \tilde{\mu} \vdash e : \Gamma', \tilde{\mu}', \tilde{H}, \tilde{\varOmega}) \wedge ((\Gamma, \tilde{\mu}) \sqsubset (\rho, \mu))$$
$$\Rightarrow \tilde{H} \sqsubset H \wedge \tilde{\varOmega} \sqsubset \varOmega \wedge (\Gamma', \tilde{\mu'}) \sqsubset (\rho', \mu')$$

The proof can be done by induction on the derivation, starting at the root and working toward the leaves.

8.2.2 Hoistable Logical Data Structures

In this subsection, we introduce the notion of hoistable logical data structures based on the points-to and dependence effect abstractions computed by the type and effect system. As discussed earlier, here we address the question "what data is hoistable in the best scenario"—that is, to find hoistable logical data structures that meet the two criteria discussed in Section 8.1. Whether and how they can actually be hoisted will be decided by the user upon inspection. This subsection presents mathematical properties of hoistable data structures.

Definition 8.2.4 *(*Constrained closures of $\tilde{H}$ and $\tilde{\Omega}$*) Constrained closures of $\tilde{H}$ and $\tilde{\Omega}$ are represented by relations $\trianglerighteq^*_{j,\tilde{i_1},\tilde{i_2}}$ and $\preceq^*_{j,\tilde{i_1},\tilde{i_2}}$, where parameters j, $\tilde{i}_1$, and $\tilde{i}_2$ denote a loop label, a lower bound, and an upper bound of ICAs, used to compute transitive relationships. We define order $\leq$ on the ICA domain $\tilde{i}$ as $\mathtt{0} \leq \mathtt{1} \leq \top$.*

(1) Closure $\trianglerighteq^*_{j,\tilde{i_1},\tilde{i_2}}$ (on $\tilde{H}$) selects a set of data structures (whose nodes are types), in which each edge has the form $o_1^{\tilde{\pi}_1} \trianglerighteq o_2^{\tilde{\pi}_2} \in \tilde{H}$, s.t. $\tilde{\pi}_1.l = \tilde{\pi}_2.l = j$, $\tilde{i}_1 \leq \tilde{\pi}_1.\tilde{i} \leq \tilde{i}_2$, $\tilde{i}_1 \leq \tilde{\pi}_2.\tilde{i} \leq \tilde{i}_2$.

(2) Similarly, closure $\preceq^*_{j,\tilde{i_1},\tilde{i_2}}$ (on $\tilde{\Omega}$) selects a set of computation chains, in which each edge has the form $o_1^{\tilde{\pi}_1} \preceq o_2^{\tilde{\pi}_2} \in \tilde{\Omega}$, s.t. $\tilde{\pi}_1.l = \tilde{\pi}_2.l = j$, $\tilde{i}_1 \leq \tilde{\pi}_1.\tilde{i} \leq \tilde{i}_2$, $\tilde{i}_1 \leq \tilde{\pi}_2.\tilde{i} \leq \tilde{i}_2$.

Note that constraint $\tilde{i}_1 \leq \tilde{\pi}.\tilde{i} \leq \tilde{i}_2$ is used to compute these closures: $\tilde{\tau}_1 \trianglerighteq^f \tilde{\tau}_2$ (or $\tilde{\tau}_1.f \preceq \tilde{\tau}_2.g$) is added into the closure $\trianglerighteq^*_{j,\tilde{i_1},\tilde{i_2}}$ (or $\preceq^*_{j,\tilde{i_1},\tilde{i_2}}$) only when the ICAs $\tilde{\tau}_1.\tilde{\pi}.\tilde{i}$ and $\tilde{\tau}_2.\tilde{\pi}.\tilde{i}$ are "between" the specified parameters $\tilde{i}_1$ and $\tilde{i}_2$. For example, the general closures $\trianglerighteq^*$ and $\preceq^*$ are special cases of their constrained closures when $\tilde{i}_1 = 0$, $\tilde{i}_2 = \top$, and j is an arbitrary loop label. It is also easy to see that $\trianglerighteq^*_{j,0,0}$ selects data structures whose objects are all created outside loops. Similarly, a data structure selected by $\trianglerighteq^*_{j,0,1}$ is such that its objects can be created both inside and outside loop j, and the set of inside objects in any run-time instance of the data structure are always allocated in the same iteration. Using constrained closures, we give the following definitions.

Definition 8.2.5 *(*Disjoint Data Structure *(DDS)) For an allocation site p located in loop j, a data structure (denoted as δ_p^j) rooted at p with respect to loop j is a graph whose edge set is a subset of $\tilde{H}$. Its run-time instances are guaranteed to be disjoint if, for any edge $\tilde{\tau}_1 \trianglerighteq^f \tilde{\tau}_2$ of the data structure, there exists a type $\tilde{\tau}$ for p, s.t.*

$$\tilde{\tau}.o = p \quad \wedge \quad \tilde{\tau}.\tilde{\pi}.\tilde{i} = 1 \quad \wedge \quad \tilde{\tau} \trianglerighteq^{*}_{j,1,1} \tilde{\tau}_1 \quad \wedge \quad \tilde{\tau} \trianglerighteq^{*}_{j,1,1} \tilde{\tau}_2$$

A DDS contains objects that are reachable from root p and that are created in the loop. Each run-time instance of a DDS is guaranteed *not* to contain any object instance created (1) outside the loop and (2) inside the loop but in different iterations. This is achieved by using constraint (j, 1, 1) for the closure computation.

Lemma 8.2.6 *(*Disjointness of DDS instances*) Given two run-time instances of a DDS δ^l_p created by two iterations of a loop, no run-time object exists in both instances.*

Proof sketch. The lemma can be proven by contradiction. Suppose there is a run-time object that exists in both instances of the data structure. At the point it is added into the second data structure instance (created by a later iteration), the abstract loop iteration count for j contained in its type must be $\top$, which is recorded in the abstract points-to effect that represents this addition. This contradicts the fact that δ^l_p is constructed using closure $\trianglerighteq^{*}_{j,1,1}$, which limits the abstract iteration count for each type to be 1. □

Definition 8.2.7 *(*Iteration-Dependent Field*) A field of the form $\tilde{\tau}.f$ is loop-iteration-dependent (LID) with respect to loop j if*

$$\text{(a) } \exists \tilde{\tau}'.g : \ \tilde{\tau}.f \preceq^{*}_{j,0,\top} \tilde{\tau}'.g \quad \wedge \quad (\tilde{\tau}' = \top \quad \vee \quad \tilde{\tau}'.\tilde{\pi}.\tilde{i} = \top)$$
$$\vee \text{ (b) } \exists \tilde{\tau}'.g : \ \tilde{\tau}'.\tilde{\pi}.\tilde{i} = 0 \quad \wedge \quad \tilde{\tau}.f \preceq^{*}_{j,0,1} \tilde{\tau}'.g \quad \wedge \quad \tilde{\tau}'.g \preceq^{*}_{j,0,1} \tilde{\tau}'.g$$

Determining whether the value of an object field depends on a specific loop iteration requires to inspect abstract dependence effects. As discussed in (condition 2 of)

Section 8.1, a field can be iteration-dependent if (1) it depends on a value read from a field of an unknown object or an object created in an unknown (different) iteration, or (2) it depends on a field of an object created outside the loop (e.g., $\tilde{\tau'}.g$), and this field is involved in a *dependence cycle* (i.e., it can transitively depend on itself).

Lemma 8.2.8 *(*Loop-Invariant Data Structure*) A data structure is loop-invariant if for each type τ in the data structure, $\forall \tilde{\tau}.f \in \textsc{Dom}(\tilde{\Omega}) : \tilde{\tau}.f$ is not an LID field.*

Proof sketch. Let us negate the two conditions in Definition 8.2.7, that is, a loop-invariant field $o.f$ can depend only on (1) fields of objects guaranteed to be created in the same iteration with o, or (2) fields of objects created outside the loop and not involved in dependence cycles. For (1), the proof can be done by induction on the chain of dependence edges leading to $o.f$. In the base case, fields without any incoming dependence edge must be assigned newly created objects or `null`, and thus must be loop-invariant. For the inductive step, consider the n-th edge along the chain. If the source field of the edge is loop-invariant, the target field of the edge must also be loop-invariant.

For (2), let us first consider a simplified situation where there is only one outside object field $p.q$ (i.e., p is created outside the loop) involved in the dependence chain. Here are two subcases. First, field $o.f$ (o is an object created inside the loop) depends on $p.q$ and $p.q$ is never written in the loop. It is straightforward to see that $p.q$ does not carry any iteration-specific values and thus $o.f$ is loop-independent.

