Uncovering Performance Problems in Java Applications with Reference Propagation Profiling

Dacong Yan[1]   Guoqing Xu[2]   Atanas Rountev[1]

[1]*Department of Computer Science and Engineering, Ohio State University*
[2]*Department of Computer Science, University of California, Irvine*

Abstract—**Many applications suffer from *run-time bloat*: excessive memory usage and work to accomplish simple tasks. Bloat significantly affects scalability and performance, and exposing it requires good diagnostic tools. We present a novel analysis that profiles the run-time execution to help programmers uncover potential performance problems. The key idea of the proposed approach is to track object references, starting from object creation statements, through assignment statements, and eventually statements that perform useful operations. This propagation is abstracted by a representation we refer to as a *reference propagation graph*. This graph provides path information specific to reference producers and their run-time contexts. Several client analyses demonstrate the use of reference propagation profiling to uncover run-time inefficiencies. We also present a study of the properties of reference propagation graphs produced by profiling 36 Java programs. Several cases studies discuss the inefficiencies identified in some of the analyzed programs, as well as the significant improvements obtained after code optimizations.**

I. INTRODUCTION

Various factors can prevent software applications from reaching their performance goals. While hardware advances have been providing free performance benefits for years, these opportunities are gradually getting smaller as software functionality and size grow faster than the hardware capabilities. The extensive use of layers of libraries and frameworks often creates unexpected performance problems. Sometimes there are run-time inefficiencies resulting from common practices (such as creating APIs for general use, and favoring method reuse without specialization).

Many applications suffer from *chronic run-time bloat*—excessive memory usage and run-time work to accomplish simple tasks—that significantly affects scalability and performance. This is a serious problem for software systems, yet their complexity and performance are not well understood by application developers and software researchers. The conclusion from detailed analysis of dozens of real-world applications is that great amounts of work and memory resources are often needed to accomplish very simple tasks [20]. A few redundant objects, calls, and assignments may seem insignificant, but the inefficiencies easily get magnified across abstraction layers, causing significant system slow-downs and soaking up excessive memory resources.

While a modern compiler (such as the just-in-time compiler in a virtual machine) offers sophisticated optimizations, they often are of very limited help in removing bloat. This is because dataflow analyses in a compiler often have small scopes (i.e., they are intraprocedural), which makes it impossible to tackle problems that can cross dozens of calls and even multiple frameworks. In addition, compiler analyses are generally unaware of the domain semantics of the program, while bottlenecks often result from inappropriate design/implementation choices. Finding and fixing these performance problems requires human insight, and thus it is highly desirable to develop diagnostic tools that can expose performance bottlenecks to the developers.

Goals and Motivation. We present a novel tool that profiles the execution to help programmers uncover potential performance problems. The key idea of the proposed approach is to track *object references*, starting from their producers (object creation statements), through assignment statements that propagate the references, eventually reaching statements that use the corresponding objects to perform useful operations. This run-time propagation is abstracted by a representation we refer to as a *reference propagation graph*. This graph contains nodes that represent statements, and edges that correspond to the flow of references between them. The edges are annotated with run-time frequencies. We have designed several client analyses that identify common patterns of bloat by analyzing various graph properties.

The motivation for the proposed reference profiling analysis is threefold. First, the creation and manipulation of objects is at the core of modern object-oriented applications. In cases where the object behavior exhibits suspicious symptoms (e.g., many objects are created by a statement, but only few of them are ever used), it is natural to investigate such symptoms. Second, the specific abstraction of run-time behavior—the reference propagation graph—provides enough information to relate the profiling information back to the relevant source code entities; this makes it easier for a tool user to understand the problematic behavior. Furthermore, the representation maintains separate propagation paths for different sources of object references, and for different contexts of the producers of these references, which allows precise identification of problematic paths. Finally, it is important not only to identify potential performance issues, but also to provide guidance on how to focus the efforts to fix them. Our approach characterizes the complexity of

interprocedural propagation, as well as of interactions with heap data structures, in order to identify the problems that are likely to be easier to explain and eliminate.

Contributions. The contributions of this work are:

- A novel dynamic analysis, *reference propagation profiling*, which tracks the propagation of object references, and produces a reference propagation graph.
- Several client analyses demonstrating the use of reference propagation profiling to uncover inefficiencies.
- An analysis implementation in the Jikes RVM [16].
- A study of the properties of reference propagation graphs produced by profiling 36 Java programs.
- Several cases studies showing inefficiencies identified in some of the analyzed programs, as well as significant improvements after code optimizations.

II. MOTIVATION

Performance inefficiency often comes from extraneous work performed to accomplish a task. One symptom of such inefficiency is the imbalance between the cost of constructing and propagating an object, and the benefit the object can bring to the progress of the application. For example, an object may be propagated to many parts of the code, but only a subset of this affected code actually benefits from having access to the object.

To characterize such imbalance, and to use it to detect potential performance problems, we track three types of run-time events: *object allocation*, *reference assignment*, and *object usage*. In Java, an object is always accessed through references. Such references can be propagated through either stack locations or heap locations. As a form of stack propagation, references can also cross method boundaries via parameter passing or method returns. By writing such references to fields of other objects, they can become accessible to large portions of the application's code.

Such propagation greatly increases the difficulty of manually tracking and understanding the behavior of the object of interest. An automatic *reference propagation profiling tool* can provide significant value and insights needed for performance tuning, especially for complex Java applications. The rest of this section demonstrates through an example how reference propagation profiling can be useful in uncovering performance inefficiencies. The next section describes the formulation and implementation of this dynamic analysis.

Motivating Example. Figure 1 shows a code example simplified and adapted from the `euler` program of the JavaGrande benchmark suite [15]. Class `Vector` represents coordinates in a 2D space. Its `sub` method subtracts one `Vector` from another, and returns the result in a newly-created `Vector` (line 4). The method would be invoked many times during a typical execution. A very large number of objects of type `Vector` would be allocated, since the loops in lines 22–38 would be executed many times. The cost of calls to `sub` (lines 25 and 32) and the object allocations

inside `sub` (line 4) is very high. However, not all of this work is necessary. Note that the object is created solely for the purpose of storing the result of the subtraction. Once the result is retrieved from the object, that object becomes useless and would be deallocated by the garbage collector.

