# PerfBlower: Quickly Detecting Memory-Related Performance Problems via Amplification

**Lu Fang[1], Liang Dou*[2], and Guoqing Xu[1]**

1 **University of California, Irvine, USA**
{lfang3, guoqingx}@ics.uci.edu
2 **East China Normal University, China**
ldou@cs.ecnu.edu.cn

## Abstract

Performance problems in managed languages are extremely difficult to find. Despite many efforts to find those problems, most existing work focuses on how to debug a user-provided test execution in which performance problems already manifest. It remains largely unknown how to effectively find performance bugs before software release. As a result, performance bugs often escape to production runs, hurting software reliability and user experience. This paper describes PerfBlower, a general performance testing framework that allows developers to quickly test Java programs to find *memory-related* performance problems. PerfBlower provides (1) a novel specification language ISL to describe a general class of performance problems that have observable symptoms; (2) an automated test oracle via *virtual amplification*; and (3) precise reference-path-based diagnostic information via *object mirroring*. Using this framework, we have amplified three different types of problems. Our experimental results demonstrate that (1) ISL is expressive enough to describe various memory-related performance problems; (2) PerfBlower successfully distinguishes executions with and without problems; *8 unknown problems* are quickly discovered under small workloads; and (3) PerfBlower outperforms existing detectors and does not miss any bugs studied before in the literature.

**1998 ACM Subject Classification** D.3.4 [Programming Languages] Processors – Memory management, optimization, run-time environments, F.3.2 [Logics and Meaning of Programs] Semantics of Programming Languages – Program analysis, D.2.5 [Software Engineering] Testing and Debugging – Debugging aids

**Keywords and phrases** Performance bugs, memory problems, managed languages, garbage collection

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2015.296

## 1 Introduction

Performance problems commonly exist in real-world applications. Much evidence [13, 35, 19] shows that seemingly insignificant performance problems can lead to severe scalability reductions and even financial losses. Performance problems are notoriously difficult to find and fix because visible performance degradation is often an accumulation of the effects of a great number of invisible problems that scatter all over the program [35]. These problems all seem harmless unless they are triggered together under a large workload, wherein their effects multiply and manifest, preventing the program from reaching its performance/scalability goals.

* Work was done while the author visited UC Irvine during 2013–2014.

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 296–320

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Most existing research efforts that attempt to find performance problems are postmortem debugging techniques [13, 22, 29, 12, 34, 42], focused on detecting the root cause of a performance problem that already manifests in a user-provided test execution or actual production run. However, in performance testing, developers often execute a large number of tests to achieve coverage, and, thus, it is unrealistic to ask developers to run a detector on each test and check if each generated report (often with hundreds of warnings) contains true performance bugs. This is a task even harder than finding a needle in a haystack because whether there exists a needle is unknown in the first place, let alone how to search the haystack to find it. Symptom-based detectors are more suitable for *postmortem debugging* when a performance bug already manifests in a user-provided test – developers are much more willing to spend their time digging into reported warnings if they already have some initial clue on the bug.

In addition to these research efforts, many open-source frameworks (e.g., [15, 18, 30, 31, 17]) have been designed to support performance testing, e.g., by generation of large tests/loads for triggering performance bugs[1]. However, these existing performance testing frameworks suffer from the following three major drawbacks: (1) the lack of a general specification to describe performance problems; (2) the lack of effective test oracles that can capture invisible performance problems under small (testing) workloads; checks on the absolute time and/or memory are often very subjective and cannot reveal small performance bugs before they accumulate; and (3) the lack of effective debugging support that can help developers find the root cause when a bug manifests. As a result, performance bugs still escape to production runs, hurt user experience, degrade system throughput, and waste computational resources [6]. They affect even well tested software such as Windows 7's Windows Explorer, which had several high-impact performance bugs that escaped detection for long periods of time [12].

**Our target.** In this paper, we propose a general performance testing framework, called PerfBlower, that can "blow up" the effect of small performance problems so that they can be easily captured during testing. Finding general performance problems is infeasible, as it requires finding an alternative, more efficient computation in a possibly infinite solution space. We first narrow our focus onto a class of performance problems that have *observable symptoms*; their symptoms can be expressed by logical statements over a history of heap updates. These problems include, to name a few, memory leaks, inefficiently-used collections, unused return values, or loop-invariant data structures. One common axis that centers around these problems is that they manifest in the form of inefficient data usage, and their symptoms can be identified by capturing and comparing heap snapshots. Many performance problems are caused by redundant computations; although they are not directly data-related, data inefficiencies can still be seen as a result of redundant computations. For example, one problem in Mozilla studied in [13] is due to the over-general implementation of an API—the `draw` method performs heavy-weight computations to draw high-quality images while the client only needs transparent ones. Even if the proposed technique cannot directly amplify redundant computations, most fields of the resulting image object are never used; these redundant data can be captured and penalized.