Second, suppose field $p.q$ is written in iteration i with a value v produced in iteration i'. Here i must equal i', because otherwise $o.f$ could depend on a value computed in a different iteration, which contradicts the statement in (1). Since value v cannot depend on $p.q$ (otherwise $p.q$ would depend on itself), it must come only from objects freshly created in this iteration. Based on the proof of case (1), we know that v must be loop-invariant. If $p.q$ is read later in iteration $k > i$ to produce another value v', v' must also be the same across iterations because $p.q$ is invariant. This reasoning can be easily generalized to handle the more complex situation where multiple outside object fields exist in the dependence chain. □

Theorem 8.2.9 *(*Hoistable Logical Data Structure *(HDS)) If a data structure δ_p^j is (1) disjoint and (2) loop-invariant, it can exhibit exactly the same behavior at run time when it is located inside the loop and outside the loop, under the assumption that the code statements that access this data structure can be safely hoisted.*

Note that the section introduces the notion of hoistable logical data structure only for formal development. In fact, instead of reporting only completely-hoistable data structures, our analysis identifies, for each logical data structure, its hoistable part (that is both disjoint and loop-invariant). The analysis eventually ranks all loop data structures based on their hoistability measurements, in order to quantify the likelihood of successful manual hoisting.

8.3 Computing Hoistability Measurements

In practice, we have observed that only a small number of data structures and statements in a large program can meet both of the hoisting criteria described in Section 8.2. This is primarily due to their complex usage and the conservative nature of any static analysis algorithm, which must model this complex usage soundly. While fully-automated transformations are desirable and sometimes possible, the usefulness of the static analyses can be increased even further by generalizing them to provide valuable support for programmer-driven manual code transformations.

Previous studies such as [156,159] have demonstrated that, in many cases, manual performance tuning with developers' insight can be much more effective than fully-automated compiler optimizations. For instance, a programmer may quickly identify that it is problematic to create a 100-field data structure in a loop with only 1 field being iteration-dependent, while the sound transformation would give up and terminate silently. To enlist human's effort, we must present to them highly-relevant information that can quickly direct them to a problematic area. In this section, we present two metrics that we use to measure *hoistability* of data structures. These measurements are computed based on the points-to and dependence relationships described earlier in this chapter.

Dependence-based hoistability (DH) The first metric we consider measures the amount of data in a data structure that is constant during the execution of a loop (i.e., the part that complies with rule 2 in Section 8.1). This dependence hoistability metric is simply defined as an exponential function $DH = s^{n/s}$, which considers two factors: the total number of fields s in a data structure and the number of its loop-invariant (i.e., non-LID, discussed in Def. 8.2.7) fields n. The larger s is, the more performance improvement could potentially be achieved by hoisting it. The larger n/s is, the easier it is for a programmer to hoist this data structure. If s is 1, the data structure contains a single field. Even though this field is invariant (i.e., n/s is 1), hoisting it may not have a large impact on performance. If n/s is a small number (i.e., most of its fields are not invariant), the result of $s^{n/s}$ can be very small (i.e., close to 1) regardless of how large s is, which also indicates it is not worth spending time as the data structure may be too difficult to hoist. In addition, we choose an exponential function instead of a polynomial function as the metric because the exponential function "penalizes" cases where n is small, while a polynomial function would be "fair" for all cases of n. For example, if $n{=}s/2$ (half the fields are invariant), our exponential function will give the square root of s, while a polynomial function may give a much larger number.

Structure-based hoistability (SH) Similarly to the first metric, the second metric considers, for each data structure, how many objects in it are guaranteed to be allocated in the same iteration (i.e., the part that complies with condition 1 in Section 8.1). This structural hoistability metric is defined as $SH = t^{m/t}$, where t is the total number of objects in the data structure and m is the number of objects whose ICA is 1. SH considers both the size of the data structure and the size of its disjoint part. Note that when m/t is 1 for a data structure, this data structure is a DDS (as discussed in Def. 8.2.5), as it is guaranteed to have disjoint instances in all loop iterations.

During our studies, we found that DH is much more useful than SH in distinguishing data structures that are easy to hoist manually from those that are not. First, s is a more accurate measurement of the size of a data structure than t, as the data structure can still be large if it contains fewer objects but each object has more fields. Second, we found that for a large number of data structures in our benchmarks, their m/t is 1, which means they are all DDS. It would be quite labor-intensive to inspect all of them and check if they are hoistable. To help the programmer quickly identify truly optimizable data structures, we focus on DDS (whose m/t is 1) and compute dependence-based hoistability (DH) only for these data structures. Finally, only DDS are ranked (based on their DH measurements) and presented to the user for inspection.

Incorporating dynamic information For performance tuning, dynamic frequency information is usually necessary to help programmers focus on "hot" entities (e.g., calling contexts, data structures, etc.). For example, it could be more beneficial to hoist a small, but frequently-allocated data structure than a big, occasionally-occurring data structure. Frequency information can be easily incorporated into the two hoistability metrics. For example, for DH, its definition can be simply extended to $DDH = (f * s)^{n/s}$, where f represents the allocation frequency of the root object of the data structure.

8.4 Implementation

The static analysis implementation has five logical components. The first component is a *data structure analysis*. In order to discover the data structure rooted at an object, this analysis employs a variation of the context-free-language (CFL)-reachability formulation of points-to analysis [133], which models both context sensitivity via method entries and exits, and heap accesses via object fields read and writes. This analysis was proposed in Chapter 7 [159] to understand container structures and semantics. While this part (data structure analysis) is the adaption of an existing technique, all other components are completely new and are novel contributions.

For each loop in an actual Java program, data structure root objects are first located. To find such root objects, we first consider objects that are created directly in the loop body. Objects that are created in a method (e.g., used as a factory) invoked by a call site in the loop and that can be returned to the loop-containing method are also considered.

Next, for each root object o, our analysis identifies the set of reachable objects and their points-to relationships. In particular, the analysis looks for chains of stores of the form $a_0.f_0 = \mathit{new}\ X; a_1.f_1 = b_0; a_2.f_2 = b_1; \ldots; a_n.f_n = b_{n-1}; b_n = o$, such that (1) the two variables in each pair (a_i, b_i) for $0 \leq i \leq n$ are aliases and (2) the CFL path for this chain contains balanced method entries and exits. If such a chain can be found, the object represented by $\mathit{new}\ X$ is in the data structure rooted at o, because it could potentially be reached from o at run time through a sequence of field dereferences $f_n.f_{n-1} \ldots f_1.f_0$. Using this formulation, our hoisting analysis can be performed on demand: it can work directly on each loop object, and performs only the work necessary to detect its data structure and to check its hoistability.

The second part of the implementation is a form of *data flow analysis* that analyzes each loop to perform type inference. An abstract heap (points-to and data dependence) effect is the join of data flow facts on all valid paths from the loop entry to the assignment that connects the two entities in the effect.

The third part is a form of *data dependence analysis* that detects loop-invariant object fields. This analysis traverses backward the def-use chains from each store that writes to a field of an object in a loop data structure, and checks whether the two conditions in Definition 8.2.7 hold for the field. Similarly to other existing static slicing algorithms, a key challenge to computing precise data dependences lies in the handling of data flow via heap locations. Our analysis initially works on top of a context-insensitive points-to analysis: for each load $a = b.f$, we find the set of all assignments $c.f = d$ such that c and d can alias context-insensitively. We next

Benchmark	(a) Analysis statistics						(b) Dependence-based hoistability			
	#M(K)	*#L*	*#Obj*	*Time (s)*	*#DDS*	*#HDS*	*#SF*	*#SIF*	*#DF*	*#DIF*
jack	12.5	88	13	1224	5	3	797	62	797	62
javac	13.4	270	89	1745	33	8	45	31	42	28
soot-c	10.4	475	17	3043	7	3	56	36	56	36
sablecc-j	21.4	202	228	7910	82	53	429	194	221	61
jess	12.8	119	32	304	7	1	1135	51	1135	51
muffin	21.4	318	96	10766	47	8	1503	198	-	-
jflex	20.2	209	17	2325	9	0	55	17	55	17
jlex	8.2	108	9	5549	4	0	36	6	36	6
java-cup	8.4	99	19	474	4	0	107	57	107	57
antlr	12.9	154	3	77	2	1	3	0	3	0
bloat	10.8	562	141	3476	36	10	1536	136	674	46
chart	17.4	482	102	12746	6	0	84	19	84	19
xalan	12.8	17	8	63	6	0	78	24	78	24
hsqldb	12.5	33	10	178	5	0	75	19	75	19
luindex	10.7	14	5	163	5	0	65	15	65	15
ps	13.5	117	21	1784	21	11	36	20	34	20
pmd	15.3	594	30	168	15	2	127	68	127	68
jython	27.5	614	48	423	24	3	77	25	190	26
eclipse	41.0	3322	93	21557	80	52	1182	180	-	-

Table 8.1: Shown in the first seven columns of the table is the general statistics of the benchmarks and the analysis: the benchmark names, the numbers of methods (in thousands) in Soot's Spark context-insensitive call graph (M), the numbers of loops inspected (L), the numbers of loop objects considered (Obj), the running times of the analysis ($Time$), the total numbers of disjoint data structures (DDS), and the total numbers of hoistable logical data structures (HDS). Columns *SF* and *SIF* in section "Dependence-based hoistability" show the total numbers of fields (SF) and the numbers of loop-invariant fields (SIF), averaged among the top 10 DDS that we chose to inspect. These data structures are ranked based on the dependence-based hoistability measurement (DH). Columns *DF* and *DIF* report the same measurements as *SF* and *SIF*, except that the inspected data structures are ranked using DDH that incorporates dynamic frequency information.

perform refinement on this candidate set using the CFL-reachability formulation of pointer-aliasing to find whether b and c can indeed alias, and if they can, the calling context for $c.f = d$ under which the value flow occurs. This calling context is used to guide the future graph traversal to follow the appropriate entry/exit edges.

