Scalable Runtime Bloat Detection Using Abstract Dynamic Slicing

GUOQING XU, University of California, Irvine
NICK MITCHELL and MATTHEW ARNOLD, IBM T.J. Watson Research Center
ATANAS ROUNTEV, Ohio State University
EDITH SCHONBERG and GARY SEVITSKY, IBM T.J. Watson Research Center

Many large-scale Java applications suffer from runtime bloat. They execute large volumes of methods and create many temporary objects, all to execute relatively simple operations. There are large opportunities for performance optimizations in these applications, but most are being missed by existing optimization and tooling technology. While JIT optimizations struggle for a few percent improvement, performance experts analyze deployed applications and regularly find gains of 2× or more. Finding such big gains is difficult, for both humans and compilers, because of the diffuse nature of runtime bloat. Time is spread thinly across calling contexts, making it difficult to judge how to improve performance. Our experience shows that, in order to identify large performance bottlenecks in a program, it is more important to understand its dynamic dataflow than traditional performance metrics, such as running time.

This article presents a general framework for designing and implementing scalable analysis algorithms to find causes of bloat in Java programs. At the heart of this framework is a generalized form of runtime dependence graph computed by *abstract dynamic slicing*, a semantics-aware technique that achieves high scalability by performing dynamic slicing over bounded abstract domains. The framework is instantiated to create two independent dynamic analyses, *copy profiling* and *cost-benefit analysis*, that help programmers identify performance bottlenecks by identifying, respectively, high-volume copy activities and data structures that have high construction cost but low benefit for the forward execution.

We have successfully applied these analyses to large-scale and long-running Java applications. We show that both analyses are effective at detecting inefficient operations that can be optimized for better performance. We also demonstrate that the general framework is flexible enough to be instantiated for dynamic analyses in a variety of application domains.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program analysis*; D.3.4 [**Programming Languages**]: Processors—*Memory management, optimization, run-time environments*

General Terms: Languages, Measurement, Performance

Additional Key Words and Phrases: Runtime bloat, dynamic analysis, abstract dynamic slicing, copy profiling, cost-benefit analysis

---

This article extends and refines work from two previous conference papers by the authors, published in the *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* in 2009 and 2010.
This material is based on work supported by the National Science Foundation under grants CCF-0546040, CCF-1017204, CCF-1319695, and CNS-1321179, and by an IBM Software Quality Innovation Faculty Award. G. Xu was additionally supported by an IBM Ph.D. fellowship award while he was a graduate student.
Authors' addresses: G. Xu (corresponding author), Department of Computer Science, University of California, Irvine, CA; email: guoqingx@ics.uci.edu; N. Mitchell and M. Arnold, IBM T.J. Watson Research Center, Yorktown Heights, NY; A. Rountev, Department of Computer Science and Engineering, Ohio State University, Columbus, OH; E. Schonberg and G. Sevitsky, IBM T.J. Watson Research Center, Yorktown Heights, NY.

DOI: http://dx.doi.org/10.1145/2560047

ACM Reference Format:
Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2014. Scalable runtime bloat detection using abstract dynamic slicing. ACM Trans. Softw. Eng. Methodol. 23, 3, Article 23 (May 2014), 50 pages.
DOI: http://dx.doi.org/10.1145/2560047

1. INTRODUCTION

A major sustained technology trend 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, to large-scale long-running servers that can process requests for many millions of concurrent users. Many achievements in the successful development of complex software are due to the community-wide recognition of the importance 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. 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 users of a reusable component, 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 these APIs can provide, wasteful operations start emerging. For example, a study from Mitchell [2006] 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.

In this article, the term *bloat* [Mitchell et al. 2010; Xu et al. 2010b; Xu 2011] 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. Removing bloat is especially relevant for multicore systems: 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.

While modern JITs have sophisticated optimizers that offer important performance improvements, 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 I 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% 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 method inliner to combine together code into larger, hopefully optimizable, regions. Forming perfect code regions and then optimizing them is an immensely challenging problem [Shankar et al. 2008]. 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., nonescaping objects). In addition, optimizations that can be easily performed by a developer (e.g., moving an invocation of a side-effect-free method out of a loop) can require great 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

Table I. In a Commercial Document Management Server, No Single Frequently Executed Method Can Be Optimized for Easy Performance Gains

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%

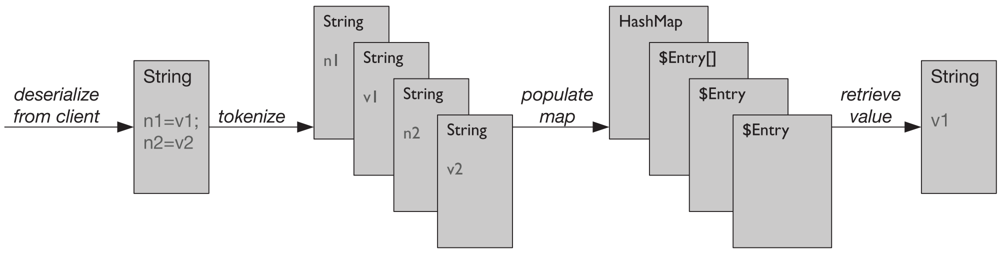
powerful analyses to work together. If language features that restrict optimization (e.g., reflection and precise exceptions) are taken into account, there is little hope that a performance bottleneck can be identified and removed by a fully automated solution. As an example, a study described later found that, to perform the seemingly simple task of inserting a single small document in the database, a document server invokes 25,000 methods and creates 3,000 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. In this article, we present two dynamic analysis techniques that can pinpoint problematic areas to help developers quickly find and fix performance problems in large-scale real-world applications.

1.1. Copy Profiling

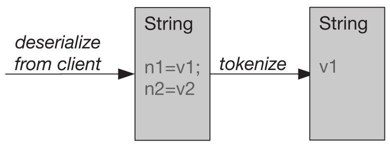
One important symptom of runtime bloat is a large volume of copies in bloated regions: data is transferred from one object to another, with no computation done on it. Long copy chains can often be seen, simply to form objects of certain types required by APIs of one framework from their original representations used in another framework. For instance, the inefficiencies in the SOAP example are closely related to copies (due to data wrapping and unwrapping).

If the SOAP example exhibits design issues, the following example that can be observed in the IBM 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 case, the caller extracts one or two elements from the 8-element map, and never uses that map again. Figure 1 illustrates the steps necessary to decode a cookie in this application. Decoding a single cookie, an operation that occurs repeatedly, costs 1,000 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 four invocations and constructs two 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 [Mitchell et al. 2006], as visualized in Figure 1. Bloat of



(a) original version



(b) specialized version

Fig. 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.

this form manifests itself in a number of ways: temporary structures to carry values, 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 1 eliminates the cost of constructing the `HashMap` and the related key-value pairs.

As the first major contribution of this work, we propose a novel dynamic technique, called *copy profiling*, to help programmers find copy-related inefficiencies. 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. To help programmers quickly find copy-induced performance problems, we propose a series of techniques that profile various kinds of copy activities, including flat copy summaries, copy chains, and copy graph. We have also built three client analyses based on copy profiles that can effectively expose optimization opportunities in large programs. These analyses are presented in Section 3.5.

1.2. Cost-Benefit Analysis

Bloat can 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). In an example mentioned earlier, in a large Java program we found that the programmer creates lists and adds many elements to them, only for the purpose of obtaining list sizes. Most fields in these list data structures do not have any benefits for the forward progress of the application. 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 trade-offs 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. As the second major contribution of this work, we propose a dynamic cost-benefit analysis that can provide automated support for performance experts to measure costs and benefits at a fine-grained (instruction) level, thereby improving the precision of the answers and making tuning an easier task.

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. Such questions can be answered efficiently by the proposed cost-benefit analysis. The detailed analysis algorithm can be found in Section 4.

1.3. Abstract Dynamic Slicing as a General Framework

At the heart of these two techniques is a dynamic dataflow tracking engine that keeps track of how data is propagated among memory locations. In the article, we develop a framework to generalize this dataflow tracking analysis so that the framework can be instantiated to implement other dynamic (bloat or bug) detection techniques. To provide sufficient dataflow information for a client, this framework needs to record the whole program execution trace. This can be achieved by performing a dynamic slicing algorithm [Korel and Laski 1990; Zhang et al. 2003; Zhang and Gupta 2004a; Wang and Roychoudhury 2008], which tracks the execution of each instruction and each memory address it accesses. In a large program, however, an instruction can be executed an extremely large number of times; the amount of memory needed for whole-program dynamic slicing is thus unbounded and is determined completely by the runtime behavior of the program. As a result, regular dynamic slicing is prohibitively expensive for large-scale and long-running applications.

To enable the analysis of large-scale, long-running applications, we introduce abstract dynamic slicing, a technique that applies dynamic slicing over an abstract domain whose size is limited by bounds independent of the runtime execution. This technique is embedded in the general framework parameterized by the abstract domain. The output of this framework is an abstract dependence graph that contains abstractions of instructions, rather than their actual runtime instances. This new approach is motivated by the observation that a client of dynamic slicing often needs to access only a small portion of the complete execution trace collected by a regular slicing algorithm and thus tremendous effort is wasted on collecting information not used by the client. The runtime (space and time) overhead can be significantly reduced if the slicing algorithm is client aware, that is, it understands what information would be needed by its client and records only such information during the execution. Abstract dynamic slicing makes this possible by asking the analysis developer to provide an abstraction that specifies this knowledge. In the article, we first define this framework (in Section 2) and then show how the framework is instantiated to derive the algorithms of copy profiling and the cost-benefit analysis (in Section 3 and Section 4).

The major contributions of this work are as follows.

—*Abstract dynamic slicing* is a general technique that performs dynamic slicing over bounded abstract domains. It produces much smaller and more relevant slices than traditional dynamic slicing and can be instantiated to implement a variety of dynamic dataflow analyses.
—*Copy profiling* is an instance of the abstract dynamic slicing framework that identifies high-overhead activities in terms of copies and chains of copies.
—*Cost and benefit profiling* is the second instance of the framework that identifies runtime inefficiencies by understanding the cost of producing values and the benefit of consuming them.
—*An implementation* of the framework and the two client analyses based on the IBM J9 commercial Java Virtual Machine demonstrates that: (1) the general framework can be easily instantiated and (2) the client analyses are useful in finding various opportunities for performance improvements.
—*A set of experimental results* shows that these two techniques can be used to help developers quickly find performance problems in large applications such as `tomcat` and `derby`; significant performance improvement has been observed after these problems are fixed.