We can reuse a single `Vector` object across multiple calls to `sub`. A variant of `sub` called `sub_rev` is shown at lines 7–10. The new method has an extra parameter to store the result of the subtraction, and the caller of this method is responsible for allocating the object. In this way, the caller would have the flexibility to reuse the object across multiple invocations of `sub_rev`. Specifically, the object returned at the call to `sub` at line 32 is immediately read (lines 36–37) and discarded. A call to `sub_rev` at line 32, with reuse of a single temporary object allocated before the `i` loop, will eliminate the cost of frequent allocation and garbage collection for these short-lived objects.

In cases when the resulting `Vector` is assigned to the heap (line 30) and becomes part of a global data structure, we need to investigate how this heap data structure (`d[i][j]` in this example) is being used. This is necessary to determine whether it is safe to perform the code transformation. We need to track how the object propagates in the memory space through references. For example, the object created at line 4 is propagated through the references `res`, `temp`, and then `d[i][j]`. After the object is assigned to `d[i][j]`, which is a heap location, we need to know whether it is ever read back from the heap. If it is not, we can safely reuse the object; if it is, meaning that there exists an assignment such as `v=d[i][j]`, we have to continue tracking how the local variable `v` is used. In this example, the object is indeed read back from the heap (line 42). Thus, the call at line 25 cannot be replaced with a call to `sub_rev`.

As described later, the reference propagation profiling can provide insights into the behavior of the objects created at line 4. In the actual `euler` benchmark, we observed that a large number of objects created at this allocation site are propagated through the call at line 32, but not any further. In the analysis results, this propagation path is clearly distinguished from the path through the call at line 25, for which there do not exist easy performance optimizations. With the code transformation outlined above, we observed a reduction of 13.3% in running time and 73.3% in number of allocated objects for this benchmark.

Similarly to other dynamic analysis techniques, the conclusions drawn from the propagation graph depend on the quality of the run-time information. In our experiments we run well-defined benchmarks on representative inputs that come with them. In practical use, such representative inputs are necessary for this (or any other) profiling analysis.

III. REFERENCE PROPAGATION PROFILING

Reference Propagation Graph. The propagation of (references to) an object during its lifetime is encoded as a

```
 1 class Vector {
 2    double x, y;
 3    Vector sub(Vector v) {
 4        Vector res = new Vector(x - v.x, y - v.y);
 5        return res;
 6    }
 7    void sub_rev(Vector v, Vector res) {
 8        res.x = this.x - v.x;
 9        res.y = this.y - v.y;
10    }
11    Vector copy() {
12        return new Vector(x, y);
13    }
14 }
15 Vector[][] a = ...;   // input data
16 Vector[][] d = ...;   // intermediate result
17 Vector temp = new Vector();
18 // m, n are typically large numbers
19 int m = readInput();
20 int n = readInput();
```

```
21 void compute() {
22    for (i = 1; i < m; i++) {
23      for (j = 1; j < n; j++) {
24        if (cond1) {
25          temp = a[i+2][j].sub(a[i-1][j]);
26        } else {
27          temp = new Vector(...);
28        }
29        ... // read/write fields of temp
30        d[i][j] = temp;
31        if (cond2) {
32          temp = a[i+1][j].sub(a[i-2][j]);
33        } else {
34          temp = new Vector(...);
35        }
36        d[i][j].x += temp.x;
37        d[i][j].y += temp.y;
38    }}
39 }
40 static void main(String[] args) {
41    compute();
42    ... // access the fields of d[i][j]
43 }
```

Figure 1.   Running example.

reference propagation graph. For illustration, the graph for the example from the previous section is shown in Figure 2.

There are three types of nodes in the graph. A *producer node* represents object allocations. Each producer node has (1) an allocation site ID which encodes the static location of the allocation expression in the source code, and (2) context information obtained when this allocation occurs at run time. The degree of context sensitivity can be tuned as a parameter of the analysis. A *reference assignment node* represents the assignment statements that propagate the objects through references. We distinguish stack-only propagation (at object allocations, between local variables, or due to parameter passing and return values), and propagation between heap and stack (caused by reading or writing instance fields, static fields, or array elements). The nodes are uniquely determined by their static location in the code, and the producer node that reaches them—that is, a single statement in the code can be represented by multiple graph nodes, one per producer of object references. A single *consumer node* represents the usage of objects. If a producer node reaches this node through a certain path, some objects propagated through that path are used. An object is used when (1) it is the receiver of a method call, (2) a field of the object is read or written, (3) it is used as a parameter in a call to a native method, or (4) it is an operand of instanceof, ==, !=, or casting.

There are three types of edges in the graph. An *alloc-assign edge*, between a producer node and a reference assignment node, corresponds to object allocations $ref = new\ X$. A *def-use edge*, connecting two reference assignment nodes, represents the def-use relationship between two reference assignment statements such as $ref = \ldots$ and $\ldots = ref$. A *usage edge*, from a reference assignment node to the consumer node, indicates a def-use relationship between an assignment $ref = \ldots$ and another statement in which the value of ref is used (as described above).

Example. The subgraph related to the allocation at line 4 for the example in Figure 1 is shown in Figure 2. In this

example, a context-insensitive scheme is used to model run-time objects. (Context sensitivity will be discussed later in this section.) Thus, the objects created at line 4 are abstracted solely with the line number, and a node Producer(4) is added to the graph. Immediately after the allocation, the object is assigned to local variable res, so there is a node RefAssign(4,4) and an alloc-assign edge to it. This node is then connected, via a def-use edge, to RefAssign(4,25), which represents the return value of the call at line 25. Here the first label on the node is the ID of the producer node (4) that created the propagated object, and the second label is the line number (25) of the actual statement that does the propagation. Similarly, RefAssign(4,32) and an edge to it are created due to the call at line 32.