**Contribution 1: A performance specification language.** To provide developers a general way to define symptoms, we propose a simple, event-based language, referred to as ISL

[1] We use "performance bug" and "performance problem" interchangeably.

(i.e., acronym for *instrumentation specification language*). ISL is a domain-specific language tailored for describing symptoms of performance problems on a JVM. Since performance problems cannot be expressed using logical assertions, an important challenge in the design of ISL is what to do when a symptom is seen. Unlike a regular detector that immediately reports a problem, ISL provides a pair of commands *amplify* and *deamplify*, which allow developers to add *memory penalties* to an object upon observing a symptom and remove penalties when a *counter-evidence* is seen. Symptoms are only *indicators* of performance problems, not *definitive evidence*. If the specified symptom does not point to a real problem (e.g., the symptom stays for a while and then disappears), the developer calls *deamplify* to remove penalties. The symptom specified in ISL is periodically checked (often during garbage collection): objects that keep satisfying the specification will keep getting penalized; all existing penalties for an object are removed at the moment we observe that the object does not satisfy the specification.

**Contribution 2: Providing test oracle via virtual amplification.** Our amplification is on a *per-object* basis: a memory penalty is created and associated with each object that satisfies the symptom specified in ISL (i.e., at the moment the object becomes "suspicious").

Instead of requesting actual memory as penalties, which would incur significant space overhead, we create *virtual penalties* by maintaining a *penalty counter* (PC) inside each object to track the size of the penalty created for the object. During each garbage collection (GC), we identify the real heap usage of the program and then compute a *virtual heap consumption* by adding up the PC for each object and the real heap consumption. The virtual heap consumption is then compared to the real heap consumption to compute a *virtual space overhead* (VSO). VSO provides an automated test oracle: our experimental results show that the overall VSOs for benchmarks with and without real performance problems are, respectively, 20+ times and 1.5 times. The gap is sufficiently large so that test runs triggering bugs can be easily identified.

**Contribution 3: Providing precise diagnostic information via object mirroring.** PERFBLOWER is not only able to amplify the effects of performance problems; it can also provide diagnostic information to help developers find root causes of these problems. Much evidence [36, 41] shows object reference path is an important piece of information that reveals both the calling context and the data structure in which a "suspicious" object is created. We propose a novel algorithm (Section 4) that *incrementally* builds a *mirror object chain* that reflects the major reference path in the object graph leading to $o$. This chain records the source code information of each object on the reference path; identifying and reporting the reference path for $o$ gets reduced to traversing *forward* $o$'s *mirror chain*, a task significantly easier than performing a *backward* traversal on the object graph which has only unidirectional edges.

Our incremental algorithm enables an important feature of the framework: *the completeness of diagnostic information provided for a "suspicious" object is proportional to the degree of its suspiciousness* (i.e., how long it gets penalized). Every time the object is penalized, the algorithm presented in Section 4 incrementally records one additional level of the reference path for the object. While a regular detector can also provide diagnostic information, it maintains a tracking space of the same size and records the same amount of information for all objects. Developers are very often interested in only a few (top) warnings in a diagnostic report; hence, it is much better to record more diagnostic information for top warnings than those that are low down on the list. Our algorithm dynamically determines the metadata

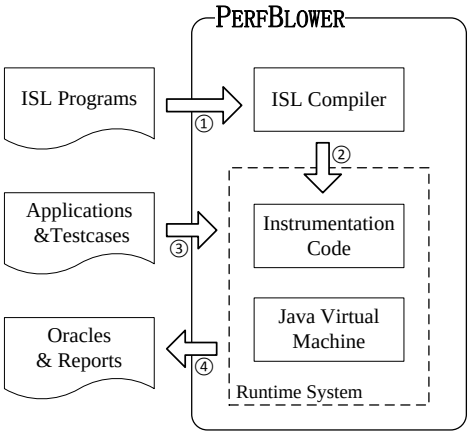



**Figure 1** The architecture of PerfBlower.

space size based on how "interesting" an object is, making it possible to *prioritize* object tracking. For example, the reference path associated with the top object in `xalan` (shown in Section 5.2) reported by our leak amplifier has 10 nodes; all of the nodes on this long path are necessary for us to understand the cause of the leak. A regular detector can never afford to store a 10-node path for all objects in the heap.