2. ABSTRACT DYNAMIC SLICING

This section describes the general framework that uses abstract dynamic slicing to provide dataflow information for client analyses implemented with the help of the framework. We demonstrate its usage through several real-world examples. While our implementation works on the low-level JVM intermediate representation, the discussion of the framework and the related algorithms uses a three-address code representation of the program. In this representation, each statement corresponds to a bytecode instruction (i.e., it is either a copy assignment `a=b` or a computation `a=b+c` that contains only one operator). We will use the terms *statement* and *instruction* interchangeably, both meaning a statement in the three-address code representation.

2.1. Abstract Dynamic Slicing

Existing dynamic analysis algorithms can be classified into two major categories: (1) *forward analysis* that maintains a set of analysis states and updates them on-the-fly along the execution, and (2) *backward analysis* that records history information as the program executes and this information needs to be consulted later for computing analysis results. For a forward analysis, the set of analysis states at any point during the execution would provide sufficient information regarding the properties of the program that the analysis is designed to discover. Analyses in this category often perform lightweight computation that needs only a small amount of information from the execution. A typical example of forward analysis is taint analysis [Newsome and Song 2005; Xu et al. 2006], which propagates taint information along the information flow. When data flows from a trusted component to an untrusted component, its taint information is checked to verify if it comes from a tainted source. At any point during the execution, having the taint information for each piece of runtime data satisfies the need of the analysis.

A backward analysis is much more complex and needs to perform heavier computation than a forward analysis. The computation is often too expensive to be done along the main execution, so it has to be undertaken either offline or in a new execution (e.g., a new thread or process), completely separated from the original one. The recorded history is used to “replay” a relevant part of the normal execution to enable this heavyweight computation. Examples of backward analyses include `null` value propagation analysis [Bond et al. 2007], automated bug location from execution traces [Zhang et al.

2007], dynamic object type-state checking [Arnold et al. 2008], event-based execution fast forwarding [Zhang et al. 2006b], as well as the copy analysis and the cost-benefit analysis that will be discussed in this article. For brevity, we will refer to such analysis problems as BDF (*Backward Dynamic Flow*) problems. In general, BDF problems can be solved by dynamic slicing [Korel and Laski 1990; Zhang et al. 2003, 2006a; Zhang and Gupta 2004a; Wang and Roychoudhury 2008]. For instance, computing the cost of a value requires the identification of all instruction instances that (directly or transitively) contribute to the computation of the value, which can only be obtained by performing slicing on a dynamic dependence graph.

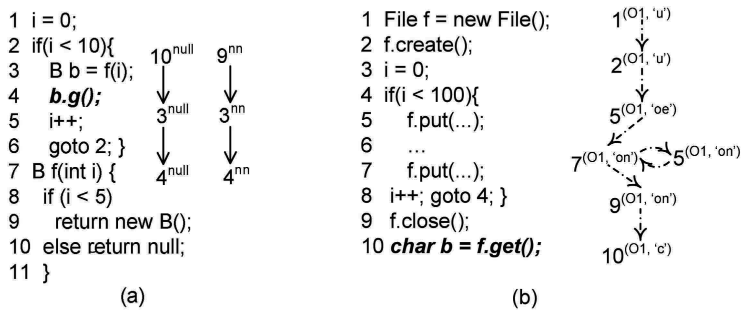
In dynamic slicing, the instrumented program is first executed to obtain an execution trace with control-flow and memory reference information. At a pointer dereference, both the data that is referenced and the pointer value (i.e., the address of the data) are captured. Based on a dynamic data dependence 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 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.

Existing dynamic slicing algorithms are client oblivious. They profile the entire execution of a program under the assumption that all details of the execution will be used later by the client. However, our experience shows that, in most cases, the client analysis needs only a very small portion of the collected information, leading to wasted runtime computation and space to collect and store data never used. We observe that, 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) has little or no influence on 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.

Based on this observation, we propose a novel runtime technique, called *abstract dynamic slicing*, that performs slicing over a bounded abstract domain. The key idea is to design a client-conscious slicing algorithm that collects only the information relevant to this domain. This bounded abstract domain $\mathcal{D}$ is client specific and provided by the analysis designer before the execution. $\mathcal{D}$ contains identifiers that define equivalence classes in $\mathcal{N}$ and that encode the semantics of the client analysis. 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 $f_a : \mathcal{N} \to \mathcal{D}$ 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. The corresponding dependence graph will be referred as an *abstract data dependence graph*.

Definition 2.2 (*Abstract Data Dependence Graph*). An abstract 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 to a location that is used by an instance of b



```
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();
```

Fig. 2. Data dependence graphs for two BDF problems. Line numbers are used to represent the corresponding instructions. Arrows with solid lines are def-use edges. (a) Null origin tracking. Instructions that handle primitive-typed values are omitted; (b) type-state history recording; arrows with dashed lines represent "next-event" relationships.

mapped to b^k, without an intervening write to that location. $\mathcal{F}$ 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 data dependence graph defined before. 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 two analyses and their formulations in our framework.

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 existing `null` value tracking approaches (e.g., Bond et al. [2007]) 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 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 [Arnold et al. 2008], this analysis tracks the typestates of the specified objects and records the history of 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 $f_a(j)$ maps each instruction instance of the form a^j to a pair (`alloc`(a^j), `state`(a^j)) if a^j invokes a method on an object $\in \mathcal{O}$, and the method can cause the object to change its state; otherwise, $f_a(j)$ is undefined (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 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 Arnold et al. [2008], and they can be easily obtained by storing 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 2(b) is not strictly a dependence graph, the “next-event” edges can be conceptually thought of as def-use edges between nodes that write and read the object state tag.

Similarly to how a dataflow analysis or an abstract interpreter employs static abstractions, abstract dynamic slicing uses abstract domains for dynamic dataflow, recognizing that it is only necessary to distinguish instruction instances that are important for the client analysis.

2.2. Modeling of Objects

It is important for certain dependence graph nodes to have information about the objects they access. This information can be used for various client analyses to discover properties of objects and data structures. A typical handling of objects for analysis of object-oriented programs is to use an allocation site to represent the set of runtime instances it creates. We use the same idea in this framework to add object information. For each instruction that has a heap access $o.f$ expression (i.e., either a read or a write of a field f in a heap object pointed-to by o), we can augment the user-provided semantic domain $\mathcal{D}_{orig}$ with the set of all possible allocation sites $\mathcal{O}$ creating objects that o may point to at runtime. Hence, the new domain $\mathcal{D}$ for this instruction becomes $\mathcal{D}_{orig} \times \mathcal{O}$, where $\mathcal{D}_{orig}$ is the original abstract domain defined by the framework user for this instruction.

An analysis that requires a highly precise handling of heap accesses may need to consider calling contexts when modeling objects. Calling contexts can be easily added into our framework. For each instruction a that accesses a heap location $o.f$, suppose $\mathcal{C}$ is a set of abstractions of all possible calling contexts under which a can be executed. $\mathcal{C}$ can be easily added by redefining $\mathcal{D}$ for a to be $\mathcal{D}_{orig} \times \mathcal{O} \times \mathcal{C}$. Different types of context representations can be employed for different analyses. For example, object-based contexts [Milanova et al. 2005] are suitable for analyzing programs with large numbers of object-oriented data structures, and call-chain-based contexts fit better with the analysis of procedural behavior. While our bloat detection techniques focus on object sensitivity [Milanova et al. 2005], all of these different context representations can be easily added into our framework.

In order to effectively reduce the runtime overhead and bound the memory space needed for abstract dynamic slicing, $\mathcal{D}$ needs to be statically bounded and its size cannot depend on any (unbounded) dynamic behavior of the program, such as loop iterations and method invocations. This imposes challenges for modeling calling contexts in our framework—it can be extremely difficult to statically identify all contexts for each instruction, especially when the program is large and uses dynamic language features such as reflection. To solve the problem, we can define $\mathcal{C}$ to be a small fixed-size set, and then design an encoding (hash) function to map each dynamic calling context to an element in $\mathcal{C}$. Example encoding functions can be found later in Section 3 and Section 4.

2.3. Framework Implementation

We have implemented this framework in J9 (build 2.4, J2RE 1.5.0 for Linux x86-32), a commercial Java Virtual Machine developed by IBM [J9 Java Virtual Machine 2011]. 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.

At the heart of this framework is a whole-program dataflow tracking engine that is used to track the flow of data and perform predefined operations as instructions are executed. These operations are declared as callback functions, each of which is implemented by the framework user to update the abstract dependence graph for a certain type of instruction. For example, for each heap load, the framework defines a callback function `handleHeapLoad(Address base, Field field, Word lhs)`, which is invoked by our framework after the heap load. The designer of the client analysis is responsible for implementing these functions to update the dependence graph to collect necessary runtime information.

For abstract dynamic slicing, this set of callback functions is implemented in a way so that the runtime dependence edges can be added among dependence graph nodes during the execution. To do this, each piece of data used during the execution is associated with a piece of *tracking data*, recording the most recent instruction that defines this piece of data. Tracking data is contained in a *shadow memory*, which is separated from the memory space created for the execution. At each instruction instance a^i that uses a piece of data, the instruction b^j defining this piece of data is retrieved from its corresponding tracking data, and a dependence edge is then added to connect the abstract dependence graph node representing a^i to the node representing b^j. Our dataflow framework supports shadowing of all memory in the application, including local variables, static fields, arrays, and instance fields. It is important to note that this handling is different from the usual implementation of dynamic slicing, which first collects an execution trace and then builds a dependence graph from this trace. Our dependence graph is constructed entirely online with the help of shadow locations.

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* [Nethercote and Seward 2007]. 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 $2\times$ 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 framework implementation supports object tagging. For example, we often need to tag each runtime object with

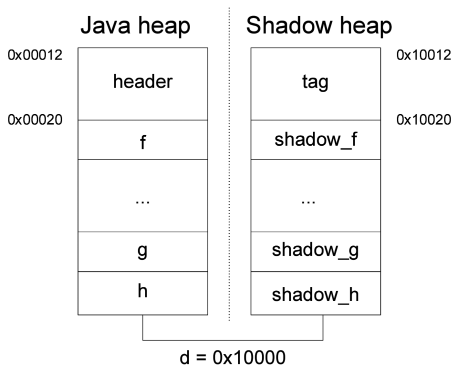


Fig. 3. Representation of an object in the Java heap and its shadow memory. `f`, `g`, and `h` represent the fields in this object. `d` represents a constant offset.

its allocation site in order to relate the runtime 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 [Xu et al. 2009, 2010a; Bond and McKinley 2006, 2007]. 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 runtime. 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 3, where the constant offset between the two heaps is 0×10000.