In subsequent statements, fields of the object are accessed (lines 29 and 36–37); thus, the two reference assignment nodes are connected to the consumer node. The objects that are propagated along the path through line 25 are later assigned to the heap at line 30 and retrieved back at line 42, so the path is extended accordingly. Such an extension is not performed for the path via 32, since there is no further propagation along that path. The graph is annotated with run-time frequency information for graph edges, similar to the frequencies observed in the actual euler benchmark.

This graph provides the foundation for *reference propagation profiling*. Each edge in the graph is associated with a counter. Whenever a statement is executed at run time, the counter of the corresponding edge is incremented. Both the *structure* of the graph as well as the *edge weights* can be used to identify execution inefficiencies. For example, with this graph, it becomes significantly easier to understand the behavior of run-time objects created at line 4 of Figure 1. First, all paths starting from the producer node contain nodes that go to the consumer node, so it is not possible to simply remove the allocation. In other words, we have to explicitly create the object (or, perhaps, use some form of object
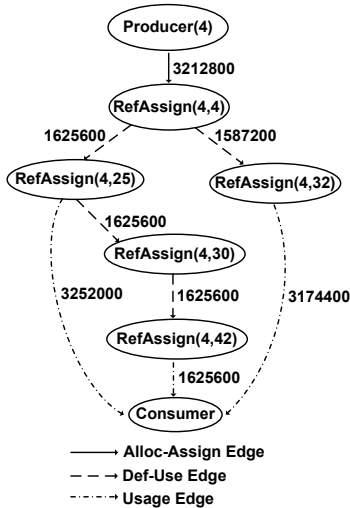
Figure 2. Reference propagation graph for the running example.

inlining [8]). Second, the path through line 32 is very short, does not contain writes to the heap (i.e., the object does not become part of larger heap data structures), and represents a significant volume of reference propagation. Thus, it presents an interesting target for performance analysis and optimization. Third, the path through line 25 is longer, spans three methods, involves propagation through the heap, and therefore is likely to be harder to understand and optimize.

Producer-Specific and Context-Specific Propagation. Each reference assignment node is specific to a particular producer node. For example, the statement at line 30 in the example is represented by RefAssign(4,30), corresponding to the flow of references produced at line 4. This same statement can also propagate the references produced at line 27. A separate node RefAssign(27,30) would represent this propagation. Similarly, line 42 would correspond to two separate nodes, one for each producer. Such per-producer representation allows better precision when characterizing the flow of references. For example, consider the flow from line 30 to line 42. If this flow is *not* distinguished based on the producer, a single frequency would be associated with this pair of statements, making it impossible to attribute the behavior to individual sources of run-time objects.

A producer node is an abstraction of a set of run-time objects, and the choice of this abstraction is an important parameter of the analysis. The simplest abstraction is to use the ID of the allocation site that created the object. However, it is well known that this abstraction can be refined by considering the *context* of the allocation. There are various definitions of context, and they can be easily incorporated in our analysis. For the current implementation, we employ the so called *object-sensitive* abstraction. In this approach, a producer node corresponds to a pair (s_1, s_2) of allocation site IDs. The first ID s_1 is for the site that creates the object. That site is in some method, and the receiver object of that method (i.e., the object to which this refers to when s_1

is executed) is the context of the allocation. Thus, s_2 is the ID of the allocation site that created this receiver. This technique is appropriate for modeling of object-oriented data structures [19] and is currently used in our implementation. The generalization in which a node is a tuple $(s_1, \ldots, s_{k+1})$ (i.e., k-object-sensitivity [19]) can also be easily applied.

Intended Uses. The graph described above can provide useful information for efficient manual investigation of application code. The patterns of reference propagation, across method calls/returns and heap reads/writes, are easy to discern from the structure of the graph. Direct connection with relevant source code locations can be visualized inside a code browsing tool. The frequency information provides insights into the amount of work related to reference propagation, and helps identify hotspots in this propagation.

The graph can also serve as the foundation for a number of client analyses (Section V). The key feature of these approaches is that they automatically identify "suspicious" allocation sites, based on properties of the propagation graph. Furthermore, the graph can provide a characterization of the complexity of propagation patterns and the required program transformations. As a result, programmers or performance tuning experts can focus on parts of the code that not only exhibit run-time inefficiencies, but are also likely to be relatively easy to understand and transform.

Certain aspects of the proposed analysis are similar to information flow analysis (e.g., [13], [25], [34], [26], [5], [4], [2], [3], [23], [18], [3], [23]). However, we record and report not only the source of the transitive run-time dependence, but also the intermediate statements along the dependence chain, as well as (an abstraction of) the actual reference value being propagated. Furthermore, the execution frequencies are collected per-producer-node, which allows unrelated flows through the same statement to be separated.

IV. ANALYSIS IMPLEMENTATION

The analysis is implemented in the Jikes RVM (Research Virtual Machine) version 3.1.1 [16]. The instrumentation is implemented in the optimizing compiler in Jikes. During execution, only this compiler is used, and every method is compiled with it before being executed for the first time.

Shadow Locations. Each memory location containing reference values is associated with a shadow location [24]. Local variables in the compiler IR are represented as symbolic registers. To create shadows for locals, we assign an ID to each symbolic register at "compile time" (actually, at run time when the optimizing compiler is compiling the method), and associate that ID with graph nodes created as the program executes. Shadows of static fields are stored in a global table, and indices into the table are determined by the class loader. Shadows of instance fields are stored in place with originally declared fields, and accessed by offsets from the base objects. The offsets are also determined

during class loading. Array elements are shadowed similarly to static fields, except that per-array tables are used.

In cases when an object is moved by a copying garbage collector, its corresponding shadows should also be moved. This can be done by modifying the garbage collector, but we choose to use a non-moving GC for ease of implementation. This decision does not affect the results of the analysis.