8.5 Evaluation

We implemented the analysis in the Soot analysis framework [132, 147] and evaluated it on the 19 Java programs shown in Table 8.1. All experiments were conducted on a dual-core machine with an Intel Xeon 2.83GHZ processor, running Linux 2.6.18. The Java library used for the analysis was Sun JDK 1.5.0_06, and 4GB of max heap space was specified to run the analysis. Programs in the table were from the `SpecJVM98`, `Ashes`, and `DaCapo` benchmark suites. We were able to run the analysis on all the programs in these sets, but we included only applications that have objects created in loops.

8.5.1 Static Analysis and Hoisting

Table 8.1 reports statistics of the benchmarks, the analysis, and the dependence-based hoistability measurements. For a GUI application `muffin`, we could not find an appropriate test case to perform profiling, and thus, "-" is used to fill out columns that report dynamic-information-based measurements. We could not perform profiling for `eclipse` either, as different plugins use their own class loaders, making it difficult for them to access our profiling library without modifying their customized class loaders. The cost of the analysis is generally proportional to the number of loop objects processed because of its demand-driven nature. The analysis running time can also be influenced by the size of the code base, as the analysis is context-sensitive and the number of contexts often grows significantly when the size of the program increases. It is clear that the analysis can scale to large applications, including the `eclipse` framework, which has millions lines of code in its implementation.

Across all applications we observe large numbers of disjoint data structures (DDS) and hoistable logical data structures (HDS). This is a strong indication of the existence of optimization opportunities that can be exploited by human experts, which motivates our proposal of computing hoistability measurements to help manual tuning. To compare the effectiveness of *DH* (dependence-based hoistability measurement proposed in Section 8.3) and *DDH* (the profile-based version of it) in finding optimization opportunities, we inspected the top 10 data structures (or the total number of data structures if it is smaller than 10) that appear in both reports. The total numbers of fields divided by the numbers of loop-invariant fields for these inspected data structures are shown in columns *SF/SIF* and *DF/DIF*, for these two kinds of reports. Profiling is implemented by Soot-based bytecode instrumentation that records the execution frequency for each loop. The goal of this comparison is to understand how much impact the dynamic information can have on the interpretation of reports. Specifically, can the use of run-time frequency f in *DDH* lower the ranks of data structures that are highly-likely to be optimized (i.e., have larger n/s but smaller f)?

The ratio between *SIF* (or *DIF*) and *SF* (or *DF*) indicates, to a large degree, the difficulty of hoisting data structures manually by inspecting the analysis reports. The larger it is, the easier it may be for a performance expert to modify the data models to hoist them. It is clear that the ratios of DIF/DF are generally close to those of SIF/SF. In some cases, the former are even greater than the latter. This observation indicates that *DDH* can expose not only hot spots (i.e., frequently-allocated objects), but also *optimizable* data structures.

8.5.2 Case Studies

We have carefully inspected the generated analysis reports (with dynamic information incorporated) for these 19 benchmarks and found optimizable data structures in almost every one of them. This subsection presents five representative case studies, in which either large performance improvement was seen, or interesting bloat patterns were found. These applications are `ps`, `xalan`, `bloat`, `soot-c`, and `sablecc-j`, all with large code base and a great number of loop objects. The performance problems we show in this chapter are new and have never been reported by any previous work.

Through these studies, we found that the analysis is quite useful in helping programmers find mostly-loop-invariant data structures and the execution inefficiencies due to these data structures. It took us about three days to find and fix problems in these five applications, among which we had studied only `bloat` before. Note that most of this time was spent on developing fixes rather than finding data structures that can be easily hoisted: for each benchmark, we looked at only the top 10 data structures in the reports (due to the limited time we had), and found that most of them were indeed hoistable. Even larger optimization opportunities could have been possible if we had inspected more warnings generated by the tool.

It is important to note that it would not be possible to find such problems by using any existing profiling tool: to detect loop-invariant data structures, a purely dynamic analysis has to perform whole-program *value profiling*, a task that is prohibitively expensive for large-scale, long-running Java programs. This is the reason why we have not compared our results with dynamic analysis reports.

ps `ps` is a postscript interpreter. The top data structure in the list ranked by *DDH* is rooted at a `NameObject` object created in a loop in method `execute` of class `makeDictOperatorDB`. The loop is used to traverse a stack: for each stack element (i.e., a `NameObject` object containing a key and a value), the goal is to remove the '/' character in the key of the element. The way the loop is implemented is that it creates another (backup) stack, pops the original stack, creates a new `NameObject` object with most values copied directly from the original object, and pushes this new object onto the backup stack. Eventually all the new objects in the backup stack are pushed back onto the original stack. The creation of such `NameObject` objects directs us to think about this implementation, and especially about the way the stack is operated. In fact, this process can be done entirely *in place* so that these objects do not even need to be created. A further inspection of code found an even more interesting problem. The programmer seems to ignore the fact that class `Stack` is a subclass of `List` in JDK and uses `push` and `pop` to implement everything related to stack. For example, this same pop-push pattern is used even for element retrieval. For almost each occurrence of this problematic stack usage pattern, there is a corresponding (mostly loop-invariant) data structure in our report. We removed only two occurrences of such a pattern (in `makeDictOperatorDB.execute` and `DictStack.getValueOf`), and this resulted in a reduction of 82.1% in running time (from 28.3s to 5.3s) and 70.8% reduction in the total number of objects created (from 10170142 to 2969354).

xalan `xalan` is an XLST processor for XML documents. The problem we found is in a test harness (`XalanHarness`) used by DaCapo to run the benchmark. This harness class creates multiple threads to transform input XML files. In method `run`, there is a `while(true)` loop that assigns jobs to different threads. Our report shows that a data structure rooted at a `Transformer` object created in the loop is a HDS (shown in Figure 8.1(b)). The same *Transformer* object is created every time the loop iterates, and then used by different threads for transforming the input files. It is highly unlikely that it would be possible to automatically hoist this data structure because this object is created by using a transformer factory object, which is obtained from a reflective call. After hoisting this allocation site, we observed a 10% reduction in running time and 1% reduction in the number of objects created. This problem has also been confirmed by the DaCapo maintainers [41] and will be handled in the next release of the DaCapo benchmark set.

bloat `bloat` is a program analysis tool designed for Java bytecode-level optimizations. Many loop data structures reported by our analysis are objects of inner classes that are declared exactly at the point where their objects are needed. In `bloat`, most of these objects are created to implement visitor patterns. Hence, the objects are used only for method dispatch and do not contain any data related to the program context under which they are created. These objects commonly exist in loops, and in many cases we found them even located in loops with many layers of nesting. This problem exemplifies a typical object-oriented philosophy: the programmer should focus on patterns and abstractions when coding, and leave the mess to the runtime system. By simply hoisting the reported objects (and the declarations of their classes) out of the loops, we saw 11.1% running time reduction and 18.2% reduction in the number of created objects.

soot-c `soot-c` is a part of the Soot analysis framework [132, 147] benchmarked in the Ashes benchmarks suite [10]. One top data structure reported by our analysis is rooted at a `StmtValueBoxPair` object created in a loop (in a constructor of `jimple.SimpleLocalUse`) that builds def-use relationships as follows:

```
while(defIt.hasNext()){
    List useList = (List) stmtToUses.get(defIt.next());
    useList.add(new StmtValueBoxPair(s, useBox));
}
```

For each statement *s* that uses a variable, the program finds a set of statements that define the variable, creates a `StmtValueBoxPair` object, and adds it to the list. These `StmtValueBoxPair` objects, while containing the same values, are created for safety purposes: if one such pair is changed later, other pairs should not be affected. Our tool cannot hoist this data structure, because it is not confined (i.e., escapes to fields of objects created outside the loop). However, after inspecting the code, we found that the use list associated with each statement is never changed after the jimple statement chain is constructed for a program. Even if a client analysis could change it by inserting statements, Soot always creates a new object to represent this (newly-established) def-use relationship rather than change the original object. This problem shows a typical example of an over-protective implementation, where several different mechanisms are used simultaneously to enforce the same property while one (or a few) of them may be sufficient to do so. By sharing one `StmtValueBoxPair` object among multiple def statements, we achieved 2.5% running time reduction and 3.5% reduction in the number of created objects. In this example, we can see once again the advantage of tool-assisted manual tuning: this data structure can never be eligible for hoisting from the perspective of any fully-automated analysis. However, the human insight did make hoisting happen as it is unnecessary to have these instances simultaneously.

sablecc-j `sablecc-j` is a version of the Sable Compiler Compiler that produces the sablecc files (parser, lexer, etc.) for a preliminary version of the jimple grammar. Similarly to the problems found for `bloat`, a large number of HDS reported are related to inner classes: two such classes are declared in `sablecc.GenParser` to perform depth-first traversal of syntax trees, and one such class is declared in `sablecc.DFA` to represent an interval in a char set. Creating multiple objects for each such class is completely unnecessary. Hoisting these class declarations and their objects resulted in 6.7% running time reduction and 2.5% reduction in the number of objects created.