Tracking stack. The framework 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. Tracking of exceptional dataflow is not supported in our framework, because it usually does not carry important data across method invocations. 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.

3. PROFILING COPIES TO FIND REDUNDANT OPERATIONS

This section presents an instantiation of the abstract dynamic slicing framework to profile copies. The copy profiling algorithm has three major components: profiling regular copy activities, profiling copy chains, and profiling the copy graph. Before discussing how to instantiate the framework, we first introduce copy operations and various kinds of copy profiles that we are interested in.

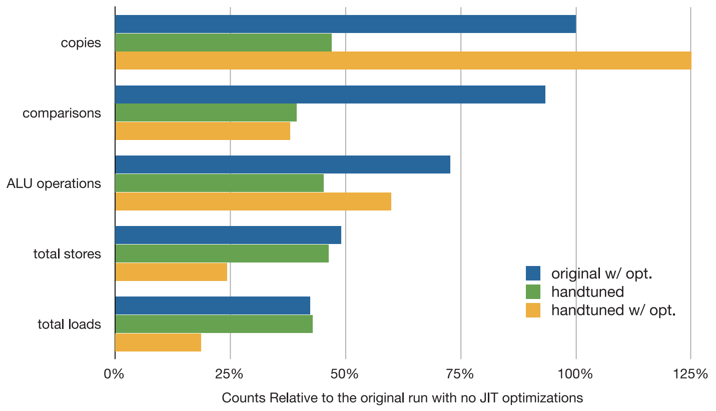


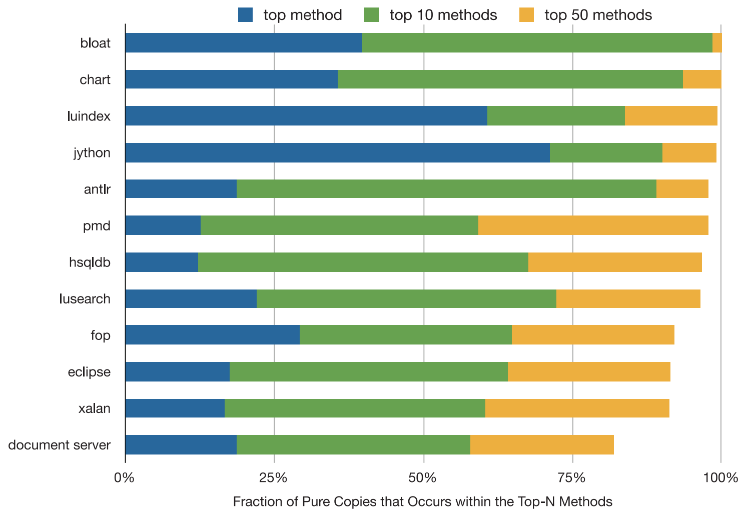
Fig. 4. A breakdown of activity in a document processing server application. The baseline, at 100%, is the original code run with JIT optimizations disabled. This baseline is compared to the original code with JIT optimizations enabled and to an implementation with a dozen hand-tunings.

3.1. Profiling Copy Activities

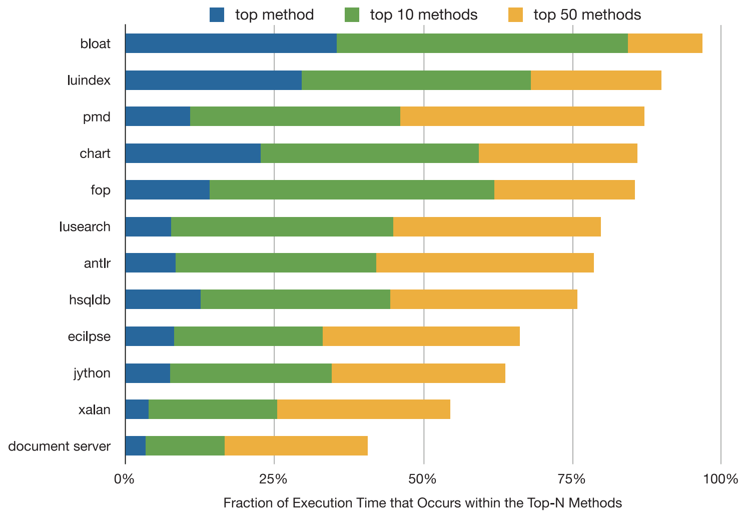
A *copy* operation is a pair of a heap load and a heap store instruction that transfers a value, unmodified, from one heap location to another. A *copy profile* counts the number of copies; a copy operation is associated with the method that performed the write to the heap. Although the profiling tracks the propagation 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 4 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 significantly affect the number of copies and comparisons, and in some cases makes things worse.

Observe that the JIT is good at what one would expect: reducing ALU operations and the total number of loads and stores; common subexpression elimination probably explains much of these effects. On the other hand, the JIT does not greatly affect the number of copies; it also has no great effect on the number of comparison instructions. Comparisons often indicate overprotective or overgeneral implementations, which also exhibit runtime bloat. The hand-tuned implementation greatly reduces the number of copies and the number of comparison operations. Flat copy profiles show that copies serve as good indicators of problems. From them, we learn that copy activity is concentrated in a small number of methods. From the copy profiles for the DaCapo benchmark suite [Blackburn et al. 2006] and the document management server, we observe the concentration of copies. Figure 5(a) shows that, across the board, a small number of methods explain most of the copy activities in these programs. Even just the top method explains at least 12% of the copies, often much more. For comparison, Figure 5(b) shows the concentration of execution time in methods. As expected, the more complex applications, such as the Eclipse DaCapo benchmark and the document



(a) copy concentration



(b) time concentration

Fig. 5. In copy concentration, a small number of methods explain most of the copies in the DaCapo benchmarks, version 2006-10-MR2, and the document management server; in contrast to copies, which are concentrated even for complex applications, the time spent in methods is only concentrated for the simpler benchmarks.

management server, have very flat execution-time method profiles; this is in contrast to the highly concentrated copy profiles for those same programs.

3.2. Copy Chains

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, *copy graph*, that can be used to identify chains of copies.

```
1 class List{
2    Object[] elems; int count;
3    List(){ elems = new Object[1000]; }
4    List(List l){ this(); // call default constructor
5       for(Iterator it = l.iterator(); it.hasNext();)
6          { add(it.next()); } }
7    void add(Object m){
8       Object[] t = this.elems;
9       t[count++] = m;
10   }
11   Object get(int ind){
12      Object[] t = this.elems;
13      Object p = t[ind]; return p;
14   }
15   Iterator iterator(){
16      return new ListIterator(this);
17   }
18 }
19 class ListIterator{
20    int pos = 0; List list;
21    ListIterator(List l){
22       this.list = l;
23    }
24    boolean hasNext(){ return pos < list.count - 1;}
25    Object next(){ return list.get(pos ++);}
26 }
27 class ListClient{
28    List myList;
29    ListClient(List l){ myList = l; }
30    ListClient deepClone(){
31       List j = new List(myList);
32       return new ListClient(j);
33    }
34    ListClient shallowClone(){
35       return new ListClient(myList);
36    }
37 }
38 static void main(String[] args){
39    List data1 = new List();
40    for(int i = 0; i < 1000; i++)
41       data1.add(new Integer(i));
42    List data2 = new List();
43    for(int i = 0; i<5; i++){
44       data2.add(new String(args[i]));
45       System.out.println(data2.get(i));}
46    ListClient c1 = new ListClient(data1);
47    ListClient c2 = new ListClient(data2);
48    ListClient new_c1 = c1.deepClone();
49    ListClient new_c2 = c2.shallowClone();
50 }
```

Fig. 6. Running example.

Definition 3.1 (*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.

A motivating example. The code in Figure 6 is used for illustration throughout the section. 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 clone methods `shallowClone` and `deepClone`, which return a new `ListClient` object by reusing the old backing list and by copying list elements, respectively. The entry method `main` creates two lists `data1` and `data2` and initializes them with 1,000 Integer and 5 String objects (lines 41 and 44). The two lists are then passed into two `ListClient` objects and eventually two new `ListClient` objects are created by calling `deepClone` and `shallowClone`. 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.

Figure 7 depicts the steps in the creation of a single-edge copy chain from the invocation of `deepClone` at line 48. The call to the constructor of `List` at line 31 reads each element from the old list and adds it to the new list to be created. This chain starts at the heap load at line 13 that reads an `Integer` object from the heap (i.e., an array) to the stack and ends at the heap store at line 8 that writes this object back to the heap (i.e., another array), abstracting away the three intermediate copies (shown as step 1 to step 3) that occur on the stack. Both the source array and the target array are denoted by O_3 because they are created at line 3 in the code. (For now, the reader can

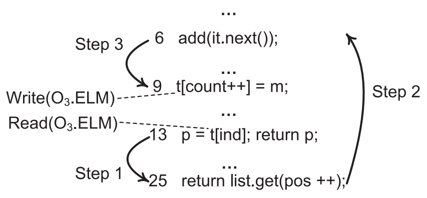


Fig. 7. A copy chain due to `ListClient.deepClone`. Line numbers 6, 9, 13, and 25 correspond to the code in Figure 6.

ignore the naming scheme; it will be discussed shortly.) The copy chain in Figure 7 is thus $O_3.ELM \rightarrow O_3.ELM$, where ELM is the placeholder for 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_{44} (i.e., `new String`) is $O_{44} \rightarrow O_3.ELM \rightarrow C$. This chain ends in consumer node C because the data goes into method `println` which eventually calls native method `write`.

It is clear that copy-chain profiling is a BDF problem that can be solved by regular dynamic slicing, which cannot scale to large real-world applications. To make the analysis scale to these applications, we simplify the problem by applying abstractions in copy chains, so that the resulting problem can be solved by the abstract dynamic slicing framework presented in Section 2.

3.3. Copy Graph

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.

Definition 3.2 (*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 runtime 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. In order to exploit the abstract dynamic slicing algorithm, we first need to determine how heap locations are abstracted in the copy graph, namely the object naming scheme.

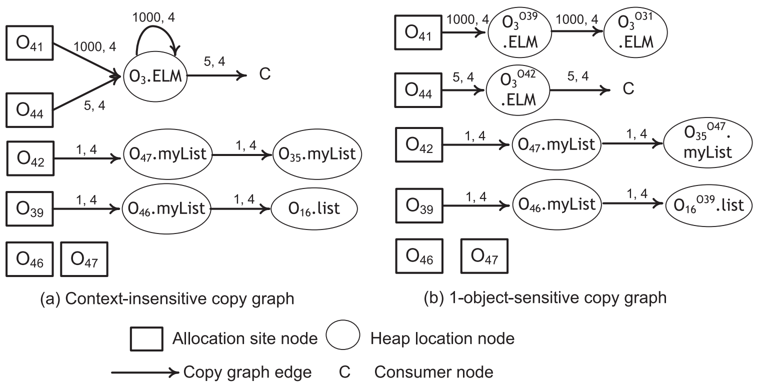


Fig. 8. Partial copy graph with context-insensitive and context-sensitive object naming. Each edge is annotated with a pair $\langle f, b\rangle$ where f is the edge frequency and b is the number of bytes copied on each edge.