We have implemented PerfBlower based on the 3.1.3 version of the Jikes Research Virtual Machine (RVM). Using the framework, we have amplified three different types of performance problems (i.e., memory leaks, under-utilized containers, and over-populated containers) on a set of real-world applications; our experimental results show that even under small workloads, there is a very clear distinction between the VSOs for executions with and without problems, making it particularly easy for developers to write simple space assertions and only inspect programs that have assertion failures. In fact, we have manually inspected each of our benchmarks for which PerfBlower reports a high VSO, and found a total of *8 unknown problems*.

We have compared the quality of leak reports between PerfBlower and an existing leak detector Sleigh [3] executed under the same (small) workloads: PerfBlower reported 7 leaks with no false positive and very detailed diagnostic information, while Sleigh reported 5 leaks with 1 false positive, and generated zero diagnostic information for 3 programs and very high-level information for the other 2 programs. We additionally performed an exhaustive study of the 14 performance bugs reported in the literature: PerfBlower found all but 4 bugs; for these bugs, we either could not run the program or did not have a triggering test case. These promising results clearly show that PerfBlower is useful in quickly building a large number of checkers that can effectively find performance bugs before they manifest.

## 2 PerfBlower Overview

**The PerfBlower architecture.** Figure 1 depicts the architecture of PerfBlower. It has two major components: an ISL compiler and a runtime system. The compiler parses an ISL program provided by the developer and automatically generates instrumentation code for a JVM. The JVM is built and a program to be tested is then executed on the modified VM. During execution, the runtime system monitors the behavior of the program, checks symptom specifications, and performs amplification and deamplification. PerfBlower reports detailed diagnostic information for programs with high virtual overhead.

```
Context ArrayContext {
    sequence = "*.main,*";
    type = "Object[]";
}

Context TrackingContext {
    sequence = "*.main,*";
    path = ArrayContext;
    type = "String";
}

History UseHistory {
    type = "boolean";
    size = UP;//User Parameter
}

Partition P {
    kind = all;
    history = UseHistory;
}
```

```
TObject MyObj{
    include = TrackingContext;
    partition =  P;
    instance boolean useFlag =
        false; //Instance Field
}

Event on_rw(Object o, Field f,
            Word w1, Word w2){
    o.useFlag = true;
    deamplify(o);
}

Event on_reachedOnce(Object o){
    UseHistory h = getHistory(o);
    h.update(o.useFlag);
    if(h.isFull()
        && !h.contains(true)){
         amplify(o);
    }
}
```

**Figure 2** An ISL program amplifying memory leaks caused by unnecessary references from arrays.

**ISL overview.** As the first step, the developer writes an ISL program to describe (1) symptoms of a performance problem (for which amplification is needed) and (2) counter-evidence that shows what is being tracked is not a performance problem (for which deamplification is needed). ISL explicitly models heap partitioning, history of heap updates, as well as collaborations between the collector and the mutator, allowing developers to easily write simple Java-like code to amplify and deamplify performance problems.

To illustrate, Figure 2 shows a sample ISL program that describes the symptom of memory leaks caused by unnecessary strings held in object arrays, as well as how their effects can be amplified. A typical ISL description consists of a set of constructs that specifies how heap updates can be tracked and where the tracking information should be stored, as well as a set of *events* that uses imperative constructs in regular Java to define actions to be taken when the events are triggered.

**Context.** Many dynamic analyses do not need to track every single detail of the execution. For different performance problems, the developer may focus on different aspects of the execution, such as the behavior of an object when it is created under a certain calling context (e.g., control-related) or referenced by a certain data structure (e.g., data-related). A context construct is designed for the developer to express the scope of the objects of interest. Context has three properties: sequence, type, and path, each of which defines a constraint in a different aspect. They can be used to specify, respectively, the calling context, the type, and the reference path for the objects to be tracked. For example, context `ArrayContext` narrows the scope of the heap to objects of type `Object[]` created under calling contexts that match string "*.main, *", where * is used as a wildcard. This string matches all call chains that start with an invocation of method "*.main".