Evaluation summary While all loop-invariant data should be hoisted out of loops, the tight data coupling in an object-oriented program makes it impossible for us to do so (either automatically or manually). To help programmers focus on data structures that are (1) easy to hoist and (2) worth hoisting, we propose to compute hoistability measurements. Through these case studies, we demonstrate that our measurements are effective in pinpointing such data structures. In fact, by inspecting reported data structures, we found many performance problems and achieved significant performance improvement. Some invariant data structures that we have managed to hoist are due to (deeper) design issues such as inefficient implementations of design patterns (e.g., visitors in `bloat`) or over-protective implementation strategies (e.g., `soot-c`). Our measurements were also helpful in revealing these issues by exposing their symptoms (i.e., mostly-invariant data structures).

8.6 Summary and Interpretation

This chapter presents a static technique that detects loop-invariant data structures. We focus on data models and look for logical data structures that can be hoisted. Instead of transforming the program and hoisting data structures automatically, we propose to measure the hoistability of a data structure: the dependence-based hoistability metric measures the amount of loop-invariant data in a data structure, and the structure-based hoistability metric measures the size of its disjoint part. We have implemented the analyses and presented an evaluation of the technique on a set of 19 Java benchmarks. Our experimental results demonstrate that the analysis can scale to large applications and the measurements can be useful in finding large optimization opportunities.

Motivated by the same insight, the two static analyses presented in this chapter and the last chapter target two specific bloat patterns that we frequently observed in the execution of large-scale real-world applications. The success of these techniques shows that by identifying concrete patterns of inefficiencies, a static approach can precisely detect potential performance problems before these problems are observed during run-time execution. With the help of lightweight dynamic information, it is even possible for the static analysis to detect performance bottlenecks that are particularly hard for a dynamic analysis to detect (e.g., to find loop-invariant data structures, a dynamic analysis needs to perform expensive value profiling). It will be an interesting future topic to investigate more effective combinations of static and dynamic information to help programmers understand significant performance issues.

CHAPTER 9: Related Work

9.1 Dynamic Analysis

This section outlines work related to the dynamic analyses presented in this dissertation. The relevant existing work falls into the following five categories: bloat detection, dynamic slicing, dynamic data flow tracking, dynamic memory leak detection, and heap assertions.

9.1.1 Bloat Detection

Dufour *et al.* propose dynamic metrics for Java [48], which provide insights by quantifying runtime bloat. Many memory profiling tools have been developed to take heap snapshots for understanding memory usage (e.g., [69]) and to identify objects of suspicious types that consume a large amount of memory (e.g., [51, 109]). However, none of these tools attempt to understand the underlying causes of memory bloat, and thus cannot help programmers pinpoint the problematic areas of the application.

Mitchell *et al.* [97] structure behavior according to the flow of information, though using a manual technique. Their aim is to allow programmers to place judgments on whether certain classes of computations are excessive. Our copy profiling work is in this same spirit, and automates an important component of this approach. Their follow-up work [96] introduces a way to find data structures that consume excessive amounts of memory. Work by Dufour *et al.* finds excessive use of temporary data structures [49, 50] and summarizes the shape of these structures. In contrast to the purely dynamic approximation introduced in our work, they employ a blended escape analysis, which applies static analysis to a region of dynamically collected calling structure with observed performance problem. By approximating object effective lifetimes, the analysis has been shown to be useful in classifying the usage of newly created objects in the problematic program region. JOLT [124] is a VM-based tool that uses a new metric to quantify *object churn* and identify regions that make heavy use of temporary objects, in order to guide aggressive method inlining.

Our dynamic approaches differ from all existing bloat detection work in two dimensions. First, our work addresses the challenge of automatically detecting bloated computations that fall out of the purview of conventional JIT optimization strategies. In general, existing bloat detection work can be classified into two major categories: manual tuning methods (i.e., mostly based on measurements of bloat) [49, 50, 96, 97], and fully automated performance optimization techniques such as the entire field of JIT technology [8] and the research from [124]. The work described in this dissertation sits in between: we provide sophisticated analyses to support manual tuning, guiding programmers to the program regions where bloat is most likely to exist, and then allowing human experts to perform the code modification and refactoring. By doing so, we hope to help the programmers quickly get through the hardest part of the tuning process—finding the likely bloated regions—and yet use their (human) insights to perform application-specific optimizations.

Second, we use different (non-conventional) symptom definitions to identify the bloated regions. For example, the work of copy profiling profiles data flows based on the observation that bloat often manifests itself in the form of large volumes of copies. On the contrary, the JIT performs optimizations based on hot spots, which are decided completely by profiling control flows. As shown in Chapter 1, performance bottlenecks do not necessarily exist in frequently-executed regions, and in many cases, they are more related to data flow, rather than control flow.

A significant difference between the cost-benefit analysis and existing bloat detection techniques is that an existing approach can usually find only one type of problems effectively. For instance, blended escape analysis [49, 50] is effective at detection of temporary objects while a container profiling technique [122, 158] works only for container bloat. Our cost-benefit analysis detects operations that have high costs and low benefits. Performing such operations is the essence of bloat and is a common characteristic of a variety of performance problems, which, however, may show different symptoms on the surface. Hence, the cost-benefit analysis is potentially capable of identifying many different kinds of bloat, and thus can be more useful in practice to help a programmer perform the tuning task.

9.1.2 Control- and Data-Based Profiling

Lossy compression of profiles has been proposed for space efficiency. These techniques include dynamic dependence profiles [3], control flow profiles [14], and value profiles [30]. While lossy compression can provide sufficient precision for many applications, evidence has been shown that they are inadequate for many others. Lossless compression techniques are thus developed to reduce space requirements and yet preserve the dynamically collected data. Research from [79, 170] studies the compressed representations of control flow traces. Value predictors [28] are proposed to compress value profiles, which can be used to perform various kinds of tasks such as code specialization [30], data compression [171], value encoding [163] and value speculation [85]. Research from [35] proposes a technique to compress an address profile, which is used to help prefetch data [67] and to find cache conscious data layouts [119]. Zhang and Gupta propose *whole execution traces* [167] that include complete data information of an execution, to enable the mining of behavior that requires understanding of relationships among various profiles.

Ammons *et al.* [7] develops a dynamic analysis tool to explore calling context trees in order to find performance bottlenecks. Srinivas *et al.* [136] use a dynamic analysis technique that identifies important program components, also by inspecting calling context trees. Chameleon [122] is a dynamic analysis tool that profiles container behaviors to provide advice as to the choices of appropriate containers. The work in [110] proposes object ownership profiling to detect memory leaks in Java programs.

When profiling to find performance problems, existing techniques typically concentrate on control flow, rather than data flow, from path profiling [14, 19, 79, 148] to feedback-directed profiling [8], all to identify heavily-executed paths for further optimization. The copy profiling technique described in Chapter 3 profiles data flow, and uses the copy profiles to determine the problematic program regions. The profiling technique in the cost-benefit analysis is similar in spirit to the dependence profiling in [3]. While both fall in the general category of lossy profile compression, our technique proposes to introduce analysis semantics into profiling. Hence, our approach is lossless in terms of the target analysis—as long as a target analysis can be formulated in our framework, the compressed profile provides all the information required by that analysis. The summarization techniques described in [3] are analysis-neutral and it is unclear what kinds of analyses can take advantage of them.

9.1.3 Dynamic Slicing

Since first being proposed by Korel and Laski [76], dynamic slicing has inspired a large body of work on efficiently computing slices and on applications to a variety of software engineering tasks. A general description of slicing technology and challenges can be found in Tip's survey [146] and Krinke's thesis [77]. The work by Zhang *et al.* [165–169] has considerably improved the state of the art in dynamic slicing. This work includes, for example, a set of cost-effective dynamic slicing algorithms [166,168], a slice-pruning analysis that computes confidence values for instructions to select those that are most related to errors [165], a technique that performs online compression of the execution trace [167], and an event-based approach that reduces the cost by focusing on *events* instead of individual instruction instances [169]. We refer the reader to Zhang's thesis [164] for a detailed description of these techniques. Sridharan *et al.* proposes thin slicing [134], a technique that improves the relevance of the slice by focusing on the statements that compute and copy a value to the seed. Although this technique is originally proposed for static analysis, it fits naturally in the work on dynamic cost-benefit analyses.