To match the object naming scheme in the abstract dynamic slicing framework, an allocation site is used to represent the set of runtime 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 is the placeholder for the field name in the abstraction. Individual array elements are not distinguished: considering each element separately may introduce infeasible time and space overhead.

For illustration, consider the partial copy graph in Figure 8(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 $O_{41} \xrightarrow{1000,4} O_3.ELM$ copies the `Integer` objects created at line 41 into the array referenced by `data1`'s `elems` field. This edge consists of a sequence of copies via parameter passing (line 41 and line 9). This sequence of copies occurs 1,000 times, and each time 4 bytes of data are transferred. Both $O_{41} \xrightarrow{1000,4} O_3.ELM$ 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 runtime, 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 dataflow illustrated in Figure 7. 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 8(a), one could imprecisely conclude that both O_{41} and O_{44} 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 that maps each runtime 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.

Context sensitivity. When naming a runtime 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) [Shivers 1988], which considers a sequence of call sites invoking the analyzed method, and object sensitivity [Milanova et al. 2005], in which the context is the sequence of static abstractions of the objects (i.e., allocation sites) that are runtime 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 (refer to Aldrich et al. [2002], Clarke and Drossopoulou [2002], Boyapati et al. [2003], and Heine and Lam [2003]) and is suitable for improving the analysis precision for real-world applications making use of a large number of object-oriented data structures.

Figure 8(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 runtime owner. By adding context sensitivity, paths that start from O_{41} and O_{44} do not share any nodes. Note that there are no contexts for nodes $O_{39}, \ldots, O_{47}$ 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.

3.4. Instantiating the General Framework to Compute a Copy Graph

As we focus on transitive copies among heap locations, instructions that read from and write to heap locations and that perform data computations are of particular interest during the dependence profiling. In the original abstract dependence graph, there may be sequences of dependence edges starting at an instruction (e.g., a) that reads a piece of data from the heap, and ending at an instruction (e.g., b) that writes this data (after being manipulated) back to the heap. Because in this section we are not interested in how this piece of data is manipulated on the stack, the original abstract dynamic slicing algorithm is optimized in a way so that we add (transitive) dependence edges directly from a to b, abstracting away dependence relationships among stack copy instructions in the middle. Nodes representing (irrelevant) stack copy instructions can thus be omitted in the dependence graph, leading to increased time and space efficiency for the profiling.

Abstract domain $\mathcal{D}$ needs to be defined only for relevant instructions. The definitions of $\mathcal{D}$ for different types of instructions are as follows.

—Alloc site $a = new\ O$: $\mathcal{D} = \{\epsilon\}$;
—Computation $a = b + c$: $\mathcal{D} = \{\epsilon\}$;
—Static field access $a = A.f / A.f = a$: $\mathcal{D} = \{\epsilon\}$;
—Instance field access $a = o.f / o.f = a$:

$$\mathcal{D} = \begin{cases} \mathcal{O} \quad \textit{for context-insensitive copy graph}, \text{ where } \mathcal{O} \text{ is the set of allocation sites} \\ \qquad \text{for all possible runtime objects pointed by } o; \\ \mathcal{O} \times \mathcal{C} \quad \textit{for 1-object-sensitive copy graph}, \text{ where } \mathcal{C} \text{ is the set of allocation sites} \\ \qquad \text{for all possible receiver objects of the method containing the instruction;} \end{cases}$$

ALLOC
$$\frac{\mathsf{P}' = \mathsf{P}[O_i \mapsto \langle allocID(new\ X), \mathsf{P}_o(O_{this})\rangle] \quad \mathsf{V}' = \mathsf{V} \cup \{a^\epsilon\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^\epsilon]}{\mathsf{V}, \mathsf{S}, \mathsf{P} \Rightarrow^{a:i=new\ X} \mathsf{V}', \mathsf{S}', \mathsf{P}'}$$

ASSIGN
$$\frac{\mathsf{S}' = \mathsf{S}[i \mapsto S(k)]\}}{\mathsf{V}, \mathsf{E}, \mathsf{S} \Rightarrow^{a:i=k} \mathsf{V}', \mathsf{E}', \mathsf{S}'}$$

COMPUTATION
$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^\epsilon\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^\epsilon] \quad \mathsf{E}' = \mathsf{E} \cup \{a^\epsilon \triangleright \mathsf{S}(k)\} \cup \{a^\epsilon \triangleright \mathsf{S}(l)\}}{\mathsf{V}, \mathsf{E}, \mathsf{S} \Rightarrow^{a:i=k \oplus l} \mathsf{V}', \mathsf{E}', \mathsf{S}'}$$

PREDICATE
$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^\epsilon\} \quad \mathsf{E}' = \mathsf{E} \cup \{a^\epsilon \triangleright \mathsf{S}(i)\} \cup \{a^\epsilon \triangleright \mathsf{S}(k)\}}{\mathsf{V}, \mathsf{E}, \mathsf{S} \Rightarrow^{a:if\ (i>k)\{\ldots\}} \mathsf{V}', \mathsf{E}', \mathsf{S}'}$$

LOAD STATIC
$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^\epsilon\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^\epsilon]}{\mathsf{V}, \mathsf{S} \Rightarrow^{a:i=A.f} \mathsf{V}', \mathsf{S}'}$$

STORE STATIC
$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^\epsilon\} \quad \mathsf{E}' = \mathsf{E} \cup \{a^\epsilon \triangleright \mathsf{S}(i)\}}{\mathsf{V}, \mathsf{E} \Rightarrow^{a:A.f=i} \mathsf{V}', \mathsf{E}'}$$

LOAD FIELD
$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^{\langle \mathsf{P}_o(O_v), \mathtt{h}(\mathsf{P}_c(O_v))\rangle}\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^{\langle \mathsf{P}_o(O_v), \mathtt{h}(\mathsf{P}_c(O_v))\rangle}\}]}{\mathsf{V}, \mathsf{E} \Rightarrow^{a:i=v.f} \mathsf{V}', \mathsf{E}'}$$

STORE FIELD
$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^{\langle \mathsf{P}_o(O_v), \mathtt{h}(\mathsf{P}_c(O_v))\rangle}\} \quad \mathsf{E}' = \mathsf{E} \cup \{a^{\langle \mathsf{P}_o(O_v), \mathtt{h}(\mathsf{P}_c(O_v))\rangle} \triangleright \mathsf{S}(i)\}}{\mathsf{V}, \mathsf{E} \Rightarrow^{a:v.f=i} \mathsf{V}', \mathsf{E}'}$$

METHOD ENTRY
$$\frac{\mathsf{S}' = \mathsf{S}[t_i \mapsto \mathsf{T}(i)] \text{ for } 1 \le i \le n \quad \mathsf{T}' = \emptyset}{\mathsf{S}, \mathsf{T} \Rightarrow^{a:m(t_1, t_2, \ldots, t_n)} \mathsf{S}', \mathsf{T}'}$$

RETURN
$$\frac{\mathsf{T} = \emptyset \quad \mathsf{T}' = (\mathsf{S}(i))}{\mathsf{T} \Rightarrow^{a:return\ i} \mathsf{T}'}$$

Fig. 9. Inference rules defining the customized abstract dynamic slicing algorithm.

For each allocation site and computation instruction, $\mathcal{D}$ is a singleton set that contains the empty calling context, and all instances share one single dependence graph node. This is because in the copy graph each allocation site is a producer and each computation is a consumer, and we do not need to distinguish their instances. The same handling is used for static field access, as each static field has one single location node in the copy graph. For an instance field access that reads from/writes to $o.f$, $\mathcal{D}$ is defined in a way so that different heap locations that $o.f$ may represent can be distinguished. For a context-insensitive copy graph, o's allocation sites are sufficient to distinguish the different objects that o points to, and thus, $\mathcal{D}$ is simply defined as a set of o's allocation sites. For a context-sensitive copy graph, as heap locations are differentiated using a combination of an object allocation site and an object context, we define $\mathcal{D}$ as the Cartesian product of the set of allocation sites for o and the set of allocation sites for the receiver object of the method containing the load/store. Hence, the copy graph can be easily derived from the abstract dependence graph collected by our abstract dynamic slicing framework.

As discussed in Section 1, in order to efficiently use abstract dynamic slicing, $\mathcal{D}$ has to be a relatively small domain. However, for a context-sensitive copy graph, $\mathcal{D}$ is a Cartesian product $\mathcal{D}_{orig} \times \mathcal{O} \times \mathcal{C}$, which can be large for real-world applications. To make the context-sensitive copy profiling scalable, the size of $\mathcal{D}$ has to be significantly reduced. Here we keep $\mathcal{O}$ while limiting the size of context set $\mathcal{C}$ to be a fixed number s (short for "slots"), specified by the user as a parameter of the profiling tool. Now the domain is still $\mathcal{O}$ for a context-insensitive copy graph and it is simplified to $\mathcal{O} \times [0, s-1]$ for a context-sensitive copy graph. An encoding function $\mathtt{h}$ is used to map an object context (i.e., the allocation site of the receiver object) to such an integer. Here a simple hash function *context* % c is employed to map the object context to a value in $[0, c-1]$. A default c value of 4 was used for the studies described in Section 3.6. Despite its simplicity, very few contexts for an object have conflicts (i.e., they map to the same value) when using this function (as reported in Section 3.7).

Profiling algorithm. Figure 9 shows a list of inference rules defining the customized abstract dynamic slicing algorithms. Each rule is of the form $\mathsf{V}, \mathsf{E}, \mathsf{S}, \mathsf{P}, \mathsf{T} \Rightarrow^{a:i=\ldots} \mathsf{V}', \mathsf{E}', \mathsf{S}', \mathsf{P}', \mathsf{T}'$ 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., when $\mathsf{S} = \mathsf{S}'$), it is omitted. Node domain V contains nodes of the form $a^{\langle alloc, h(c)\rangle}$, where a denotes the instruction, $alloc$ denotes an allocation site ID, and $\mathtt{h}$ is the encoding function that returns the encoded integer of the (context) allocation site c. Dependence 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 $\mathsf{S} : \mathsf{M} \to \mathsf{V}$ maps a runtime 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.