Abstractions for Run-time Entities. The reference propagation graph construction has two components: (1) "compile-time" instrumentation, which happens in the optimizing compiler at run time, and (2) run-time profiling, which builds the graph as the program executes. The instrumentation tags each object with its allocation site information. Specifically, we write an allocation site ID to the header of each object, and the ID can be used to look up the source code location of the allocation site. For a context-sensitive setting, the context information is also recorded in the header. For example, when we use the object-sensitivity representation, the allocation site ID of the receiver object is written to the object header as well. To introduce approximations and tune the overhead, we map the allocation site IDs id of receiver objects into c equivalence classes using a simple mapping function $f(id, c) = id \% c$, where c is a pre-defined value. To achieve full precision (i.e., no approximations), c can be set to the number of allocation site IDs, in which case every equivalence class is a singleton.

Besides allocation site information, we also reserve one extra word in the object header for uses specific to client analyses. For example, such analyses can use one bit to mark whether an interesting event occurs on the object (e.g., whether the object is ever assigned to the heap). Section V discusses how this can be useful for implementing client analyses. The source information of executed reference assignment statements is maintained in a similar way.

Run-time Event Tracking. Each run-time object has a producer node associated with it. To enable fast lookups, producer nodes are stored in a table $prods$, and can be accessed with an index i, a combination of the allocation site IDs of the object, and the receiver object of the surrounding method (a default value 0 is used for a context-insensitive setting). Suppose the two IDs are $allocId$ and $recvId$, and c equivalence classes are used in the object-sensitivity encoding. The index i is computed as $i = allocId \times c + recvId\%c$. Thus, each pair $(allocId, recvId)$ is mapped to an index ranging from 0 to the number of allocation sites multiplied by c. When an object is created, we first look up the table to see whether there is already a producer node at $prods[i]$. If there is one, we increase the frequency of the existing node; otherwise, we create a new producer node, remember it in $prods[i]$, and write the IDs to the header of the newly-created object. In addition, we create a reference assignment node to be the shadow of the variable getting the new object, and connect the producer node with it. If the producer node already exists, the frequency of the edge is incremented.

For a reference assignment `lhs = rhs`, we (1) create a new reference assignment node, (2) remember the node in the shadow of `lhs`, and (3) connect the node stored in the shadow of `rhs` to it. When the edge between the two nodes already exists, its frequency is incremented instead. Parameter passing and method returns are treated as special forms of reference assignments. To pass the shadow information into and retrieve it back from callees, we maintain a per-thread scratch space to temporarily store shadows of parameters and return variables.

As described in Section III, an object can be used at certain statements. For example, when a heap access $v.fld = \ldots$ or $\ldots = v.fld$ is executed, we create a usage edge between the node stored in the shadow of v, and the consumer node. If such an edge already exists, its frequency is incremented.

V. CLIENT ANALYSES

This section describes several client analyses built on top of the reference propagation profiling described earlier. These analyses examine the reference propagation graph and report to programmers a ranked list of suspicious producer nodes that should be examined for performance tuning. The criterion as to what producers are suspicious is defined by individual client analyses. The reported producer nodes are ranked based on the number of times they are instantiated.

In addition, for each reported node, several metrics are computed and provided in the analysis output. The role of these metrics is to estimate the ease with which the propagation starting from this producer can be understood and optimized. Specifically, all reference assignment nodes reachable from a reported suspicious producer node are examined. The number of such reachable nodes that correspond to *calls and returns* is an indication of how widely the references are propagated throughout the calling structure. The higher this number, the more complex the interprocedural propagation, which means that code transformations are likely to be difficult (or impossible). Another metric is the number of reachable nodes that represent *heap reads and writes*. A large number of such nodes indicates that the objects created by the producer node interact in complex ways with heap data structures, which makes their understanding and transformation more challenging.

Not-Assigned-To-Heap (NATH) and Mostly-NATH Analysis. The NATH client analysis detects allocation sites that create many objects, but none of these objects are stored into the heap (i.e., no instance field, static field, or array element ever contains a reference to them). These sites are promising for tuning because the objects created at these sites may be roots of temporary data structures that are expensive to construct. In addition, these objects are typically short-lived, potentially leading to frequent garbage collection. The escape analysis performed by a JIT compiler usually cannot identify such redundancies, because many such objects do escape the methods that created them. Using

the propagation graph, this analysis finds and reports all producer nodes that never reach reference assignment nodes corresponding to assignments from the stack to the heap.

If most of the objects created by a site are NATH, that site is still a good candidate for tuning. We refer to such sites as "mostly-NATH". For example, Line 4 in Figure 1 is a mostly-NATH site, and refactoring it brings significant performance improvements for the `euler` benchmark. Implementing this analysis requires a small extension to tag each object with an assigned-to-heap bit, and store a counter of assigned-to-heap objects in the producer node. The analysis reports any producer node for which the percent of NATH objects exceeded a given threshold. When such sites are reported, the propagation graph can be used to determine the specific paths in the code along which these objects are assigned to the heap (e.g., the path through line 25 in Figure 2). This information provides insights into the run-time object propagation, and eases the task of refactoring the NATH paths (i.e., the paths through which objects are not assigned to the heap).

Analysis of Cost-Benefit Imbalance. In cases when run-time cost is significantly higher than benefits, there could be some redundancies; in terms of objects, there may be excessive allocation or propagation. In general, it is inefficient to allocate a lot of objects but seldom use them. Also, it is suspicious to write an object to the heap significantly more times than it is being read back. This client analysis is a framework to detect such imbalances between cost and benefit, and can be instantiated with different definitions of cost and benefit. For example, we can consider writing an object to the heap as *cost* (because the object had to be created and propagated), and reading it back as *benefit* (since the object was needed by some method). If the ratio between these two is very high (*write-read-imbalance*), it is possible that we do not need that many objects, or the way the program organizes data is problematic. To implement this analysis, we can analyze the reference propagation graph. For a producer node, the cost is the sum of node frequencies for the reachable *stack-to-heap* reference assignment nodes, and the benefit is defined similarly for the *heap-to-stack* ones. The analysis reports all producer nodes for which this ratio is greater than a certain threshold value.