Our work on abstract dynamic slicing is fundamentally different from these existing techniques in the following ways. Orthogonal to the existing profile summarization techniques such as [3, 14, 30, 167], abstract slicing achieves efficiency by introducing analysis semantics to profiling, establishing a foundation for solving a range of dynamic data flow problems. If an analysis can be formulated in our framework, the profiled information is sufficiently precise for this particular analysis. Hence, although our approach falls into the general category of lossy compression, it is lossless for the specific analysis formulated. The work from [169] is more related to our work in that the proposed event-based slicing approach is a special case of abstract slicing with the domain $\mathcal{D}$ containing a set of pre-defined events. In addition, existing work on dynamic slicing targets its use for automated program debugging, whereas the goal of our work is to understand performance and to find bottlenecks.

9.1.4 Dynamic Information Flow Analysis

Dynamic taint analysis [38, 59, 103, 108, 162] tracks input data from untrusted channels to detect potential security attacks. Debugging, testing, and program understanding tools track dynamic data flow for other specialized purposes (e.g., [87]). The work in [24] tracks the origins of undefined values to assist debugging. Research from [88] proposes to measure the strength of information flows and conducts an empirical study to better understand dynamic information flow analysis. Work from [31, 32, 101] describes approaches to enforcing information flow analysis in Java virtual machines.

Our dynamic analyses combine information flow tracking and profiling to efficiently form execution representations (e.g., graph G_{cost}) that are necessary for the client analyses. Because information flow analysis is expensive in general, approaches such as [108] have been developed to reduce its run-time cost. These techniques can also be employed in the future to make our techniques more scalable.

9.1.5 Dynamic Memory Leak Detection

Dynamic analysis [20,43,44,51,61,62,71,80,109] has typically been the "weapon of choice" for detecting memory leaks in real-world Java software. However, as described in Chapter 5, existing techniques have a number of deficiencies. The work in [43,44,51,109] enables visualization of objects of different types on the heap, but does not provide the ability to directly identify the cause of the memory leak. Existing techniques use growing types [71,95] (i.e., types whose number of instances continues to grow) or object staleness [20] to identify suspicious data structures that may contribute to a memory leak. However, in general, a memory leak caused by redundant references is due to a complex interplay of memory growth, staleness, and possibly other factors. By considering a single metric which combines both factors, our technique could potentially improve the precision of leak identification. In addition, all existing dynamic-analysis-based leak detection approaches start by considering the leak symptoms (e.g., growing types or stale objects), and then attempt to trace back to the root cause of the leak. As discussed in the description of the JDK bugs from Chapter 5, the complexity of such bottom-up tracking makes it hard to generate precise analysis reports, and ultimately puts a significant burden on the programmer.

In contrast, the approach described in Chapter 5 is designed from a container-centric point of view—it automatically tracks the suspicious behavior in a top-down manner, by monitoring (1) the object graph reachable from a container, and (2) the container-level operations. Our second approach described in Chapter 6 solves these problems by allowing programmers to explicitly express their interests (i.e., related to high-level semantics), based on the insight that developers' knowledge is essential for a leak detector to produce highly relevant reports. These higher levels of abstraction (e.g., container-centric view, transactions, etc.), compared to traditional low-level tracking of arbitrary objects, simplifies the difficult task of identifying the sources of memory leaks.

9.1.6 Heap Assertions

Closest to our LeakChaser work (described in Chapter 6) are heap property assertion frameworks such as GC Assertions [1, 112] and QVM [9, 149]. For example, one can explicitly specify at a certain program point that an object should be dead soon (*assertDead*), or that an object must be owned by another object in the object graph (*assertOwns*). While such assertions can be quite useful in helping diagnosis, they are limited in the following three important aspects related to leak detection.

First, reachability information is used to approximate the liveness of objects, which may result in false positives. For example, *assertOwns* (a, b) asserts a reachability relationship between objects a and b, and it fails when b can be reached along a path that does not contain a in a certain GC run. However, it is possible that object b becomes unreachable from object a in one GC, while later b is owned by a again. Second, because a GC assertion predicts a future heap state (i.e., the state at the closest GC run) and the global reachability information evolves all the time, whether or not this assertion will fail depends significantly on when and where the next GC occurs, which may in turn be affected by many factors, such as the initial and maximum heap sizes, and specific GC implementation strategies. Third, these approaches are intended for programmers who have sufficiently deep program knowledge and insights. In real-world software development, only a handful of programmers can have such knowledge, especially when a performance problem occurs in program code that is not written by themselves.

Our work solves the first and second problems by allowing programmers to specify object lifetime relationships instead of reachability properties. In order to tackle the third problem, we use a combination of assertion checking and transaction property inference to allow programmers with little application-specific knowledge to quickly identify the cause of the problem.

The GC Assertions [1] framework includes a block-structured *assert-alldead* assertion, which asserts that all objects allocated in the block must be dead by the end of the block. While it is related to the transaction abstractions proposed in Chapter 6, there are two important distinctions between them. First, *assert-alldead* does not allow objects in the specified structure to escape the structure, while our approach allows checking and inferring shared objects, providing more flexibility for diagnosing problems. Second, our transaction abstraction separates temporal and spatial scopes of the structure, while these scopes are combined in this earlier work.

Merlin [65] is an efficient algorithm that can provide precise time-of-death statistics for heap objects by computing when objects die using collected timestamps. LeakChaser could potentially exploit this technique in the future to capture assertion failures between GCs, as we currently report an assertion failure only when it is actually witnessed during a GC run.

9.2 Static Analysis

This section outlines static analysis work related to the techniques presented in this dissertation. The related work falls into four major categories: static memory leak detection, CFL-reachability-based analyses, object-reachability-analysis, and compiler optimization techniques that transform programs for performance improvement.

9.2.1 Static Memory Leak Detection

Static analysis can find memory errors such as double frees and missing frees for programs written in non-garbage-collected languages. For example, [34] reduces the memory leak analysis to a reachability problem on the program's guarded value flow graph, and detects leaks by identifying value flows from the source (malloc) to the sink (free). Saturn [154], taking another perspective, reduces the problem of memory leak detection to a boolean satisfiability problem, and uses a SAT-solver to identify potential bugs. Dor *et al.* [47] propose a shape analysis based on 3-valued logic, to prove the absence of memory leaks in several list manipulation functions. Hackett and Rugina [58] use a shape analysis that tracks single heap cells to identify memory leaks. Orlovich and Rugina [106] propose an approach that starts by assuming the presence of errors, and performs a backward dataflow analysis to disprove their feasibility. Clouseau [63] is a leak detection tool that uses pointer ownership to describe variables responsible for freeing heap cells, and formulates the analysis as an ownership constraint system. Its follow-up work [64] proposes a type system to describe the object ownership for polymorphic containers and uses type inference to detect constraint violations. Although both this work and our technique focus on containers, the target of this previous effort is C and C++ programs whereas we are interested in a garbage-collected language. The analysis described in [64] does not help detect unnecessary references in a Java program.

More generally, all static approaches are limited by the lack of general, scalable, and precise reference/heap modeling. Despite a large body of work on such modeling, it remains an open problem for analysis of large real-world systems, with many challenges due to analysis scalability, modeling of multi-threaded behavior, dynamic class loading, reflection, etc. By profiling container behavior and finding memory leaks based on container-related heuristics, our dynamic memory leak detection work complements the existing static leak detection approaches, and can help a programmer quickly find the cause of a memory leak in large Java applications.

9.2.2 CFL-Reachability-Based Analyses

It is well known that method calls and returns can be treated as pairs of balanced parentheses using a context-free language [66,113,115–117]. Sridharan *et al.* propose a CFL-reachability formulation to precisely model heap accesses and calling contexts for computing a points-to solution for Java [133]. As an extension of this formulation, Zheng and Rugina [172] propose a CFL-reachability formulation for C/C++ alias analysis. Our previous work [160] proposes to use an offline *must-not-alias* analysis to reduce the amount of work that needs to be performed by the CFL-reachability computation to speed up the actual points-to analysis. CFL reachability can also be used to implement polymorphic flow analysis [111] and shape analysis [114]. The work in [75,90] studies the connection between CFL-reachability and set constraints, shows the similarity between the two problems, and provides new implementation strategies for problems that can be formulated as CFL-reachability and set constraints. While based on the Sridharan-Bodik CFL-reachability formulation of points-to analysis, our static analysis of container operation detection takes into account container-specific structures, and thus, can potentially have a wide range of applications that need to reason about container behavior, including summary generation for containers for more scalable static analyses, static container-based memory leak detection [63], and other techniques (e.g., thin slicing [134]) that need to track the flow of container elements and ignore all other objects constituting container structures.

9.2.3 Object Reachability Analysis

The work closest to the static analysis described in Chapter 7 is the disjoint reachability analysis proposed by Naik and Aiken [100] for eliminating false positives in their Java data race detector. This analysis is also flow-insensitive and takes into account loop information to distinguish instances created by the same allocation site. Unlike this analysis that uses object-sensitivity to compute reachability information, we employ a CFL-reachability formulation that is capable of filtering out information irrelevant to container objects, and thus, our analysis may scale to larger programs. Other reachability analysis algorithms range from flow-sensitive approximations of heap shape (e.g., [33, 121]) to decision procedures (e.g., [81, 89]). While our analysis is less precise in discovering the shape of data structures, it is more scalable and has been shown to be effective in detecting container problems.