Domain P maps each runtime object (represented by O_i for variable i) to a pair $\langle alloc_i, alloc_j\rangle$, where $alloc_i$ and $alloc_j$ are the IDs of the allocation sites creating O_i and the receiver object of the method containing $alloc_i$, respectively. $\mathsf{P}_o(O_i)$ and $\mathsf{P}_c(O_i)$ are used to represent $alloc_i$ and $alloc_j$, respectively. Based on the definitions of $\mathcal{D}$ for different types of instructions (shown earlier in this section), all rules are defined in expected ways. Dependence edges are added only among nodes representing instances of relevant instructions. Thus, rule ASSIGN propagates tracking data (containing the node that defines the original data), but does not add any dependence edge to the dependence graph.

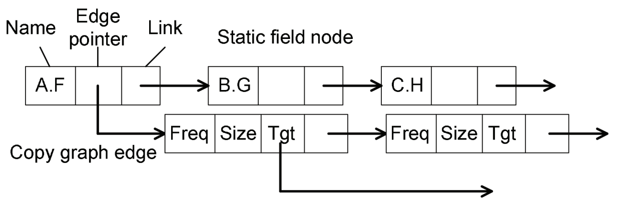
Note that the slicing algorithm shown in Figure 9 is a form of *thin slicing* [Sridharan et al. 2007], a technique that considers only direct locations that are part of the dataflow in the generated slice, while filtering out locations that are indirectly used to obtain the direct locations. For instance, 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 copy profiling, because we focus only on the flow of data among heap locations and do not need to worry about the base pointers via which heap locations are referenced.

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 $\mathsf{T}(1), \ldots, \mathsf{T}(n)$. 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 stack is updated by removing the tracking data for the actuals. At the return site, T is updated to store the tracking data for the return variable i.

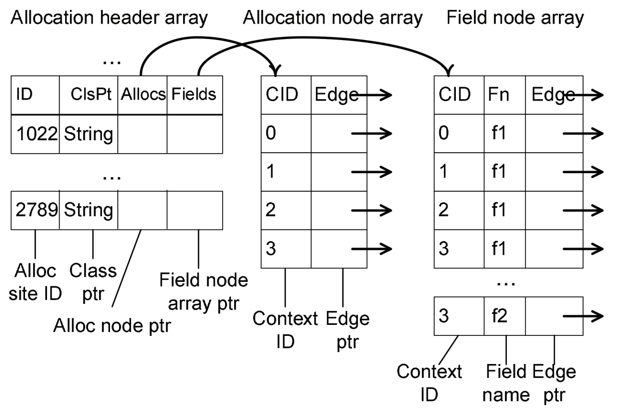
The rule for call sites is not shown in Figure 9, as it requires splitting a call site into a *call* part and a *return* part and reasoning about both of them. Immediately before the call, the tracking data for the actual parameters is pushed on tracking stack T. Immediately after the call, the tracking 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 *and* S. Although environments P and S have different mathematical meanings, both are implemented using shadow locations. For each runtime object O_v, the encoded context $\mathsf{P}(O_v)$ is implemented as O_v's tag stored in the location corresponding to O_v's header in the shadow heap (shown in Figure 3).

From dependence graph to copy graph. It is straightforward to build a copy graph from the dependence graph, as instruction instances in the dependence graph are



(a) data structure for static field nodes



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

Fig. 10. Data structure overview.

abstracted in the same way as heap locations in the copy graph are abstracted. For each collected dependence graph edge $a^c \rightarrow b^d$, a corresponding edge $m \rightarrow n$ is added into the copy graph. The rules are outlined as follows.

—If a is an allocation site instruction, m is an allocation site node in the copy graph.
—If a (or b) is a computation instruction, a predicate instruction, or a call to a native method, m (or n) is the consumer node C.
—If a (or b) is an instruction accessing static field, m (or n) is a static field node $A.f$.
—If a (or b) is an instruction accessing instance field, c (or d) must have the form $\langle alloc, cxt\rangle$. m (or n) is thus an instance field node, represented by $alloc.f$ and $alloc^{cxt}.f$, for the context-insensitive copy graph and the context-sensitive copy graph, respectively.

Frequency information for each dependence edge is collected in the underlying framework, which is annotated with the corresponding copy graph edge during the copy graph building.

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 runtime heap location to its name (which in our analysis is a copy graph node address). Figure 10 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 that points to a linked list of copy graph edges that leave this node. Edge adding occurs at runtime. 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 predefined value c (as mentioned earlier in this section). We have experimented with different values of c and these results are reported in Section 3.7.

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 1,000, 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 $alloc\_headers[1000]$.

3.5. 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 runtime behavior and pinpointing performance bottlenecks.

3.5.1. 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 problems. Another important metric is chain length—the longer a copy chain, 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 predefined 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., 17,7581), and the data size (e.g., 2 bytes).

3.5.2. 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, `deepClone` 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 subgraph, 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 predefined 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, `deepClone` 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 4,000 is associated with the two array objects (1,000 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_{46}, 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_{46}, O_{32}) highlights this underlying cause.

3.5.3. 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_{46} and O_{47} 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 a 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.

3.5.4. 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 to which they belong. 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.

3.6. 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.

DaCapo bloat benchmark. 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[1]. 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.

Java 5 GregorianCalendar. A recurring problem with the Java 1.5 standard libraries is the poor performance of calendar-related classes [Sun Java Forum 2014]. 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

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. We have looked at the top 20 pairs reported by the clone detector and found that all of them revealed heavy

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

Table II. The Top 9 Allocation Sites Reported for Eclipse 3.1

Rank	*Frequency*	*Detailed Description*
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]

copies between the reported data structures. These copies are not necessarily object clones (e.g., a method call `list1.addAll(list2)` would make the clone detector report the pair <`list1`, `list2`> while they are not true clones), but they all indicate problems that need to be fixed for improved performance.

DaCapo eclipse benchmark. 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 [Java Development Blog 2009]. We ran Eclipse 3.1 from the DaCapo benchmark set and used the NATH analysis to identify allocation sites whose runtime objects are never assigned to the heap. The top nine allocation sites are shown in Table II.

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 295,004 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 requester (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 invokes 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 the number of allocated objects by millions and improves the running time by 2.8%. In large applications

Table III. Eclipse 3.1 Performance Problems, Fixes, and Performance Improvements

Class	*Site#*	*Modification*	*#Total Objs*	*#GCs*	*Time(s)*
Original	—	—	*273,991,250*	*478*	*143.6*
MethodInfo, Field-Info, TypeInfo	1, 3, 5	Directly pass the data	272,461,138	460	139.6
PackageFragment	6, 7	Get IResource directly from String	272,429,471	448	138.3
SimpleWordSet	2	In-place rehash	272,395,776	430	136.8
HashtableOfObject	9	In-place rehash	272,320,499	424	134.0
Total Reduction	—	—	*1,670,751*	*54*	*9.6*

with no single hot spot, significant performance improvements are possible by accumulating several such "small" improvements, as illustrated next.

Table III shows a list of several problems we identified with the help of the analyses. For each problem, the table shows the problematic class (*Class*), the IDs of its corresponding allocation sites in Table II, our code modification, the number of total allocated objects (*#Objs*), the total number of GC invocations (*#GCs*), and the running times. Row *Original* characterizes the original execution. 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.

3.7. 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 (in the DaCapo 9.12 version) 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 IV 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 V, Table VI, and Table VII, 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), 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 IV), where $i \in \{4, 8, 16\}$. The copy graph itself consumes a relatively small amount of memory. Other than for IBM

Table IV. Copy Graph Size and Time/Space Overhead, Part 1

	Original	Context-insensitive			
Program	$T_{orig}(s)$	$\#N_0$	$\#E_0$	$M_0(Mb)$	$T_0(s)$ $(\times)$
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	1800·0∗	17146	12637	502.4	1800·0∗
IBM DMS	1800·0∗	147517	87531	519.6	1800·0∗

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.

Table V. Copy Graph Size and Time/Space Overhead, Part 2

	1-object-sensitive ($c = 4$)			
Program	$\#N_4$	$\#E_4$	$M_4(Mb)$	$T_4(s)$ $(\times)$
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	1800·0∗
IBM DMS	193707	180187	533.7	1800·0∗

The columns report the same measurements as Table IV, but for 1-object-sensitive copy graph with 4 context slots.

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

Table VI. Copy Graph Size and Time/Space Overhead, Part 3.

	1-object-sensitive ($c = 8$)			
Program	$\#N_8$	$\#E_8$	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	1800·0$*$
IBM DMS	381072	242049	571·2	1800·0$*$

The columns report the same measurements for 1-object-sensitive copy graph with 8 context slots.

Table VII. Copy Graph Size and Time/Space Overhead, Part 4

	1-object-sensitive ($c = 16$)			
Program	$\#N_{16}$	$\#E_{16}$	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	1800·0$*$
IBM DMS	755829	304759	652.3	1800·0$*$

The columns report the same measurements for 1-object-sensitive copy graph with 16 context slots.

implementation. Hence, the difference between the running times of profiling context-insensitive and context-sensitive copy graphs is noise. The only 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. A possible future direction is to use sampling-based profiling to obtain the same or similar results. Another possibility is to employ static preanalyses to reduce the cost of the subsequent dynamic analysis.

Table VIII shows measurements for the copy chains obtained from a context-insensitive copy graph, including the total number of generated chains (*#Chains*) and

Table VIII. Copy Chains and NATH Objects

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

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 runtime objects.

Table IX. Average Node *Fanout* 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

	Average fan-out				Context conflict ratio		
Program	*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

the average length of these chains (*Length*). The table also shows the number of NATH allocation sites and NATH runtime 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 IX lists the average node *fanout* 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 fanout is the number of its outgoing edges. The average fanout 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 sites) that do not have outgoing edges. In addition, the more slots are used to represent contexts, the smaller the average fanout, 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:

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

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 their lifetimes.

Summary. The study presented in this section 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), these data-oriented observations become valuable and more informative. This section also demonstrates that the abstract dynamic slicing framework can be easily instantiated to collect runtime dataflow-based information. Excessive copying is just one example of such an activity. The next section describes another (more complex) instantiation of the framework to find high-cost low-benefit data structures.

4. COMPUTING COSTS AND BENEFITS TO FIND LOW-UTILITY DATA STRUCTURES

One primary symptom of runtime bloat is an imbalance between costs and benefits. The costs of forming a data structure, of computing and storing its values, are out of line with the benefits gained over the uses of those values. In this section, we focus on these *low-utility data structures*. A data structure has low utility if the cumulative costs of creating its member fields outweighs a weighted sum over the subsequent uses of those fields. We show how the costs and benefits of data structures can be obtained from the abstract dynamic slicing framework.