Analysis of Never-Used and Rarely-Used Allocations. One can identify *never-used object allocations* by finding the producer nodes that cannot reach the consumer node; the next section provides several examples of this situation. Or, similarly to the mostly-NATH analysis, one can develop an analysis of *rarely-used allocations*: allocation sites that instantiate many objects, but only a small percentage of these objects are used. As discussed later, our experimental results indicate that never-used objects and never-used allocation sites occur surprisingly often.

Other Potential Uses. There are other performance analyses that can make use of reference propagation profil-ing. For example, such profiling can be used to study *container-related inefficiencies*. The write-read-imbalance objects, those that are written to the heap significantly more times than they are read back, are often written to a heap location which is part of a container data structure. We can locate low-utility containers (many elements are added but only a few are retrieved) by tracking the heap locations to which those imbalanced objects are written. This can be done through inspection of the source code, aided by the path information in the reference propagation graph.

VI. CASE STUDIES

To evaluate the effectiveness of reference propagation profiling, we performed several case studies on Java applications from prior work [28], [32], [33], and found several interesting examples of performance inefficiencies. Due to space limitations, only some of the case studies are discussed. All problems uncovered in these case studies are completely new and have never been reported before. It took us about two days to locate and fix these problems. All programs were new to us. Most of the time was spent on producing a correct fix rather than locating problematic data structures. Such manual tuning is commonly used in practice [20], and without tool support it can be very labor-intensive.

mst This program, from a Java version [17] of the Olden benchmarks, solves the minimum spanning tree (MST) problem [6]. The tool report shows that for an input graph with 1024 nodes, 1047552 objects of type `Integer` are created; the same number of instances is also reported for type `HashEntry`. All of these objects are assigned to the heap, but only half of the `Integer` objects are read back. The large volume of object allocation and the significant cost-benefit imbalance (recall Section V) are highly suspicious. We inspected the code and found that the program uses an adjacency list representation. For each node in the graph, it uses a hash table to store the distances to its adjacent nodes. The distance is represented by an `Integer` object. Thus, for each distance value, it has to create a new `Integer` object. For a graph with 1024 nodes, it creates 1024 hash tables (the tool shows that 1024 arrays of `HashEntry` are created, which corresponds to the 1024 hash tables), and each table has 1023 entries, storing the distances to the other 1023 nodes. So, the program needs $1047552 = 1024 \times 1023$ objects of type `Integer`, and similarly for type `HashEntry`. In addition, the input graphs used by the benchmark are all complete graphs (i.e., each node is connected to each other node).

In general, an adjacency matrix is the preferred representation for dense graphs. Also, for undirected graphs, the distance from node n_1 to n_2 is the same as that from n_2 to n_1, so the way this program stores distances has unnecessary space overheads, which is exactly why only half of the `Integer` objects are read back from the heap, rendering the other half redundant. To confirm our understating on

the tool report without too much refactoring effort, we kept the adjacency list representation, and only slightly changed the code to store and look up distances in an undirected manner. Specifically, for nodes n_1 and n_2, we do not add n_1 to the adjacent list of n_2 anymore, and when we need the distance between them, we look up the adjacency list of n_1, the one with a smaller node ID. This simple change alone reduced running time by 62.5%, and object creation by 39.6% (measured with input graphs of 1024 nodes, and large enough heap sizes). For a fixed heap size of 128MB, the original version can only finish its execution with graphs of at most 1731 nodes, while the modified version can handle 2418 nodes, an input size 39.7% larger. If we refactor the code more aggressively and use an adjacency matrix representation instead, the performance improvement could potentially be even higher.

euler This program is from the Java Grande benchmark suite [15]. The tool shows that the `svect` method of the `Statevector` class creates a large number of `Statevector` objects, while only a small percentage of them are assigned to the heap. After inspecting the code, we found that the program creates temporary objects to serve as the return value of the `svect` method. Once the method finishes its execution, the caller would retrieve the computation result. Afterward, some of the returned objects are stored in an array to be used later, but most of them are not (recall the running example from Figure 1). Method `svect` is invoked inside nested loops that iterate many times, so it is very likely that it will degrade the performance significantly. To solve this problem, we modify the code to make `svect` share one common `Statevector` object to store the result, and make a copy of the objects only when they are to be assigned to the heap. By changing this site alone, we achieved performance improvement of 13.3% in running time and 73.3% in the number of allocated objects.

jflex In the report generated from running JFlex, we found that a large number of `String` and `StringBuffer` objects are created in the `toString` method of a variety of classes. Most of the `String` objects created at these sites are ultimately used to construct the parameter of the static method `Out.debug` which prints out debugging messages when certain debugging flag is turned on. The debugging message is constructed even when the debugging flag is turned off, making the `String` objects redundant. This is confirmed by our report that the `String` objects created at call sites of the `Out.debug` method are never used. To eliminate such redundancies, we change the code to manually inline the calls to `Out.debug` so that no debugging messages would be constructed when the debugging flag is turned off. This modification reduced the running time by 2.9% and the number of created objects by 26.9%.

bloat The analysis of this DaCapo benchmark [7] shows that there is excessive object allocation in method `entrySet` of class `NodeMap`. The program uses `NodeMap`, an inner

class of `Graph`, to ensure there are no duplicate nodes in the graph, and the `NodeMap` uses a `HashMap` for the underlying storage. To implement `entrySet`, one can simply return the entry set of the underlying `HashMap`. However, the program instead returns a newly-created instance of a specialized `AbstractSet` implementation which incorporates sanity checks whenever element removal is to be performed. Specifically, it adds sanity checks to the `remove` and `removeAll` methods of the set object. In addition, in the set implementation, it has a specialized `Iterator` implementation which has similar checks in its `remove` method. These objects are not assigned to the heap, and present an opportunity for optimizations.