There exists a large body of work on ownership types and their inference algorithms [5, 26, 37, 54, 63, 86]. Ownership types provide a way of specifying object encapsulation, and enable local reasoning about program correctness in object-oriented languages. Existing ownership type inference algorithms may not be able to provide precise container information, because containers are usually designed to have polymorphic object ownership—some objects of a container type may own their elements while others may not. In addition, containers are complicated by the fact that they can be nested: the elements in a container may themselves be containers. As a pointer to a container is passed around, ownership of the container may transfer from one pointer to another. While the work from [64] proposes an abstract object ownership model specifically for containers, it requires the tool users to specify correct interfaces for container implementation routines, which are then used in the ownership inference algorithm. Our approach is completely automated, and does not require any user annotations for detecting problems with the built-in Java collections.

9.2.4 Recency Abstraction

The loop iteration count abstraction used in Chapter 8 is inspired by the early work from [13, 100]. This abstraction is first proposed in [100] for computing their conditional must-not-alias properties. Our abstraction extends this abstraction to allow objects guarded by conditionals to be selected for detecting hoistable data structures in real-world Java programs. Work from [13] presents *recency abstraction*, a technique that distinguishes most-recently-allocated-object (MRAO) and non-MRAO for each allocation site in order to enable strong updates for a points-to analysis. While this is similar to our iteration abstraction that distinguishes objects created in the current iteration and previous iterations, our analysis uses such an abstraction for identifying loop-invariant data structures, instead of improving the precision of a points-to analysis.

9.2.5 Compiler Optimizations for Performance Improvement

Pioneered by the commercial Smalltalk implementation, the Just-In-Time compiler [8, 11, 45] offers many sophisticated intraprocedural static analyses that can effectively improve the performance of an application by finding hotspots and performing optimizations at these hotspots. However, JITs can miss many optimization opportunities due to the lack of hotspots in large applications and the limitations of method inlining, is necessary for the majority of optimization techniques. This dissertation advocates new approaches that develop automated program analyses to "simulate" (the more powerful versions) of the JIT dataflow analyses that could not be done in a typical JIT compiler due to the natural limitations of run-time compilation. These analyses can help a programmer quickly find missed opportunities that are easy to understand and fix.

Object inlining [46,84] is a static technique that consists of finding sets of objects that can be efficiently fused into larger objects, and fusing them. While both object inlining and our hoisting analysis (Chapter 8) concern performance problems and need to find objects created in the same control flow region, we target a different class of performance problems. In addition, our analysis can assist a programmer to do manual tuning, a task that is difficult for object inlining to perform.

In the literature of compiler optimization [4], loop optimization is an important technique that improves performance by exploiting parallelism and data locality. There is a large body of work that has been devoted to making the execution of loops faster. This set of techniques includes, for example, loop interchange, loop splitting, loop unrolling, loop fusion, loop-invariant code motion, etc.

In high-performance computing, loop optimizations play a key role in automated parallelization for exploiting potential parallelism. We refer the reader to the survey by Bacon *et al.* [12], Wolfe's book [152], and Kennedy and Allen's book [72] for a broader overview and more detailed description of these techniques.

As a more general technique, partial redundancy elimination (PRE) (e.g., [18, 27, 36, 74, 98]) performs common subexpression elimination and loop-invariant code motion at the same time. It is powerful in removing redundancy and has been used widely in optimizing compilers. As a step beyond these traditional loop optimization techniques, our work aims to find optimization opportunities by hoisting complex heap data structures that can be constructed by multiple instructions and across many method invocations, which traditional techniques cannot handle.

CHAPTER 10: Conclusions

Software applications are now assembled from many abstractions, and programmers trust compilers to avoid low-level tuning of the implementation and composition of these abstractions, in the hope that automated optimizations will take care of those details. As a result, questionable decisions are often made—for example, the use of an overly-general library to a achieve a simple task, or the addition of yet another layer of delegation in the data model. The cost of one additional method call or one more allocated object seems insignificant. In reality, the effects of these decisions can accumulate, and the underlying compilers and runtime systems cannot eliminate these inefficiencies.

There are two key observations on which our work relies. First, runtime bloat is primarily a by-product of object-orientation, whose culture encourages a certain level of excess. Programmers are taught to follow standard principles (e.g., freely create objects, favor reuse and generality, etc.) and leave performance to compilers and runtime systems. Many inefficiencies in the execution are due to designs and development guided by such principles, and thus, these is much high-level semantic information (expressing developers' intent) associated with these inefficiencies. It can be extremely difficult to develop a fully automatic optimization technique that can effectively remove these inefficiencies. We believe that understanding such semantic information and bringing it into tool design are necessary to develop useful performance analysis tools that can help programmers identify large optimization opportunities. Our second key observation is that code that contributes to runtime bloat can be identified by focusing not on the flow of control, but rather on the activities of data. Dynamic and static analyses that track the creation and usage of object-oriented data structures are at the heart of our work.

Based on these observations, this dissertation first presents a copy profiling technique that identifies program regions containing large volumes of data copying. However, different types of bloat may manifest themselves through different observable symptoms, including not only data copies, but also other signs such as temporary objects [50, 124], highly-stale objects [20, 105], and inappropriate collection choices [122, 158]. A common effect of these different performance problems is an imbalance between costs and benefits: the cost of forming a data structure, of computing and storing its values, is too high compared to the benefits gained over the uses of those values. The second analysis proposed by the dissertation is to profile costs and benefits, and detect performance bottlenecks by locating operations that have unbalanced cost-benefit rates. The underlying analysis technique of abstract thin slicing serves as the foundation for the cost-benefit computation as well as for a number of other dynamic analyses that require backward traversals of the execution history. These dynamic cost-benefit analyses have been implemented in J9, IBM's commercial Java Virtual Machine, and have been shown to be effective in helping a programmer quickly find problematic code that needs to be further inspected and optimized.

Misuse of containers is an important source of runtime bloat. This dissertation presents novel algorithms targeting detection of container bloat. A special type of bloat that has severe impact on performance is a memory leak, caused by keeping references to objects that are no longer used. We propose a dynamic analysis that identifies container-induced memory leaks by profiling container behaviors. Concrete container functionality is abstracted into three basic operations *ADD*, *GET*, and *REMOVE*. Using a novel leak confidence model that considers a combination of container staleness and container memory consumption, the tool can precisely pinpoint the containers that are not appropriately used (i.e., the causes of memory leaks). In this work, these operations are manually annotated. Once the interface of a container is modified, these annotations have to be modified accordingly. This may create a considerable amount of work for a tool user. The profiling would be easier if the process of container modeling could be (even partially) automated. A static analysis technique that can automate this process is proposed in Chapter 7. At the core of this analysis is a CFL-reachability formulation of container behavior. We also propose two options for employing these automatically-detected operations, one that instruments them and detects bloat by profiling their frequencies, and a second one that approximates their frequencies using the nesting of the loops where these operations are located. We show in Chapter 7 that (1) the static tool is useful in finding inefficiencies during coding, and (2) later, for performance tuning that requires precise identification of hot spots, large optimization opportunities can be found by statically analyzing hot containers identified through dynamic information.

Many memory leaks are caused not by the misuse of containers, but rather by cached references that the programmer forgot to invalidate. To help programmers quickly diagnose such general memory leak problems, we propose a specification-based technique, called LeakChaser (Chapter 6), that can not only capture precisely the unnecessary references leading to leaks, but can also explain, with high-level application semantics, why these references become unnecessary.

Another bloat pattern that can be frequently seen is the creation and initialization of loop-invariant data structures. In Chapter 8 we propose a type and effect system that can help programmers find and hoist such data structures. By hoisting loop-invariant data structures early during the development, the programmer is likely to prevent small performance issues from accumulating, and thus simplify significantly the task of subsequent performance tuning.

We have presented a series of case studies to show the effectiveness of our dynamic and static bloat detection analyses. With varying space and time overheads, these techniques are designed to target different types of performance problems, and can be used in different phases of software development. For example, the cost-benefit analysis and the copy profiling analysis can assist a programmer to make appropriate design/implementation choices related to modeling and design. The static container bloat detection tool can be used during coding to find (unnoticed) inefficiently-used containers that could potentially lead to bloat. Similarly, the static detection of loop-invariant data structures can be employed during coding to identify likely performance problems. All proposed tools can help performance diagnosis when a problem is observed during tuning.

We hope that with the help of the techniques we have developed, performance tuning could be made much easier and will no longer be a daunting task that requires special skills and experience. Developers should be able to easily understand performance and perform optimizations, when they are assisted by good tools and do not need to focus on every low-level detail of the execution behavior and the analysis process. The productivity-performance gap between managed languages and unmanaged languages could be further reduced by using these techniques and tools so that performance would no longer be an issue that stands in the way of using object-oriented languages to implement performance-critical systems. Furthermore, we hope that the examples and patterns discovered by this dissertation can be used to raise awareness of bloat in real-world software development. Developers should understand the performance impact of their implementation choices and should try to avoid these bloat patterns in order to have high-performance implementations.