Next, context `ArrayContext` is used to define the reference path in a new context `TrackingContext`, which describes the constraint that we are only interested in the `String` objects that are (1) created under context "*.main, *" (i.e., defined by sequence), and (2) reachable directly from the array objects created under the same calling context in the object graph (i.e., defined by path). Note that this context unifies two orthogonal object naming schemes (i.e., call-chain-based and access-path-based), providing much flexibility for the developer to narrow down interesting objects.

**History.** Since we target heap-related performance problems, their symptoms can often be identified by comparing old and new values in the tracked objects. A History construct is such an abstraction that models the heap update history (i.e., in an execution window) that needs to be tracked on the objects of interest. A history has two important properties: the type of a history element (i.e., type) and the length of the execution window tracked by the history (i.e., size). The element type has to be a primitive type because tracking a program execution often requires collaborative work between the mutator and the collector; creating tracking objects during a GC would not be allowed. The length of the execution window specifies how many user-defined state updates can be recorded in the history. The recorded updates will be used to determine whether a symptom is seen. A history has to be attached to a partition, allowing for the tracking of heap updates at different heap abstraction levels. We provide a few functions to manipulate a history; these functions will be discussed shortly.

**Partition.** Different analyses may need to collect tracking information at different abstraction levels. For example, analyses that report information regarding allocation sites can maintain a piece of tracking information for each allocation site, while analyses designed to identify type-state bugs have to keep per-object tracking information. A Partition construct can be used to define the partitioning of the heap needed by an analysis. Properties kind and history specify, respectively, how the heap should be partitioned and what history should be attached to each heap partition. In other words, one history instance (as defined in History) will be created and maintained for each partition of the heap defined by kind. kind can have five different values—all, context[$i$], alloc, type, and single—which specify heap abstractions with increasing levels of granularity.

In particular, single means that all heap objects will be in the same partition, and therefore will share the same tracking information (i.e., a history instance). type and alloc specify that our runtime system will create one history instance per Java type and per allocation site, respectively; context[$i$] informs the runtime system to create one history instance for each allocation site executed under a distinct calling context; since each allocation site can have an extremely large number of distinct contexts, we allow the developer to define a (statically-fixed) integer number $i$ to limit the number of contexts for each allocation site; a hash-based encoding function proposed by Bond and McKinley [4] is used to map a full context to a number in [0, $i$ - 1]; this option can be used to enable context-sensitive tracking of certain heap properties; finally, all means that one history instance will be created per heap object.

**TObject.** TObject defines the type of objects to be tracked, using two properties, include and partition, that specify the tracking context and the heap partitioning, respectively. In our example, the system tracks objects under the context `TrackingContext` and creates a history for each tracked object. In addition, each tracked object has an instance field `useFlag`, which stores object-local state information necessary for identifying the symptom. The keyword `instance` can be used to declare (primitive-typed) per-object metadata information.

**Event.** At the center of an ISL program is a set of event constructs that define what to do when important events occur on tracked objects. ISL supports seven different events: alloc, read, write, rw, call, reached, and reachedOnce. The first five are regular mutator events while the last two are GC events. reached is triggered every time an object is reached during a GC object graph traversal. Since an object may be reached multiple times (through different references), we use reachedOnce to define actions that need to be taken only once

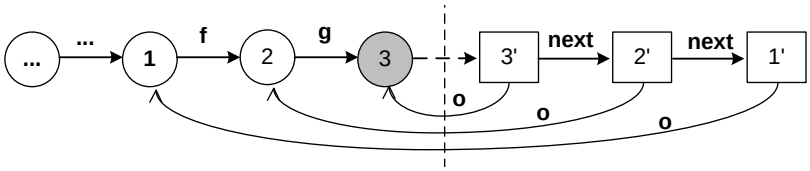



**Figure 3** An example mirror path.

on the object during each GC—reachedOnce is triggered only at the first time the object is reached. An event construct takes a few parameters exposing the related run-time values at the event. In Figure 2, the parameters of the `rw` event include the base object, the field being accessed, and the value being read/written. Word is a special type representing a 32-bit value (regardless of its Java type). If this event reads/writes a 64-bit value, this value will be broken into two words $w_1$ and $w_2$, representing the high and the low 32 bits, respectively.