The specializations introduced by these objects are useful for debugging purposes. They are needed during the development phase, but redundant after the correctness of the program has been established. To eliminate the redundancy, we removed the checks and used the entry set of the underlying `HashMap` as the return value instead. After the refactoring, we achieved reduction of 10.4% in running time and 11.3% in the number of allocated objects.

chart As shown in the next section, 67.2% of the allocation sites in the `chart` DaCapo benchmark are never-used, meaning that all objects created at such sites are never used. When we examined these sites, we found that the most significant source of never-used objects was a site that creates a large number of `SeriesChangeEvent` objects, but none of them are used. The program creates these objects to notify the listeners that the data series has been changed, and they only contain one single field to represent the source of the event. Since there is no concurrent access to the listener-notification method, we can share one common `SeriesChangeEvent` and update its event source field whenever it is about to be passed to listeners. After this code transformation, we achieved a reduction of 7.7% in running time and 7.8% in the number of allocated objects.

VII. PROPERTIES OF PROPAGATION GRAPHS

This section presents measurements that provide insights into the properties of reference propagation graphs. The measurements are based on a set of 36 programs used in prior work [28], [32], [33], including benchmarks from SPEC JVM98 [30], Java Grande v2.0 (Section 3) [15], a Java version [17] of the Olden benchmarks, and DaCapo 2006-MR2 [7]. The experimental results were obtained on a machine with a 3.4GHz Quad Core Intel i7-2600 processor.

As with similar work on dynamic analysis, a threat to external validity comes from the choice of analyzed programs and their test inputs. We have tried to ameliorate this problem by using a large number of programs from diverse sources, and the representative inputs included with them.

The running time overhead of the analysis is typically around 30–50×. Such overheads are common for similar performance analyses from existing work, and are also ac-

Table I
PROPERTIES OF THE CONTEXT-INSENSITIVE REFERENCE PROPAGATION GRAPHS.

Program	Classes	Methods	Alloc Sites	NATH		Never-Used		WRI Sites		Call/Ret	Write/Read
				Sites	Objs	Sites	Objs	$t = 2$	$t = \infty$		
compress	18	67	22	9	109	0	0	1	1	5.14	3.41
db	9	52	31	16	122	1	30236	1	1	6.87	4.48
jack	53	294	264	107	457449	12	34104	6	5	8.16	7.09
javac	146	779	409	88	1141931	24	254718	41	18	26.69	29.98
jess	140	445	206	36	3359830	6	2087	53	53	8.79	5.93
mpegaudio	49	225	104	7	7	5	212	16	16	5.54	5.13
mtrt	34	196	137	52	4577717	23	465747	4	1	17.6	6.26
search	6	25	3	3	3	0	0	0	0	9.67	0
euler	5	25	19	11	4789005	2	19630	1	1	3.53	18.89
moldyn	5	22	6	2	2	0	0	0	0	5.33	17
montecarlo	14	96	23	15	365202	0	0	0	0	9.48	2.04
JGraytracer	13	55	44	18	51238212	11	4753813	5	5	4.23	3.73
bh	7	49	12	5	126422990	0	0	0	0	15.17	11.33
bisort	3	14	2	1	2	0	0	0	0	14	8.5
em3d	6	16	8	2	2	0	0	0	0	5.88	4.38
health	6	18	17	8	2571333	1	21895	1	1	4.41	2.35
mst	7	31	10	4	1026	0	675444	1	0	6.8	4.8
perimeter	11	42	10	3	3	0	0	0	0	13.8	5.5
power	7	31	16	4	21	0	0	0	0	5.31	3.75
treeadd	3	5	5	3	3	0	0	0	0	4.4	1.2
tsp	3	14	3	1	1	0	4	0	0	9	34.33
voronoi	7	43	12	6	196609	0	0	0	0	19.75	5.75
antlr	109	1256	1151	796	482074	115	152949	89	66	9.05	5.03
bloat	230	1639	969	508	9439879	46	113786	40	25	28.84	10.63
chart	285	1418	1926	732	1174156	1295	578854	243	237	2.96	2.1
eclipse	1210	9558	3694	1485	1802921	688	1678586	363	285	13.08	10.04
fop	663	2661	1246	484	243275	535	42110	109	95	6.34	3.32
hsqldb	112	1012	461	243	67745	60	29735	26	20	7.02	3.66
jython	622	2775	3328	457	5579384	269	807139	1725	1368	11.36	4.82
luindex	96	529	258	141	2167954	46	17888	8	5	6.78	4
lusearch	100	508	228	89	2280855	43	2119431	16	12	7.93	4.09
pmd	377	2175	669	232	11382153	150	2187057	87	65	17.24	7.25
xalan	343	2133	778	149	367534	141	233745	151	134	14.29	7.49
JFlex	35	264	286	74	990370	70	20325	65	65	7.06	4.56
jbb2000	56	476	512	385	7693562	70	10070051	16	12	7.21	3.47
jbb2005	73	601	566	378	916103	69	1642515	22	17	6.95	2.31

ceptable for performance tuning and debugging tasks (rather than for production runs). In our case, the overhead is high because we have to track all instructions involving reference values. The typical memory usage overhead is around 2–3×. Still, we were able to use the tool to study real-world programs, including large applications such as eclipse, and to uncover interesting performance inefficiencies in them. An intriguing possibility for future work is to consider how to reduce the overhead. For example, static analysis can rule out certain uninteresting sites. It may also be possible to apply sampling to track the propagation for only some of the objects created at an allocation site.

Table I shows measurements of the reference propagation graphs obtained in a context-insensitive setting. The first two columns contain the number of loaded non-library classes and the number of executed methods in those classes. The third column shows the number of allocation sites (in these methods) that were executed at least once. These measurements characterize the sizes of the analyzed programs.

The NATH columns show the number of NATH allocation sites and NATH run-time objects. NATH objects are those that are never assigned to the heap. A NATH allocation site creates only NATH objects, but some NATH objects may be created by non-NATH sites. These measurements

indicate the existence of objects that do not interact with the rest of the heap. An interesting observations is that the percentages of NATH allocation sites (the ratios between columns 4 and 3) are typically large for almost all of the programs. This result indicates that Java programs often employ relatively temporary and localized data structures, which presents opportunities for optimizations.