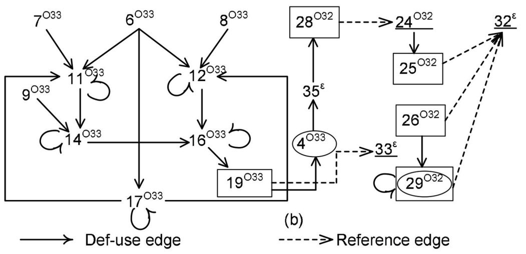
4.1. Cost Computation

Definition 4.1 (*Absolute Cost*). Given a nonabstract 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.

We first define the absolute cost of a runtime value as the total number of bytecode instructions transitively required to produce it; each instruction is treated as having unit cost. Here we still use *thin slicing* [Sridharan et al. 2007], because the cost of a pointer value (e.g., a) should not be attributed to the cost of the value computed from the heap location (e.g., $a.f$) accessed through the pointer a. 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 in understanding the overall execution. To mitigate the problem, we propose to compute *abstract costs* instead of *absolute costs*. Unlike an absolute cost that is computed based on a concrete dependence graph, an abstract cost is computed based on an abstract dependence graph, obtained from the abstract dynamic slicing framework presented in Section 2. Therefore, the computation of abstract costs is much more efficient than that of absolute costs. The abstraction we use to instantiate the framework is described as follows.

```
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
$O_{32}{}^{\varepsilon}$	4	1	6	1
$O_{33}{}^{\varepsilon}$	4005	4	4005	4

(d)

Fig. 11. (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.

Each instruction instance in the concrete dependence graph is abstracted based on dynamic calling contexts. Similarly to copy graph profiling in Section 3, contexts are represented by object sensitivity [Milanova et al. 2005], which is well suited for modeling of object-oriented data structures. However, unlike 1-object sensitivity in Section 3 that only uses one-level receiver objects for contexts, a calling context for an instruction in this technique is represented by the entire chain of receiver objects for the invocation of the instruction, which is potentially more precise than 1-object sensitivity. Thus, domain $\mathcal{D}_{cost}$ contains all possible chains of allocation sites. Abstraction function is defined as $f_a(j) =$ `objCon(cs(`a^j`))`, where function `cs` takes a snapshot of the call stack when a^j is executed, and function `objCon` (i.e., short for object-concatenation) maps this snapshot to the corresponding chain of allocation sites O_i for the runtime receiver objects.

$\mathcal{D}_{cost}$ is unbounded in the presence of recursion, and even for a recursion-free program its size is exponential. Similarly to the context reduction approach used in Section 3, we limit the size of $\mathcal{D}_{cost}$ further to be a fixed number s, specified by the user as a parameter of the profiling tool. Now the domain is simply the set of integers 0 to $s-1$. Similarly, an encoding function `h` is designed 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 sum of the absolute costs of values produced by the instruction instances represented by the node.

Definition 4.2 (*Abstract Cost*). Given an abstract dependence graph G_{cost}, the abstract cost of a node n^k is defined as $\Sigma_{a^j | a^j \rightsquigarrow n^k}$ *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 11 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 to 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 $a \rightsquigarrow n$, there may not exist any dependence 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 4.2, 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 bloat, we augment the graph with two special kinds of nodes, namely, *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 11(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.

4.1.1. Instantiating the Framework to Compute 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 Bond and McKinley [2007] is adapted to perform this mapping: $g_i = 3 * g_{i-1} + o_i$, 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 Bond and McKinley [2007]. In the second step, this encoded value is mapped to an integer in the range $[0, \ldots, s-1]$ using a simple `mod` operation.

Computing cost from the abstract dependence graph G_{cost}. Figure 12 shows a list of inference rules (similar to those in Figure 9) defining the customized abstract dynamic slicing algorithm for cost computation. Similarly, 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 runtime storage location to the content in its corresponding shadow

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

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

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

RETURN
$$\frac{\mathsf{T} = \emptyset \quad \mathsf{T}' = (\mathsf{S}(i))}{\mathsf{T} \Rightarrow^{a:return\ i} \mathsf{T}'}$$

Fig. 12. Inference rules defining the customized abstract dynamic slicing for cost computation.

location (i.e., to its tracking data). 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}. $\mathsf{H} : \mathsf{V} \to \mathsf{Z}$ 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 runtime object. This is done using tag environment P that maps a runtime 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 $('U', (new\ X)^{\mathtt{h}(c)}, '\ ')$, where the allocation site *new X* is annotated with the encoded context integer $\mathtt{h}(c)$. This triple matches only (store) node with effect $('B', (new\ X)^{\mathtt{h}(c)}, *)$, and many spurious reference edges can thus be eliminated. In rule ALLOC, $(new\ X)^{\mathtt{h}(c)}$ is used to tag the newly created runtime object O_i (by updating tag environment P), and this information will be retrieved later when

O_i is dereferenced. In rules LOAD FIELD and STORE FIELD, O_v denotes the runtime 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. Tracking stack T here still contains the tracking data for the actual parameters of the call, as the n top elements $\mathsf{T}(1), \ldots, \mathsf{T}(n)$. T additionally stores 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 runtime 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.

4.2. Relative Object Cost-Benefit Analysis

This section describes a diagnosis technique that identifies data structures with high cost-benefit rates. As discussed in Section 4.3, 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.

4.2.1. Analysis Algorithm.

Definition 4.3 (*Relative Abstract Cost*). Given G_{cost}, the heap-relative abstract cost (HRAC) of a node n^k is $\Sigma_{a^j | a^j \rightharpoonup n^k} \mathit{freq}(a^j)$, where $a^j \rightharpoonup n^k$ if $a^j \rightsquigarrow n^k$ 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.

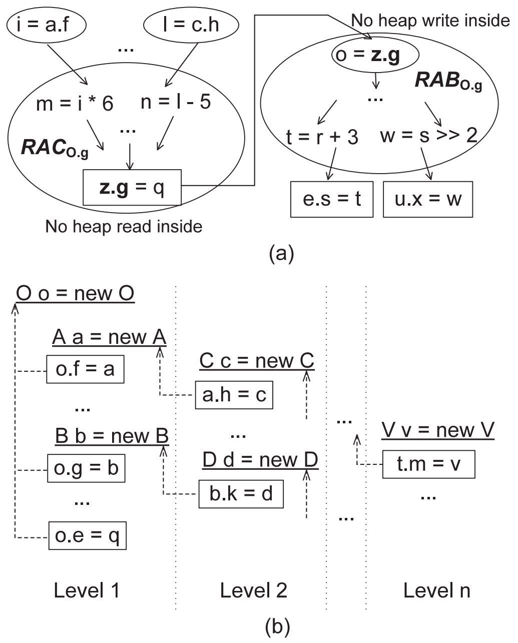


Fig. 13. (a) Relative abstract cost and benefit. Nodes considered in computing RAC and RAB for $O.g$ (where O is the allocation site for the object referenced by z) are included in the two circles, respectively; (b) illustration of n-RAC and n-RAB for the object created by $o = new\ O$; dashed arrows are reference edges.

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} in Figure 11 is only 1 (instead of 4,007), 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 $O^{\epsilon}_{33}.t$ is the HRAC for $19^{O_{33}}$, which is 4,005. The RAC for $O^{O_{32}}_{24}.ELM$ (i.e., the elements of the array object) is 2, which equals the HRAC of node $28^{O_{32}}$ that writes this field.

Definition 4.4 (*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 $O^{\epsilon}_{33}.t$ 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 13(a) illustrates the computation of RAC and RAB.

This definition of benefit captures both the frequency and the complexity of data use. First, the more target heap values 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 recomputing it can be avoided. Whether to store a value in a heap location is essentially a decision involving space-time trade-offs. 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 4.5 (*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 13(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 11 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 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 14 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. Method `isPackage` returns