In our example, the events `rw` and `reachedOnce` specify the symptom of a memory leak and when to amplify its effect. Once a tracked object $o$ is used (i.e., read or written), we set its `useFlag` true. Since we see a use of the object, we call a library method `deamplify` to cancel all the space penalty previously added to $o$. When $o$ is reached in a `reachedOnce` event, we update the history associated with $o$'s partition with $o.useFlag$. Since the history is defined to track only the most recent UP[2] updates, the `update` operation will add the current boolean value and discard the oldest value from the history if it is full. Finally, if the history has a full record of UP updates (i.e., `isFull` returns `true`) and the record does not contain any `true` value, object $o$ is stale and thus we add space penalty to amplify the staleness effect by calling method `amplify`.

**Virtual amplification and deamplification.** Amplification of a performance problem is done on the objects that satisfy the symptom specification of the problem. Method amplify($o$) takes an object $o$ as input and increases the PC in $o$ by the size of the object. When deamplify is executed on $o$, we set $o$'s PC back to zero. During each GC, we modify the reachability analysis to compute the sum of the size of the live heap ($S$) and the PCs of all live objects ($P$). We define VSO to be $(P + S)/S$, which simulates the space overhead of an execution had the real memory penalty been used. VSO is a metric that measures the severity of performance problems relative to the amount of memory available to a program—the same problem is more serious if the program is run with a small heap than with a large heap. Since $P$ has a time component (e.g., objects that keep satisfying the symptom specification will make $P$ keep growing), a program may have increasingly large VSOs as it executes. Eventually, the VSOs computed at all GCs are compared and the maximal VSO is reported to indicate the severity of the performance problems.

**Providing diagnostic information by mirroring reference paths.** When a test fails, it is important to provide highly-relevant diagnostic information that can help developers quickly find the root cause and fix the problem. Evidence [36, 41] shows that heap reference paths leading to suspicious objects are very important information as they reveal calling contexts and data structures containing these objects. However, it can be quite expensive for the runtime system to identify and report reference paths for objects. Although the GC traverses all live heap objects, path information is not easy to obtain—to be scalable,

[2] UP is an acronym for user parameter.

the reachability analysis in the GC uses a breadth-first algorithm that does not record any backward information. In addition, reporting a reference path requires recording the source code information (e.g., allocation site) of each object on the path, which can introduce significant runtime overhead.

In PerfBlower, we solve the problem by building a *mirror object chain* for $o$ that reflects the major reference path leading to $o$. Figure 3 shows an example mirror path. Each object in the original Java heap is annotated with an integer $i$ and its corresponding mirror object is annotated with $i$'. To report object 3's reference path, for instance, we only need to traverse *forward* its mirror chain (as apposed to traversing *backward* on the graph) and print information contained in each mirror object. This eliminates the need to either modify the GC algorithm or increase the size of a regular object to store source code information. PerfBlower tracks only one single reference path for each suspicious object. We find it to be sufficient in most cases. A more detailed discussion can be found at the end of Section 4. After the execution finishes, allocation sites are ranked based on the total size of the memory penalties added to their objects; their related reference paths are reported as well.

## 3 ISL: A Systematic Way to Describe Symptoms

**ISL syntax and semantics.** In this section, we briefly describe the core parts of the syntax and semantics of ISL. An ISL program is a set of contexts, histories, partitions, tracked objects, and events. The path field of a context $c_1$ is defined by another context $c_2$, which specifies that the objects constrained by $c_1$ have to be directly reachable from the objects constrained by $c_2$ on the object graph. If we wish to specify transitive reachability, e.g., via a path of $n$ nodes, we can define $n$ contexts from $c_2$ to $c_{n+1}$, each constraining a node on the path. History supports the following seven operations: int length(), boolean isFull(), void update(T), T get(int), boolean contains(T), boolean containsSameValue(), and History subHistory(int, int). The first five operations are defined in expected ways. containsSameValue returns true if all elements of the history have the same value; subHistory converts a sub-region of the current history into another history instance.

PerfBlower supports simultaneous checking of multiple problems. This is done by assigning a unique ID to each ISL program specifying a problem. The ID will be passed (implicitly) as a parameter to `amplify`($o$) – it is written into each mirror object created for $o$ so that when the mirror path for $o$ is traversed and printed, the description of the problem to which $o$ is related is reported as well.