The next two columns report the number of never-used allocation sites and never-used run-time objects. An allocation site is said to be never-used when all of the objects it allocates are never used. These measurements characterize how efficiently the allocated objects are used. If a program creates a large number of objects, but never or seldom uses them, it is certainly inefficient, and improvements may be achievable after code transformations. High percentages of never-used sites (i.e., ratios between columns 6 and 3) provide a symptom of potential bloat, and could lead a programmer or a performance tuning expert to uncover performance problems.

Columns "WRI Sites" show the number of *write-read-imbalance* sites under two different threshold values t. Recall from Section V that for a producer node, a cost-benefit ratio is taken between the sum of node frequencies for the reachable *stack-to-heap* reference assignment nodes (heap

Table II

COMPARISON OF GRAPH SIZES FOR CONTEXT-INSENSITIVE AND FOUR OBJECT-SENSITIVE SETTINGS.

Program	ctx-insen		c=4		c=8		c=16		c=#AllocSites	
	#Nodes	#Edges	#Nodes	#Edges	#Nodes	#Edges	#Nodes	#Edges	#Nodes	#Edges
compress	227	375	227	375	227	375	227	375	227	375
db	405	692	457	768	489	808	511	835	511	835
jack	4383	7793	4699	8241	4874	8468	5117	8775	6459	10332
javac	23307	26126	25632	50256	26366	51575	26706	52097	27407	53205
jess	3340	5507	3602	5933	3611	5947	3735	6148	3840	6289
mpegaudio	1232	2063	1740	2903	1813	2982	1813	2982	1813	2982
mtrt	3727	7913	8872	19447	13079	27990	13508	28654	14347	30484
search	37	73	37	73	37	73	37	73	37	73
euler	451	950	451	950	451	950	451	950	451	950
moldyn	156	277	156	277	156	277	156	277	156	277
montecarlo	312	536	312	536	330	570	333	570	333	570
JGraytracer	425	575	456	610	462	616	462	616	462	616
bh	331	654	343	671	343	671	343	671	343	671
bisort	55	137	55	137	55	137	55	137	55	137
em3d	92	151	92	151	92	151	92	151	92	151
health	146	216	146	216	146	216	146	216	146	216
mst	131	232	131	232	131	232	131	232	131	232
perimeter	214	415	214	415	214	415	214	415	214	415
power	195	287	269	387	269	387	269	387	269	387
treeadd	39	56	51	72	57	76	57	76	57	76
tsp	121	419	121	419	121	419	121	419	121	419
voronoi	327	758	327	758	327	758	327	758	327	758
antlr	18199	36137	18815	37249	19175	37696	19385	38245	19561	38409
bloat	39505	85509	47569	102695	51734	111081	53674	114273	61618	126784
chart	12646	16876	14346	19486	15314	20853	16950	23431	17664	23955
eclipse	92206	179670	106219	201934	107015	206926	109244	209218	110346	212998
fop	14305	22741	15153	23811	15787	24708	16043	24973	16528	25524
hsqldb	5873	10601	6649	11656	7104	12464	7280	12686	7973	13819
jython	58835	96775	60495	99011	61774	100825	63221	103286	67127	107407
luindex	3226	5727	3441	6024	3480	6084	3511	6096	3558	6158
lusearch	3102	5158	3469	5660	3601	5888	3607	5956	3730	6103
pmd	17469	32074	18046	32984	18126	32996	18409	33431	19288	34749
xalan	18056	32584	19645	35293	20458	36887	20437	36805	21486	38397
JFlex	3860	5887	4204	6510	4304	6668	4313	6676	4421	6861
jbb2000	6451	11596	7457	13480	7957	14491	8707	16042	8729	16287
jbb2005	6198	10577	6870	11627	7037	11830	7245	12082	7559	12526

writes), and that of *heap-to-stack* ones (heap reads). An allocation site is counted when the cost-benefit ratio of its corresponding producer node is greater than the threshold. The sites without any heap writes (i.e., NATH sites) are already identified by the NATH analysis, and are not considered for the WRI analysis. Threshold $t = 2$ selects sites whose allocated objects are written to the heap at least twice as many times as they are read back from the heap. The special threshold value $t = \infty$ covers the cases when the objects are only written to the heap but never read back. Larger numbers of WRI sites indicate higher degrees of wasted heap propagation, which could potentially be eliminated by code transformations.

The last two columns show the average numbers of (1) method invocation nodes (calls and returns), and (2) heap propagation nodes (heap writes and reads) reachable from a producer node. They characterize the complexity of the reference propagation, from the perspective of inter-procedural control-flow and heap data structure interactions. If the number of method invocation nodes is high, objects are propagated through large portions of the call structure, and the propagation is likely to be more difficult to understand and refactor. The same is true for the average number of heap propagation nodes, which indicate points of interaction with

other heap objects. By presenting to the programmer these two metrics for a suspicious allocation site, our analysis can help to distinguish objects that are relatively easy to understand from objects whose behavior may be too complex to be worth further investigation.

Table II shows the size of the reference propagation graph (number of nodes and number of edges) under different context-sensitivity abstractions. The first two columns show the measurements for the context-insensitive setting, followed by object-sensitive settings with different numbers c of equivalence classes in the context encoding ($c = 4, 8, 16$). The last column shows the measurements under a full object-sensitivity setting, where each receiver object ID belongs to a separate equivalence class (Section IV).

As the degree of context-sensitivity increases, graph size typically remains about the same or grows slightly. With more precise context information, we can better distinguish the run-time allocations, and more producer nodes can be created. Such a graph presents a more precise and detailed picture: instead of describing the "per producer" propagation, it provides insights into the "per producer, per context" behavior of objects. Although in principle the cost of collecting this more precise information can be high (in terms of running time and memory consumption), in

reality this does not appear to be the case: context-sensitive information can be collected with little additional overhead. For the programs we studied, the average running time overhead when using the fully context-sensitive encoding is 1.4% (compared to using the context-insensitive one). For the memory usage overhead, the increase is 3.5%. This observation indicates that future work could investigate even more precise context-sensitivity abstractions.