```
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;
  }
}
```

Fig. 14. Real-world bloat example that our analysis found in `eclipse`.

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. This fix has reduced the number of objects by almost a million.

4.3. Evaluation

We have performed a variety of studies with our technique using the DaCapo benchmark set (version 9.12) [Blackburn et al. 2006], which contains 11 programs in its original version (from `antlr` to `eclipse` in Table X and Table XI) 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. Specifically, 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. 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.

4.3.1. G_{cost} Characteristics and Bloat Measurement. Parts (a) and (b) in Table X 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 is 500Mb for all programs. 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× slowdown on average for $s = 8$ and 72× 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. As with the copy-chain analysis from Section 3, we did not

Table X. Characteristics of G_{cost} (I)

	(a) $s = 8$					(b) $s = 16$				
Program	#*N*(K)	#*E*(K)	*M*(Mb)	*O*(×)	*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

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*).

Table XI. Characteristics of G_{cost} (II)

	(c) Bloat measurement for $s = 16$			
Program	#*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

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*).

attempt to tune the performance of the analysis, but rather to focus on identifying instances of 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). It is also possible to employ various sampling-based or static preprocessing techniques (e.g., from Zhang and Gupta [2004a]) 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 runtime behavior of the application. *CR* measures the degree to which distinct contexts are mapped to the same slots by our encoding function `h`. Following Section 3, *CR-s* for an instruction i is defined as

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

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 XI 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 nonconsumer 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 are 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 overprotective or overgeneral 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.

4.3.2. Case Studies. We have carefully inspected the tool reports for the following six large applications in the DaCapo benchmark set (version 9.12): `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 (i.e., 2 days per application) to find the problems and implement the fixes for these six applications that we had never studied before.

DaCapo sunflow benchmark. 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. The cost-benefit analysis 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 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 a 9%–15% running-time reduction.

DaCapo eclipse benchmark. 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 runtime 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).

It is worth noting that while our fixes passed the DaCapo validation, they might cause latent bugs in real-world development. For example, caching hash codes only works if the key objects are immutable, that is, the refactoring relies on the invariant that the objects being hashed are immutable. This is a determination that cannot be made by a compiler and must be made by a human expert. In this work, we rely on developers to establish the invariant and make the correct refactoring. How to develop techniques to validate the refactoring is an interesting future research topic.

DaCapo bloat benchmark. A case study in Section 3 has found that `bloat` suffers from excessive string creations. This finding is confirmed by our cost-benefit analysis report. Specifically, 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 Section 3) 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`. We replaced the recursion-based implementation with a breadth-first worklist-based implementation, which makes it possible for us to create a very small number of objects to compare all reachable nodes. Eliminating the unnecessary `String` and `StringBuffer` objects as well as the change of the implementation algorithm result in a 37% reduction in running time and a 68% reduction in the number of objects created.

DaCapo derby benchmark. 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%.

DaCapo tomcat benchmark. `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 hard-coded 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).

DaCapo tradebeans benchmark. `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 directly using the int array, we have managed 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 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`). These case studies clearly demonstrate that the cost-benefit analysis is able to identify various performance problems caused by inappropriate design and implementation choices to help developers make informed decisions. 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. For example, inner classes often have large numbers of objects with disjoint lifetimes and very simple data content; it may be possible to keep reusing one single object and resetting its content to improve performance.

5. DISCUSSION

5.1. Comparing Copy Profiling and Cost-Benefit Profiling

Usage scenarios. While both techniques are effective at exposing performance bottlenecks, their design goals are completely different. Cost-benefit analysis is a technique targeting *general bloat* that can be caused by many different factors and can exhibit different symptoms. Hence, it provides a systematic way to answer fundamental questions including “what is bloat” and “how bloat should be found”. The report of the cost-benefit analysis may thus contain many potential problems (of different kinds), some of which are not truly optimizable. Copy profiling is designed to find a specific type of inefficiencies that manifest themselves through excessive copy activities. Thus, it may not be able to find performance problems that do not exhibit large volumes of copies, such as inappropriate choice of collections. However, reports generated by copy profiling can be more focused; they contain only copy-related problems that can be easily understood and fixed by developers. Cost-benefit analysis and copy profiling can be used together in a way so that they can complement each other. For instance, cost-benefit analysis is suitable for an initial round of diagnosis when a performance bottleneck is seen, because the developer may have very little knowledge of the likely cause at this time. After the cost-benefit analysis report is inspected and the developer gains more insights into the problem, a specialized analysis such as copy profiling can be performed to help programmers narrow down the root cause of the bottleneck.

Context sensitivity. From our experimental results, it appears that our framework has a smaller conflict rate for the (encoded) full-chain context sensitivity (in Table IX, Section 3) than the 1-object-sensitivity context representation (in Table X, Section 4), despite the significantly larger number of distinct contexts full-chain context sensitivity may need to represent, compared to 1-object-sensitive context sensitivity. We observed that the following two factors may contribute to this result. First, for a typical method invocation, the number of its distinct runtime receiver object chains is relatively small, at least not substantially bigger than the number of its distinct receiver objects. Receiver objects for many methods on the call stack can be created only by one single allocation site. In fact, we have seen in our experiments that for a large number of method invocations, the numbers of both full receiver object chains and one-level receiver objects are smaller than 16, the maximum number of slots used in our system. This indicates that an ideal solution can be even completely lossless in representing these contexts. Second, the encoding of a receiver object chain (using probabilistic calling context) produces values that are more randomly distributed than regular allocation site IDs (used by 1-object sensitivity), leading to higher likelihood of mapping them into different context slots.

Despite the context-sensitive modeling, it is often still difficult to distinguish objects created by factory methods. For copy profiling, if 1-object sensitivity is not sufficient to distinguish certain objects, the precision for these objects would be the same as the context-insensitive modeling. Increasing the depth of calling contexts is possible only for small programs. For programs such as `Eclipse`, $k = 1$ is the longest context length before they run out of memory. For the cost-benefit analysis, since the full calling context is encoded and used, objects can often be easily distinguished. However, this approach is limited by the fixed number of context slots. Despite much work done on faithfully recording calling contexts, existing methods still leave much to be desired.

5.2. Use of the General Framework for Future BDF Analyses

Both copy profiling and cost-benefit analysis require whole-program dataflow information. Developing them using our framework indicates that the abstract dynamic slicing approach and the shadow memory implementation are flexible enough to be customized to support a variety of other complex dynamic analyses. These two analyses can also be used as examples for: (1) abstracting concrete execution information and (2) instantiating the framework to collect such information, when future analyses need to be designed and implemented in this framework.

6. RELATED WORK

This section outlines work related to the dynamic analyses presented in this article. The relevant existing work falls into the following five categories: bloat detection, dynamic slicing, dynamic dataflow tracking, dynamic memory leak detection, and heap assertions.

6.1. Bloat Detection

Dufour et al. propose dynamic metrics for Java [Dufour et al. 2003], which provide insights by quantifying runtime bloat. Many memory profiling tools have been developed to take heap snapshots for understanding memory usage (e.g., Java Heap Analyzer Tool [2014]) and to identify objects of suspicious types that consume a large amount of memory (e.g., Quest Software [2011] and ej-technologies GmbH [2011]). However, none of these tools attempts to understand the underlying causes of memory bloat, and thus cannot help programmers pinpoint the problematic areas of the application. Mitchell et al. [2006] 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 [Mitchell and Sevitsky 2007] 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 [Dufour et al. 2007, 2008] and summarizes the shape of these structures. In contrast to the purely dynamic approximation introduced in our work, they employ a blended escape analysis that applies static analysis to a region of dynamically collected calling structure with observed performance problems. By approximating object effective lifetimes, the analysis has been shown useful in classifying the usage of newly created objects in the problematic program region. JOLT [Shankar et al. 2008] 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) [Mitchell et al. 2006; Mitchell and Sevitsky 2007; Dufour et al. 2007, 2008; Han et al. 2012; Xiao et al. 2013; Nistor et al. 2013], and fully automated performance optimization techniques such as the entire field of JIT technology [Arnold et al. 2005] and the research from Shankar et al. [2008]. The work described in this article 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 (i.e., finding the likely bloated regions) and yet use their (human) insights to perform application-specific optimizations. Second, we use different (nonconventional) symptom definitions to identify the bloated regions. For example, the work of copy profiling profiles dataflows 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 Section 1, performance bottlenecks do not necessarily exist in frequently executed regions, and in many cases, they are more related to dataflow rather than control flow. This observation is also the basis of work by Yan et al. [2012], in which the flow of reference values is tracked similarly to our copy-chain profiling technique. Bloat detection has also been attempted by developing dynamic techniques to identify reusable data structures [Xu 2012, 2013b] and cacheable data values [Nguyen and Xu 2013]. In addition, Xu proposes an adaptive technique in Xu [2013a] that can automatically switch container implementations online to improve the performance of programs that make intensive use of object-oriented containers.

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 [Dufour et al. 2007, 2008] is effective at detection of temporary objects while a container profiling technique [Xu and Rountev 2008; Xu 2013a; Shacham et al. 2009] 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.

6.2. Control- and Data-Based Profiling

Lossy compression of profiles has been proposed for space efficiency. These techniques include dynamic dependence profiles [Agrawal and Horgan 1990], control-flow profiles [Ball and Larus 1996], and value profiles [Calder et al. 1997]. 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 Larus [1999] and Zhang and Gupta [2001] studies the compressed representations of control-flow traces. Value predictors [Burtscher and Jeeradit 2003] are proposed to compress value profiles, which can be used to perform various kinds of tasks such as code specialization [Calder et al. 1997], data compression [Zhang and Gupta 2002], value encoding [Yang and Gupta 2002], and value speculation [Lipasti and Shen 1996]. Research from Chilimbi [2001] proposes a technique to compress an address profile, which is used to help prefetch data [Jacobson et al. 1997] and to find cache-conscious data layouts [Rubin et al. 2002]. Zhang and Gupta propose *whole execution traces* [Zhang and Gupta 2004b] 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. [2004] develop a dynamic analysis tool to explore calling context trees in order to find performance bottlenecks. Srinivas and Srinivasan [2005] use a dynamic analysis technique that identifies important program components, also by inspecting calling context trees. Chameleon [Shacham et al. 2009] is a dynamic analysis tool that profiles container behaviors to provide advice as to the choices of appropriate containers. The work in Rayside and Mendel [2007] 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 dataflow, from path profiling [Ball and Larus 1996; Larus 1999; Bond and McKinley 2005; Vaswani et al. 2007] to feedback-directed profiling [Arnold et al. 2005], all to identify heavily executed paths for further optimization. The copy profiling technique described in Section 3 profiles dataflow 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 Agrawal and Horgan [1990]. While both fall in the general category of lossy profile compression, our technique proposes to introduce client analysis semantics into profiling. Hence, our approach loses zero or very little information 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 Agrawal and Horgan [1990] are analysis neutral and it is unclear what kinds of analyses can take advantage of them.