**Example ISL programs.** In addition to the memory leak example given in Section 2, Figure 4, Figure 5, and Figure 6 show, respectively, the simplified ISL descriptions for amplifying three other types of performance problems: (a) extensive creation of objects with invariant data contents, (b) under-utilized Java maps, and (c) unused return values. While there are multiple ways to find these problems, we show how a unified amplification-based approach can be used to detect all of them.

**Allocation sites creating invariant objects.** The symptom is that certain allocation sites keep creating objects with identical data values. The goal of the ISL program is to identify these allocation sites and then penalize their objects. The partitioning is based on allocation site, which means all objects created by the same allocation site share one history instance. We define the history as a list of `long` values (as `long` can express values of any type) over an execution window of UP updates. Here we are interested in values stored in each primitive-

```
History UpdateHistory {
  type = "long";
  size = UP * MAX_NUM_FIELDS;
       //UP means User Parameter
}

TObject MyObj { partition =  P;}

Event on_reachedOnce(Object o){
  UpdateHistory us =
               getHistory(o);
  for(int i = 0;
      i < us.length();i += UP){
    UpdateHistory h =
           us.subHistory(i, UP);
    if(!h.isFull() ||
        !h.containsSameValue())
          return;
  }
  amplify(o); }
```

```
Partition P {
  kind = alloc;
  history = UpdateHistory;
}

Event on_write(Object o,
     Field f, Word w1, Word w2){
  if(f.isPrimitive()){
    long l = combine(w1, w2);
    UpdateHistory hs =
           getHistory(o);
    UpdateHistory h =
           hs.subHistory
            (f.index() * UP, UP);
    h.update(l);
    if(h.isFull() &&
        !h.containsSameValue())
      deamplify(o);
  }
}
```

**Figure 4** An ISL description for amplifying allocation sites creating objects with invariant data contents.

```
Context MapContext {
  type = "java.util.Map";
}
Context TrackingContext{
  type = "Object[]";
  path =  MapContext;
}
History UtilityRates {
  type = "double";
  size = UP;//User Parameter
}
Partition P {
  kind = all;
  history = UtilityRates;
}
TObject MyObj{
  include = TrackingContext;
  partition =  P;
}
```

```
Event on_reachedOnce(Object o){
  UtilityRates h = getHistory(o);
  Object[] arr = (Object[]) o;
  int numNonNull = 0;
  for(Object ele : arr){
    if(ele != null)
      numNonNull ++;
  }
  double rate = (double)
          numNonNull/arr.length;
  h.update(rate);
  if(rate > 0.5) deamplify(o);
  else if(h.isFull()){
    for(int i = 0; i < UP; i ++){
      if(h.get(i) > 0.5)
        return;
    }
    amplify(o);
  }
}
```

**Figure 5** An ISL description for amplifying under-utilized Java maps.

```
History UseHistory {
  type = "boolean";
  size = UP; //User Parameter
}

Event on_rw(Object o, …){
  if(o.returned){
    getHistory(o).update(true);
    deamplify(o);
  }
}

Event on_reachedOnce(Object o){
  UseHistory h = getHistory(o);
  if(o.returned &&
      h.isFull() &&
      !h.contains(true))
    amplify(o);
}
```

```
Partition P {
  kind = all;
  history = UseHistory;
}

TObject MyObj {
  partition =  P;
  instance boolean returned =
                          false;
}

Event on_call(Object o, Method m,
         Word ret1, Word ret2){
  if(m.getRetType().
                isReference()){
    Object obj = addrToObj(ret1);
    obj.returned = true;
  }
}
```

**Figure 6** An ISL description for amplifying never-used return objects.

typed field of an object. Since our History construct can model only scalar values, we *linearize* histories of all fields into one single history whose size is UP ∗MAX_NUM_FIELDS where MAX_NUM_FIELDS is the maximal number of fields in a class (e.g., 100 is a reasonably large number). In other words, for a field with index $i$, elements from $i$∗UP to $(i+1)$∗ UP in the history model the updates of the field.

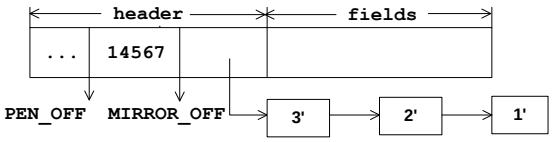
In the write event, we first obtain the sub-history for the field being accessed ($f$) from the history associated with the object, and then update the sub-history with the value to be written into the field. If a different value is seen in the sub-history, the object is no longer invariant and we cancel all the penalty previously added to the object. In the reachedOnce event, we check whether the sub-history for each field has a full record in which all values are the same. If it is the case, we start penalizing the object. Note that we can make amplification even finer-grained by using a context-based partitioning (e.g., kind = context[i]), which will enable us to identify certain contexts that are more likely to create identical objects than others and only amplify objects created under such contexts.