VIII. Related Work

Detection of Run-time Bloat. A number of tools have been proposed to quantify various symptoms of bloat (e.g., [9], [14], [27], [12]), without providing insights into the reasons why this bloat occurs. Mitchell *et al.* [22] consider the transformations of logical data in order to explain run-time behavior and to assist a programmer in deciding whether execution inefficiencies exist. The approach in this work is not automated. Their follow-up work [21] focuses on deciding whether data structures have unnecessarily high memory consumption. Work by Dufour *et al.* analyzes the use and shape of temporary data structures [10], [11]. Their approach is based on a blended analysis, where a run-time call graph is collected and a static analysis is applied based on this graph. JOLT [29] 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 method inlining inside a just-in-time compiler.

In general, existing bloat detection work can be classified into two major categories: manual tuning methods (i.e., mostly based on measurements of bloat) [22], [21], [10], [11], and fully automated optimization techniques such as the entire field of JIT technology [1] and the research from [29]. We provide analyses to support manual tuning, guiding programmers to code where bloat is likely to exist, and then allowing human experts to perform the code modification. By doing so, we hope to help the programmers quickly get through the hardest part of the tuning process—finding the likely bloated regions—and yet use their (human) insights to perform application-specific optimizations.

Our previous work on dynamic analysis for bloat detection includes techniques that focus on different bloat patterns (such as excessive copy activities [32] and inefficient use of data structures [31]) to help programmers identify performance bottlenecks. The previous work closest to the technique proposed in this paper is the profiling of copy activities from [32]. While both techniques track the flow of data, the proposed reference propagation analysis is more general and powerful in several aspects. First, the analysis records much more detailed information on how objects propagate, including information that connects the propagation with the corresponding source code statements. This level of detail makes it easier to explain and fix a performance problem. Second, the abstractions used to represent the propagation are more powerful, since they are specific to a producer

of references, while the profiling in [32] merges the flow from multiple producers. Third, our work reports potential problems together with indicators of the likely difficulty of explaining and eliminating them. This approach is based on properties of the propagation that capture the complexity of interprocedural control-flow and of interactions with heap data structures.

Information Flow Analysis. Dynamic taint analysis [13], [25], [34], [26], [5] is a popular technique for tracking inputs from untrusted sources to detect potential security attacks. Debugging, testing, and program understanding tools track dynamic data flow for a number of purposes (e.g., for detecting illegal memory accesses [4] and for tracking the origins of undefined values [2]). Research from [18] proposes to measure the strength of information flows and conducts an empirical study to better understand dynamic information flow analysis. Work from [3], [23] describes approaches to enforcing information flow analysis in Java virtual machines. Our analyses combine information flow tracking and profiling to efficiently form producer-specific and context-specific execution representations that are necessary for the client analyses to identify inefficiencies.

IX. Conclusions

This paper presents a novel reference propagation profiling tool used to uncover performance problems in Java applications. It tracks the propagation of object references and encodes the results in a reference propagation graph. The information stored in the graph is specific to producers of object references (and the run-time contexts of these producers). Several client analyses are developed to analyze these graphs, and to report to developers a ranked list of suspicious allocation sites, annotated with information about the likely ease of performing transformations for them. Interesting performance inefficiency patterns are discovered by these clients. The properties of the reference propagation graphs are studied on 36 Java programs. The experimental results show that the degree of context-sensitive precision can be increased without significant additional costs. The running time reduction achieved by optimizing suspicious allocation sites can be significant, as demonstrated in several case studies. These findings suggest that our approach is a good foundation for implementing various client analyses to uncover reference-propagation performance problems, and to explain these problems to the developers.

REFERENCES

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

[2] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 405–422, 2007.

[3] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *Annual Computer Security Applications Conf.*, pages 463–475, 2007.

[4] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic. Effective memory protection using dynamic tainting. In *Int. Conf. Automated Software Engineering*, pages 283–292, 2007.

[5] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ACM SIGSOFT Int. Symp. Software Testing and Analysis*, pages 196–206, 2007.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms, 2nd ed., 2001.

[7] DaCapo Benchmarks, www.dacapo-bench.org.

[8] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 345–357, 2000.

[9] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 149–168, 2003.

[10] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ACM SIGSOFT Int. Symp. Software Testing and Analysis*, pages 118–128, 2007.

[11] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, pages 59–70, 2008.

[12] ej technologies. *JProfiler*. www.ej-technologies.com.

[13] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Annual Computer Security Applications Conf.*, pages 303–311, 2005.

[14] *Java Heap Analyzer Tool (HAT)*. hat.dev.java.net.

[15] Java Grande Forum Benchmark Suite, www.epcc.ed.ac.uk/research/java-grande.

[16] Jikes RVM, jikesrvm.org.

[17] M. Marron, M. Mendez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2008.

[18] W. Masri and A. Podgurski. Measuring the strength of information flows in programs. *ACM Trans. Software Engineering and Methodology*, 19(2):1–33, 2009.

[19] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Software Engineering and Methodology*, 14(1):1–41, 2005.

[20] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1):56–63, 2010.

[21] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. *ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 245–260, 2007.

[22] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *European Conf. Object-Oriented Programming*, pages 429–451, 2006.

[23] S. K. Nair, P. N. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, 2008.

[24] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Int. Conf. Virtual Execution Environments*, pages 65–74, 2007.

[25] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed Systems Security Symp.*, 2005.

[26] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Int. Symp. Microarchitecture*, pages 135–148, 2006.

[27] Quest Software. *JProbe Performance Debugging*. www.quest.com/jprobe.

[28] A. Rountev, K. Van Valkenburgh, D. Yan, and P. Sadayappan. Understanding parallelism-inhibiting dependences in sequential Java programs. In *Int. Conf. Software Maintainance*, page 9, 2010.

[29] A. Shankar, M. Arnold, and R. Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. In *ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 127–142, 2008.

[30] SPEC JVM98 Benchmarks, www.spec.org/jvm98.

[31] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 174–186, 2010.

[32] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 419–430, 2009.

[33] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *Int. Conf. Software Engineering*, pages 151–160, 2008.

[34] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, pages 121–136, 2006.