6.3. Dynamic Slicing

Since first being proposed by Korel and Laski [1990], 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 [Tip 1995] and Krinke's thesis [Krinke 2003]. The work by Zhang et al. [2003, 2006a, 2006b] and Zhang and Gupta [2004a, 2004b] has considerably improved the state-of-the-art in dynamic slicing. This work includes, for example, a set of cost-effective dynamic slicing algorithms [Zhang et al. 2003; Zhang and Gupta

2004a], a slice-pruning analysis that computes confidence values for instructions to select those that are most related to errors [Zhang et al. 2006a], a technique that performs online compression of the execution trace [Zhang and Gupta 2004b], and an event-based approach that reduces the cost by focusing on *events* instead of individual instruction instances [Zhang et al. 2006b]. We refer the reader to Zhang's thesis [Zhang 2006] for a detailed description of these techniques. Sridharan et al. propose thin slicing [Sridharan et al. 2007], 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 Agrawal and Horgan [1990], Ball and Larus [1996], Calder et al. [1997], and Zhang and Gupta [2004b], abstract slicing achieves efficiency by introducing client analysis semantics to profiling, establishing a foundation for solving a range of dynamic dataflow 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 Zhang et al. [2006b] 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 predefined 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.

6.4. Dynamic Information-Flow Analysis

Dynamic taint analysis [Haldar et al. 2005; Newsome and Song 2005; Xu et al. 2006; Qin et al. 2006; Clause et al. 2007] tracks input data from untrusted channels to detect potential security attacks. Debugging, testing, and program understanding tools track dynamic dataflow for other specialized purposes (e.g., Masri and Podgurski [2006]). The work in Bond et al. [2007] tracks the origins of undefined values to assist debugging. Research from Masri and Podgurski [2009] proposes to measure the strength of information flows and conducts an empirical study to better understand dynamic information-flow analysis. Work from Chandra and Franz [2007], Nair et al. [2008], and Chandra [2006] 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 Qin et al. [2006] have been developed to reduce its runtime cost. These techniques can also be employed in the future to make our techniques more scalable.

7. 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.

To help developers find these performance problems, this work presents two dynamic techniques, one identifying program regions containing large volumes of data copies and the second finding data structures with high costs and low benefits. We develop a general abstract dynamic slicing framework by extracting the common dataflow tracking functionalities from the two analyses. The framework tracks the propagation of data among heap locations based on an abstract domain specified by the user so that the resulting dependence graph is bounded and does not depend on any dynamic behavior of the program. Both the framework and the analyses have been implemented in J9, IBM's commercial Java Virtual Machine, and have been shown effective in helping a programmer quickly find problematic code that needs to be further inspected and optimized.

Our experimental results clearly demonstrate that the developer's involvement is irreplaceable in the process of performance optimization: many of the refactorings that we have performed either rely on high-level invariants that hold across multiple compilation units or require the deep observation that a specialized version of the computation would be sufficient (and more efficient) under the given context. The insights required to perform such refactorings are way beyond the scope of current automated analyses and transformation techniques. The proposed approach sheds a new light on the optimization of modern large-scale applications: through a combination of automated analyses and manual tuning, we can quickly guide developers to the problematic areas of the application. Fixing the reported problems often leads to large performance gains that cannot be obtained by any compiler transformations.

REFERENCES

H. Agrawal and J. R. Horgan. 1990. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90)*. 246–256.

J. Aldrich, V. Kostadinov, and C. Chambers. 2002. Alias annotations for program understanding. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*. 311–330.

G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. 2004. Finding and removing performance bottlenecks in large systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'04)*. 172–196.

M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. 2005. A survey of adaptive optimization in virtual machines. *Proc. IEEE* 92, 2, 449–466.

M. Arnold, M. Vechev, and E. Yahav. 2008. QVM: An efficient runtime for detecting defects in deployed systems. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08)*. 143–162.

T. Ball and J. Larus. 1996. Efficient path profiling. In *Proceedings of the International Symposium on Microarchitecture (MICRO'96)*. 46–57.

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. Stefanovic, T. Vandrunen, D. Von Dincklage, and B. Wiedermann. 2006. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*. 169–190.

M. D. Bond and K. S. McKinley. 2005. Continuous path and edge profiling. In *Proceedings of the International Symposium on Microarchitecture (MICRO'05)*. 130–140.

M. D. Bond and K. S. McKinley. 2006. Bell: Bit-encoding online memory leak detection. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*. 61–72.

M. D. Bond and K. S. McKinley. 2007. Probabilistic calling context. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*. 97–112.

M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. 2007. Tracking bad apples: Reporting the origin of null and undefined value errors. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*. 405–422.

C. Boyapati, B. Liskov, and L. Shrira. 2003. Ownership types for object encapsulation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*. 213–223.

M. Burtscher and M. Jeeradit. 2003. Compressing extended program traces using value predictors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*. 159–169.

B. Calder, P. Feller, and A. Eustace. 1997. Value profiling. In *Proceedings of the International Symposium on Microarchitecture (MICRO'97)*. 259–269.

D. Chandra. 2006. Information flow analysis and enforcement in java bytecode. Ph.D. thesis, University of California, Irvine, CA.

D. Chandra and M. Franz. 2007. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'07)*. 463–475.

T. M. Chilimbi. 2001. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. 191–202.

D. Clarke and S. Drossopoulou. 2002. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*. 292–310.

J. Clause, W. Li, and A. Orso. 2007. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'07)*. 196–206.

B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. 2003. Dynamic metrics for java. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*. 149–168.

B. Dufour, B. G. Ryder, and G. Sevitsky. 2007. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'07)*. 118–128.

B. Dufour, B. G. Ryder, and G. Sevitsky. 2008. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'08)*. 59–70.

Ej-Technologies Gmbh. 2011. JProfiler. http://www.ej-technologies.com.

V. Haldar, D. Chandra, and M. Franz. 2005. Dynamic taint propagation for java. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'05)*. 303–311.

S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. 2012. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the International Conference on Software Engineering (ICSE'12)*. 145–155.

D. L. Heine and M. S. Lam. 2003. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*. 168–181.

J9 Java Virtual Machine. 2011. The j9 java virtual machine. http://wiki.eclipse.org/J9.

Q. Jacobson, E. Rotenberg, and J. E. Smith. 1997. Path-based next trace prediction. In *Proceedings of the International Symposium on Microarchitecture (MICRO'97)*. 14–23.

Java Development Blog. 2009. Java development blog. cld.blog-city.com.

Java Heap Analyzer Tool. 2014. Java heap analyzer tool (hat). http://hat.dev.java.net.

B. Korel and J. Laski. 1990. Dynamic slicing of computer programs. *J. Syst. Softw.* 13, 3, 187–195.

J. Krinke. 2003. Advanced slicing of sequential and concurrent programs. Ph.D. thesis, University of Passau.

J. Larus. 1999. Whole program paths. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. 259–269.

M. H. Lipasti and J. P. Shen. 1996. Exceeding the dataflow limit via value prediction. In *Proceedings of the International Symposium on Microarchitecture (MICRO'96)*. 226–237.

W. Masri and A. Podgurski. 2006. An empirical study of the strength of information flows in programs. In *Proceedings of the International Workshop on Dynamic Analysis (WODA'06)*. 73–80.

W. Masri and A. Podgurski. 2009. Measuring the strength of information flows in programs. *ACM Trans. Softw. Engin. Methodol.* 19, 2, 1–33.

A. Milanova, A. Rountev, and B. G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Engin. Methodol.* 14, 1, 1–41.

N. Mitchell. 2006. The runtime structure of object ownership. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'06)*. 74–98.

N. Mitchell, E. Schonberg, and G. Sevitsky. 2010. Four trends leading to java runtime bloat. *IEEE Softw.* 27, 1, 56–63.

N. Mitchell and G. Sevitsky. 2007. The causes of bloat, the limits of health. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*. 245–260.

N. Mitchell, G. Sevitsky, and H. Srinivasan. 2006. Modeling runtime behavior in framework-based applications. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'06)*. 429–451.

S. K. Nair, P. N. Simpson, B. Crispo, and A. S. Tanenbaum. 2008. A virtual machine based information flow control system for policy enforcement. *Electron. Notes Theor. Comput. Sci.* 197, 1, 3–16.

N. Nethercote and J. Seward. 2007. How to shadow every byte of memory used by a program. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'07)*. 65–74.

J. Newsome and D. Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS'05)*.

K. Nguyen and G. Xu. 2013. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'13)*. 268–278.

A. Nistor, L. Song, D. Marinov, and S. Lu. 2013. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. 562–571.

F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. 2006. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the International Symposium on Microarchitecture (MICRO'06)*. 135–148.

Quest Software. 2011. JProbe memory debugging. http://www.quest.com/jprobe.

D. Rayside and L. Mendel. 2007. Object ownership profiling: A technique for finding and fixing memory leaks. In *Proceedings of the International Conference on Automated Software Engineering (ASE'07)*. 194–203.

S. Rubin, R. Bodik, and T. Chilimbi. 2002. An efficient profile-analysis framework for data-layout optimizations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*. 140–153.

O. Shacham, M. Vechev, and E. Yahav. 2009. Chameleon: Adaptive selection of collections. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. 408–418.

A. Shankar, M. Arnold, and R. Bodik. 2008. JOLT: Lightweight dynamic analysis and removal of object churn. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08)*. 127–142.

O. Shivers. 1988. Control-flow analysis in scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*. 164–174.

M. Sridharan, S. J. Fink, and R. Bodik. 2007. Thin slicing. In *Proceedings of the ACMSIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. 112–122.

K. Srinivas and H. Srinivasan. 2005. Summarizing application performance from a component perspective. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'05)*. 136–145.

Sun Java Forum. 2014. http://forums.java.net/jive/thread.jspa?messageID=180784.

F. Tip. 1995. A survey of program slicing techniques. *J. Program. Lang.* 3, 121–189.

K. Vaswani, A. V. Nori, and T. M. Chilimbi. 2007. Preferential path profiling: Compactly numbering interesting paths. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. 351–362.

C. Wang and A. Roychoudhury. 2008. Dynamic slicing on java bytecode traces. *ACM Trans. Program. Lang. Syst.* 30, 2, 1–49.

X. Xiao, S. Han, T. Xie, and D. Zhang. 2013. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'13)*. 90–100.

G. Xu. 2011. Analyzing large-scale object-oriented software to find and remove runtime bloat. Ph.D. thesis, The Ohio State University.

G. Xu. 2012. Finding reusable data structures. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. 1017–1034.

G. Xu. 2013a. CoCo: Sound and adaptive replacement of java collections. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'13)*. 1–26.

G. Xu. 2013b. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'13)*. 111–130.

G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. 2010a. Finding low-utility data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. 174–186.

G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. 2009. Go with the flow: Profiling copies to find runtime bloat. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. 419–430.

G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. 2010b. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP Working Conference on the Future of Software Engineering Research (FoSER'10)*. 421–426.

G. Xu and A. Rountev. 2008. Precise memory leak detection for java software using container profiling. In *Proceedings of the International Conference on Software Engineering (ICSE'08)*. 151–160.

W. Xu, S. Bhatkar, and R. Sekar. 2006. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*. 121–136.

D. Yan, G. Xu, and A. Rountev. 2012. Uncovering performance problems in java applications with reference propagation profiling. In *Proceedings of the International Conference on Software Engineering (ICSE'12)*. 134–144.

J. Yang and R. Gupta. 2002. Frequent value locality and its applications. *ACM Trans. Program. Lang. Syst.* 1, 1, 79–105.

X. Zhang. 2006. Fault localization via precise dynamic slicing. Ph.D. thesis, University of Arizona.

X. Zhang, N. Gupta, and R. Gupta. 2006a. Pruning dynamic slices with confidence. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. 169–180.

X. Zhang, N. Gupta, and R. Gupta. 2007. Locating faulty code by multiple points slicing. *Softw. Pract. Exper.* 37, 935–961.

X. Zhang and R. Gupta. 2004a. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*. 94–106.

X. Zhang and R. Gupta. 2004b.Whole execution traces. In *Proceedings of the International Symposium on Microarchitecture (MICRO'04)*. 105–116.

X. Zhang, R. Gupta, and Y. Zhang. 2003. Precise dynamic slicing algorithms. In *Proceedings of the International Conference on Software Engineering (ICSE'03)*. 319–329.

X. Zhang, S. Tallam, and R. Gupta. 2006b. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'06)*. 81–91.

Y. Zhang and R. Gupta. 2001. Timestamped whole program path representation and its applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. 180–190.

Y. Zhang and R. Gupta. 2002. Data compression transformations for dynamically allocated data structures. In *Proceedings of the International Conference on Compiler Construction (CC'02)*. 14–28.

Received January 2013; revised November 2013; accepted December 2013