CHAPTER 11: Future Work

Over the course of 17 years from 1986 to 2002, the performance of microprocessors improved at the rate of 52% per year. What grows even faster than the clock speed is the functionality and size of software (a.k.a. Myhrvold's Law). Myhrvold's premise that "software is a gas" describes the phenomenon that no matter how much improvement has been achieved on hardware, developers always have the tendency to add functionality to make their software push the performance boundaries. There is an ever-increasing demand for performance optimization in modern software despite the deployment of faster CPUs and larger memory systems. Future work will continually address this demand by inventing new approaches to alleviate bloat across the entire software lifecycle: new language features, type systems, design models, and testing and analysis tools. It is also interesting to consider how to adapt this chain of techniques to the development of performance-critical software, such as real-time and mobile applications, to improve their functionality while reducing the required human labor. We believe there are larger opportunities than ever before that we, as programming language and software engineering researchers, can exploit in order to make software more efficient, and this can happen entirely at the application level, without the help of ever-increasing hardware capabilities.

This chapter describes future research opportunities, with a focus on both our own future directions and what the PL/SE community can do to address the ever-increasing levels of inefficiency in object-oriented applications.

Thin patterns While design patterns [53] have been extremely successful in helping programmers write highly-manageable code, they are the root causes of many types of runtime bloat. In many large applications, for instance, in order to implement a visitor pattern, the programmer uses inner classes to represent different kinds of visitors, which contain nothing but a `visit` method. Such a visitor class can be instantiated an extremely large number of times (e.g., the allocation sites are located in loops with many layers of nesting), and all objects created are identical: they have no data and are used only for dynamic dispatch. It is *not* free to create and deallocate these objects, and significant overhead reduction can be seen when we use only the method without creating objects.

Future research on patterns may consider the creation of a few specialized versions for each existing pattern (i.e., *thin patterns*), which provide different tradeoffs between inefficiency and modularity. On the compiler side, pattern-aware optimization techniques could be expected to remove inefficiencies and generate higher-quality code.

Performance-conscious modeling languages and tools While performance-aware design has been studied in the field of software performance engineering (e.g., [126, 153]), this research focuses primarily on high-level architectures and processes, rather than low-level program inefficiencies. Hence, the problem is worth re-considering in the future, and additional efforts should be focused on explicit bloat avoidance in the state-of-art modeling languages (e.g., UML) and tools (e.g., EMF and Rational Software Modeler).

Careless design can lead to significant runtime bloat, especially when modeling tools are used to automatically generate code skeletons from the design. As an example, one cause of bloat are carelessly chosen associations. Consider several classes X_1, X_2, ... together with the associations between them (e.g., as defined in object-oriented design and captured in UML class diagrams). The associations typically include directionality (uni- vs. bi-directional) and multiplicity (e.g., one-to-many). There are often many semantically-equivalent ways to implement these associations in code. The programmer may choose one of these possibilities without truly understanding the implications of her/his choice on the memory footprint of the application. Even worse, in many cases, the programmer does not make this choice at all—the default data model in the modeling tool is applied automatically behind the scenes. A performance-conscious design model will take performance requirements as an explicit parameter, and this will result in extended modeling languages and tools that incorporate various resource constraints.

Unit testing/checking bloat It is important to avoid inefficiencies early during development, before they accumulate and become observable. This calls for novel program analysis and testing techniques that can work for incomplete programs. While there exists a body of work on unit testing and component-level analysis, it is unclear how to adapt them to verify non-functional properties. For example, it may not be easy to write assertions (i.e., test oracles) for unit testing, as redundancy at the unit level may not be obvious and thus the assertions are likely to be insensitive to explicit performance checks (e.g., running time and space).

This difficulty actually points to the more general problem of non-functional specifications. What can we assert about performance other than running time and space?

Can any functional properties of a program be employed to specify performance requirements? Good specifications must be closely related to a certain bloat problem, and not simply describe the symptom that the problem exhibits. Significant improvements could be achieved in the research of performance analysis if such specifications were designed and evaluated.

Autonomous system and program synthesis Looking a bit far into the future, the feedback-directed compilation techniques in a JVM may be powerful enough to handle the many layers of abstractions [93] during the execution. For example, dynamic object inlining may be an effective approach to reduce pointer overhead. In order to remove container inefficiency, the runtime system could automatically shrink the space allocated for the container if it observes that much of the space is not used. These technique of course require sophisticated profiling techniques that are semantics-aware and incur sufficiently low overhead. Advances in program synthesis [127–131, 137] shed new light on solving the execution bloat problem. Given a user-defined specification, a program synthesis tool can automatically choose, from a space of algorithms, the most efficient one. This can apply naturally in the research of bloat detection to find efficient implementations, and may further be used to generate implementations for performance-critical tasks that are guaranteed to meet performance requirements.

Data-based profiling Existing profiling work focuses primarily on control-based profiling (e.g., path profiling, method profiling, trace profiling), under the assumption that substantial performance gains can be obtained by optimizing frequently-executed control-flow regions. However, we found that in many cases, high frequency of certain data-manipulating activities (e.g., pure copies) is even a stronger indicator of optimization opportunities than that of control-flow regions. Are there any other bloat-indicating data activities? Can we incorporate such profiling work into a modern feedback-directed compiler to improve application performance? This is an entire area that is almost untouched by existing work, and presents exciting opportunities for future developments.

BIBLIOGRAPHY

[1] E. E. Aftandilian and S. Z. Guyer. GC Assertions: Using the garbage collector to check heap properties. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 235–244, 2009.

[2] *Agile Software Development.* `http://www.agilealliance.com`.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 246–256, 1990.

[4] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[5] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 311–330, 2002.

[6] E. Altman, M. Arnold, R. Bordewekar, R. Delmonico, N. Mitchell, and P. Sweeney. Observations on tuning a Java enterprise application for performance and scalability. *IBM Journal of Research and Development*, 54(5):1–12, 2010.

[7] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 172–196, 2004.

[8] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 92(2):449–466, 2005.

[9] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 143–162, 2008.

[10] *Ashes Suite Collection*. `http://www.sable.mcgill.ca/software`.

[11] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[12] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[13] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *Static Analysis Symposium (SAS)*, pages 221–239, 2006.

[14] T. Ball and J. Larus. Efficient path profiling. In *International Symposium on Microarchitecture (MICRO)*, pages 46–57, 1996.

[15] 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, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[16] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 22–32, 2008.

[17] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, 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. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, 2008.

[18] R. Bodík, R. Gupta, and M. L. Soffa. Complete removal of redundant expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–14, 1998.

[19] M. D. Bond and K. S. McKinley. Continuous path and edge profiling. In *International Symposium on Microarchitecture (MICRO)*, pages 130–140, 2005.

[20] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 61–72, 2006.

[21] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 97–112, 2007.

[22] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 109–126, 2008.

[23] M. D. Bond and K. S. McKinley. Leak pruning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–288, 2009.

[24] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 405–422, 2007.

[25] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[26] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 213–223, 2003.

[27] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 159–170, 1994.

[28] M. Burtscher and M. Jeeradit. Compressing extended program traces using value predictors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 159–169, 2003.

[29] C. Calcagno, D. Distefano, P. OHearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 289–300, 2009.

[30] B. Calder, P. Feller, and A. Eustace. Value profiling. In *International Symposium on Microarchitecture (MICRO)*, pages 259–269, 1997.

[31] D. Chandra. *Information Flow Analysis and Enforcement in Java Bytecode*. PhD thesis, University of California, Irvine, 2006.

[32] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *Annual Computer Security Applications Conference (ACSAC)*, pages 463–475, 2007.

[33] B. Chang and X. Rival. Relational inductive shape analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 247–260, 2008.

[34] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 480–491, 2007.

[35] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 191–202, 2001.

[36] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 273–286, 1997.

[37] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310, 2002.

[38] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 196–206, 2007.

[39] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[40] *DaCapo Benchmarks.* `http://www.dacapo-bench.org`.

[41] *DaCapo Bug Repository. Bloat Report.* `http://sourceforge.net/tracker/?func=detail&aid=2975679&group_id=172498&atid=861957`.

[42] A. Deepak, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies.* Prentice Hall, 2003.

[43] W. DePauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 219–234, 1998.

[44] W. DePauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. *Concurrency: Practice and Experience*, 12(14):1431–1454, 2000.

[45] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 297–302, 1984.

[46] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 345–357, 2000.

[47] N. Dor, M. Rodeh, and S. Sagiv. Checking cleanness in linked lists. In *Static Analysis Symposium (SAS)*, pages 115–134, 2000.

[48] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 149–168, 2003.

[49] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 118–128, 2007.

[50] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 59–70, 2008.

[51] ej-technologies GmbH. *JProfiler*. `http://www.ej-technologies.com`.

[52] *Extreme Programming*. `http://www.extremeprogramming.org`.

[53] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[54] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 241–255, 2001.

[55] S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 375–385, 2009.

[56] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 127–139, 2009.