**Under-utilized Java maps.** The goal is to penalize Java maps that take a large memory space but contain only a very small number of elements. Using two contexts, we narrow our focus onto object arrays that are referenced by objects of any subtype of `Map`. We use a per-object partitioning with a history that tracks the most recent UP *utility rates* of each array. A utility rate of an array is defined as the ratio between the number of non-null elements and the length of the array. Every time a tracked array is traversed in the GC, we compute its utility rate and update the history. If the rate is greater than a user-provided threshold value (i.e., 0.5), the array is in good health and thus we cancel all its previous penalty. If the history has a full record in which all elements are smaller than 0.5, we start penalizing this array.

**Never-used return values.** Our goal is to detect and penalize objects that are never used after being returned by a method. Such objects are often indicators of wasteful computation done during the method invocation. Amplifying this problem is similar to amplifying a memory leak. The only difference is that we use a boolean instance field `returned` in each object to indicate whether the object has been returned by a method call. This flag is set in event on_call if a method call returns an object.

**Summary.** Observe that ISL has the following four advantages. (1) It has a Java-like syntax and thus is easy to use in real-world development; amplification for even complicated performance problems can often be specified using only a few lines of ISL code. On the contrary, had a JVM been modified manually to implement amplification, developing a testing tool for each problem would have needed modification of thousands of lines of code, which can take a skillful programmer several months or even longer.

(2) The `amplify` and `deamplify` commands enable developers to easily develop dynamic analyses with fewer false positives, making PerfBlower less sensitive to user parameters and heuristics. For example, although PerfBlower still needs a user threshold $i$ to determine when to perform amplification, its reliance on finding a perfect $i$ is significantly reduced by turning a symptom into a *cancelable penalty*. In our experiments, we observe that a small $i$ often works very well – if a program only has benign problems, although penalties are still created when a symptom appears, these penalties will be removed later by deamplification and thus not accumulate. As demonstrated in Section 5.4, for large applications such as




**(a) The layout of a tracked object**

```
Class Mirror{
  Object  o;    // the actual object it mirrors
  AllocSiteInfo info; // source code info of o's alloc site
  Field fInfo;  // information of the corresponding edge
  Mirror next;  // link to the next mirror object
}
```

**(b) Metadata Class Mirror**

**Figure 7** The layout of a tracked object at run time and the metadata class `Mirror` in PERFBLOWER.

`eclipse` and `mysql`, many thousands of false warnings are eliminated by deamplification but would otherwise have been reported by an existing leak detector.

(3) While PERFBLOWER penalizes individual objects, it is easy to attribute penalties to a data structure as a whole by defining reference-path contexts. For example, although array objects are penalized in the detection of under-utilized containers, reference-path contexts are used to connect those low-level arrays with high-level maps, lists, and sets, and hence, PERFBLOWER is able to report not only individual objects but also logical data structures.

(4) By exploiting a combination of instrumentation and runtime system support, ISL provides a unified way of expressing various memory-related performance problems which were defined separately in the existing work.

## 4 The PERFBLOWER Runtime System

This section describes our amplification runtime system.

**ISL compilation.** During compilation, contexts are broken into a set of $\langle ms, type \rangle$[3] pairs. A new `History` class is generated from a template based on the properties declared in history. An additional header space is requested per object to accommodate instance fields declared in TObject. Declaring many instance fields would create tremendous space overhead and thus developers are encouraged to declare as few fields as possible. Note that we do not generate a Java class for a TObject construct. The contexts used in the construct are checked at run time to determine whether an event needs to be invoked. We do not typecheck the imperative statements in an event construct; these events are directly translated into Java methods that are invoked at various program points; the generated methods will be typechecked when the RVM code is compiled.

**Incrementally adding penalties and mirroring reference paths.** Figure 7 (a) shows the layout of a tracked object in PERFBLOWER. We add two words in each object's header space to store (1) its penalty count (in numbers of bytes) and (2) a pointer pointing to the head of its mirror chain. Constants `PEN_OFF` and `MIRROR_OFF` are used to locate the offsets of

[3] *ms* and *type* refer to a method sequence and a type constraint, respectively.