[57] S. C. Gupta and R. Palanki. *Java memory leaks – catch me if you can*, 2005. http://www.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki.

[58] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 310–323, 2005.

[59] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Annual Computer Security Applications Conference (ACSAC)*, pages 303–311, 2005.

[60] W. Halfond, A. Orso, and P. Manolios. WASP: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 34(1):65–81, 2008.

[61] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Winter 1992 USENIX Conference*, pages 125–138, 1992.

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

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

[64] D. L. Heine and M. S. Lam. Static detection of leaks in polymorphic containers. In *International Conference on Software Engineering (ICSE)*, pages 252–261, 2006.

[65] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Generating object lifetime traces with Merlin. *ACM Transactions on Programming Languages and Systems*, 28(3):476–516, 2006.

[66] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 104–115, 1995.

[67] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *International Symposium on Microarchitecture (MICRO)*, pages 14–23, 1997.

[68] *Java Development Blog*, 2009. cld.blog-city.com.

[69] *Java Heap Analyzer Tool (HAT).* `http://hat.dev.java.net`.

[70] *Jikes Research Virtual Machine.* `http://jikesrvm.org`.

[71] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 31–38, 2007.

[72] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., 2002.

[73] B. Kent. *Implementation Patterns.* Addison-Wesley, 2007.

[74] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.

[75] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 207–218, 2004.

[76] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[77] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs.* PhD thesis, University of Passau, 2003.

[78] C. Larman. *Applying UML and Patterns.* Prentice Hall, 2nd edition, 2002.

[79] J. Larus. Whole program paths. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 259–269, 1999.

[80] *LeakHunter.* `http://www.wilytech.com/solutions/products/LeakHunter.html`.

[81] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, and S. Srivastava. Simulating reachability using first-order logic with applications to verification of linked data structures. In *International Conference on Automated Deduction (CADE)*, pages 99–115, 2005.

[82] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction (CC)*, LNCS 2622, pages 153–169, 2003.

[83] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction (CC)*, pages 153–169, 2003.

[84] O. Lhoták and L. Hendren. Run-time evaluation of opportunities for object inlining in Java. *Concurrency and Computation: Practice & Experience*, 17(5-6):515–537, 2005.

[85] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *International Symposium on Microarchitecture (MICRO)*, pages 226–237, 1996.

[86] Y. Liu and A. Milanova. Ownership and immutability inference for UML-based object access control. In *International Conference on Software Engineering (ICSE)*, pages 323–332, 2007.

[87] W. Masri and A. Podgurski. An empirical study of the strength of information flows in programs. In *International Workshop on Dynamic Analysis (WODA)*, pages 73–80, 2006.

[88] W. Masri and A. Podgurski. Measuring the strength of information flows in programs. *ACM Transactions on Software Engineering and Methodology*, 19(2):1–33, 2009.

[89] S. McPeak and G. Necula. Data structure specifications via local equality axioms. In *International Conference on Computer Aided Verification (CAV)*, pages 476–490, 2005.

[90] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248:29–98, 2000.

[91] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.

[92] N. Mitchell. The runtime structure of object ownership. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 74–98, 2006.

[93] N. Mitchell. The big pileup. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 1–1, 2010.

[94] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1):56–63, 2010.

[95] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 351–377, 2003.

[96] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 245–260, 2007.

[97] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 429–451, 2006.

[98] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[99] V. Nagarajan and R. Gupta. Architectural support for shadow memory in multiprocessors. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 1–10, 2009.

[100] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 327–338, 2007.

[101] S. K. Nair, P. N. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, 2008.

[102] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 65–74, 2007.

[103] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Annual Network & Distributed System Security Symposium (NDSS)*, 2005.

[104] E. Nicholas. `http://weblogs.java.net/blog/enicholas/archive/2006/04/leaking_evil.html`.

[105] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 397–407, 2009.

[106] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *Static Analysis Symposium (SAS)*, pages 405–424, 2006.

[107] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 291–302, 2005.

[108] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *International Symposium on Microarchitecture (MICRO)*, pages 135–148, 2006.

[109] Quest Software. *JProbe Memory Debugging*. `http://www.quest.com/jprobe`.

[110] D. Rayside and L. Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *International Conference on Automated Software Engineering (ASE)*, pages 194–203, 2007.

[111] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 54–66, 2001.

[112] C. Reichenbach, N. Immerman, Y. Smaragdakis, E. Aftandilian, and S. Z. Guyer. What can the GC compute efficiently? A language for heap assertions at GC time. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 256–269, 2010.

[113] T. Reps. Solving demand versions of interprocedural analysis problems. In *International Conference on Compiler Construction (CC)*, pages 389–403, 1994.

[114] T. Reps. Shape analysis as a generalized path problem. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 1–11, 1995.

[115] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

[116] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 49–61, 1995.

[117] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 11–20, 1994.

[118] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, 2004.

[119] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 140–153, 2002.

[120] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[121] 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, 1999.

[122] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 408–418, 2009.

[123] R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in Java. In *International Conference on Compiler Construction (CC)*, pages 50–66, 2000.

[124] A. Shankar, M. Arnold, and R. Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 127–142, 2008.

[125] O. Shivers. Control-flow analysis in Scheme. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 164–174, 1988.

[126] C. Smith and L. Williams. Software performance engineering: A case study including performance comparison with design alternatives. *IEEE Transactions on Software Engineering*, 19:720–741, 1993.

[127] A. Solar-Lezama. *Program Synthesis by Sketching.* PhD thesis, UC Berkeley, 2008.

[128] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 167–178, 2007.

[129] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 136–148, 2008.

[130] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 281–294, 2005.

[131] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. A. Seshia. Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, 2006.

[132] *Soot Analysis Framework*. `http://www.sable.mcgill.ca/soot`.

[133] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 387–400, 2006.

[134] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 112–122, 2007.

[135] M. Sridharan, D. Gopan, L. Shan, and R. Bodik. Demand-driven points-to analysis for Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 59–76, 2005.

[136] K. Srinivas and H. Srinivasan. Summarizing application performance from a component perspective. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 136–145, 2005.

[137] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 313–326, 2010.

[138] *SPECjbb2000*. `http://www.spec.org/jbb2000`.

[139] *SPECjvm98*. `http://www.spec.org/jvm98`.

[140] Z. Su and D. Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computer Science*, 345(1):122–138, 2005.

[141] *Sun Bug Database*. `http://bugs.sun.com/bugdatabase`.

[142] *Sun Java Forum*. `http://forums.java.net/jive/thread.jspa?messageID=180784`.

[143] *The Common Language Runtime*. `http://msdn.microsoft.com/en-us/library/ddk909ch(VS.71).aspx`.

[144] *The J9 Java Virtual Machine*. `http://wiki.eclipse.org/J9`.

[145] *The Java Virtual Machine Tool Interface*. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`.

[146] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[147] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction (CC)*, pages 18–34, 2000.

[148] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: Compactly numbering interesting paths. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 351–362, 2007.

[149] M. Vechev, E. Yahav, and G. Yorsh. PHALANX: Parallel checking of expressive heap assertions. In *International Symposium on Memory Management (ISMM)*, pages 41–50, 2010.

[150] C. Wang and A. Roychoudhury. Dynamic slicing on Java bytecode traces. *ACM Transactions on Programming Languages and Systems*, 30(2):1–49, 2008.

[151] J. L. Whitten, L. D. Bentley, and K. C. Dittman. *Systems Analysis and Design Methods*. McGraw-Hill, 2005.

[152] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[153] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *Future of Software Engineering (FOSE)*, pages 171–187, 2007.

[154] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 115–125, 2005.

[155] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 174–186, 2010.

[156] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 419–430, 2009.

[157] G. Xu, M. D. Bond, F. Qin, and A. Rountev. Leakchaser: Helping programmers narrow down causes of memory leaks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–282, 2011.

[158] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *International Conference on Software Engineering (ICSE)*, pages 151–160, 2008.

[159] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 160–173, 2010.

[160] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 98–122, 2009.

[161] G. Xu, D. Yan, and A. Rountev. Finding loop-invariant data structures. In *Technical Report OSU-CISRC-8/11-TR24, CSE/OSU*, 2011.

[162] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, pages 121–136, 2006.

[163] J. Yang and R. Gupta. Frequent value locality and its applications. *ACM Transactions on Programming Languages and Systems*, 1(1):79–105, 2002.

[164] X. Zhang. *Fault Localization via Precise Dynamic Slicing.* PhD thesis, University of Arizona, 2006.

[165] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 169–180, 2006.

[166] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 94–106, 2004.

[167] X. Zhang and R. Gupta. Whole execution traces. In *International Symposium on Microarchitecture (MICRO)*, pages 105–116, 2004.

[168] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *International Conference on Software Engineering (ICSE)*, pages 319–329, 2003.

[169] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 81–91, 2006.

[170] Y. Zhang and R. Gupta. Timestamped whole program path representation and its applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 180–190, 2001.

[171] Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. In *International Conference on Compiler Construction (CC)*, pages 14–28, 2002.

[172] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 197–208, 2008.