**Algorithm 1:** Incrementally creating virtual penalties and building mirror paths when AMPLIFY($o$) is invoked.

**Input**: Object *obj* to be penalized, Object graph G
**Output**: Object graph G′ with additional mirror objects, and the VSO computed in this GC

```
1  Map newRefEdges // Reference edges to be added in the mirror chain
2  Set objToBeMirrored // Objects to be mirrored
3  List mirrorObjCurrGC // Mirror objects at the end of each mirror chain
4  List mirrorObjLastGC // Chain-ending mirror objects found in the previous GC
5  long VP ← 0 // the total size of the virtual penalty
6  BEGIN procedure PREGC
7  lastMark ← currMark // The initial value of currMark is 1
8  currMark ← currMark%2 + 1
9  END procedure PREGC
10 BEGIN procedure INGC // that traverses object currObj
11 if currObj = obj /*The object to be penalized is currently being traversed*/ then
12     s ← LOAD(obj, PEN_OFF) + SIZE(obj)
13     STORE(obj, PEN_OFF, s)
14     Mirror head ← LOAD (obj, MIRROR_OFF)
15     if head = null    //The mirror chain does not exist yet then
16         objToBeMirrored ← objToBeMirrored ∪ {obj}
17     else
18         Mirror m ← head
19         Mirror n ← m.next
20         while n ≠ null do
21             if refExists(n.o, m.o) then
22                 m ← n
23                 n ← n.next
24             else
25                 // the recorded path has changed
26                 m.next ← null
27                 break
28         // mark the object n mirrors
29         SETMARK(n.o, currMark)
30         mirrorObjCurrGC ← mirrorObjCurrGC ∪ {n}
31 foreach object p referenced by currObj do
32     if GETMARK(p) = lastMark then
33         newRefEdges ← newRefEdges ∪ {⟨currObj, p⟩}
34         UNMARK(p)
35 VP ← VP + LOAD (obj, PEN_OFF)
36 END procedure INGC
37 BEGIN procedure POSTGC
38 foreach Object m ∈ objToBeMirrored do
39     Mirror m′ ← CREATEMIRROR()
40     WRITESOURCECODEINFO(m, m′)
41     STORE(m, MIRROR_OFF, m′)
42 foreach Mirror object p ∈ mirrorObjLastGC do
43     if ∃⟨i, j⟩ ∈ newRefEdges: j = p.o then
44         /*The edge between i and j needs to be mirrored*/
45         Mirror i′ ← CREATEMIRROR()
46         WRITESOURCECODEINFO(i, i′)
47         p.next ← i′
48 mirrorObjLastGC ← mirrorObjCurrGC
49 mirrorObjCurrGC, objToBeMirrored, newRefEdges ← ∅
50 VSO ← (VP + LIVESPACESIZE())/ LIVESPACESIZE()    //compute virtual space overhead
51 END procedure POSTGC
```

these two locations upon accesses. Figure 7 (b) shows the definition of the `Mirror` class. Each mirror object contains two outgoing edges, one pointing to the next mirror object (i.e., via field `next`) and the other pointing to the actual object it mirrors (i.e., via field $o$). The mirror object also contains the source code information of the object it mirrors as well as the field information of the corresponding reference edge in the original path. In the example mirror chain shown in Figure 3, field `fInfo` in object 3' and object 2' record field $g$ and $f$ respectively.

Algorithm 1 shows our algorithm for incrementally adding penalties and building mirror paths. The algorithm has three major procedures: PREGC, INGC, and POSTGC. PREGC sets two mark values that will be used later in GC to mark objects (lines 6–9): *lastMark* and *currMark*, which represent, respectively, the mark values used in the last and the current GC, alternate between 1 and 2. Procedure INGC traverses the object graph to build mirror reference paths (lines 10–36). Because the object graph traversal cannot go backward, it is impossible to mirror a reference path with multiple edges in one single GC run. The basic idea of the incremental algorithm is *to mirror one edge on the path in each GC, starting from the one closest to the penalized object.* For example, to build the mirror path in Figure 3, we first mirror the edge going directly to object 3 (i.e., $2 \xrightarrow{g} 3$) by creating an edge between $3'$ and $2'$. Next, we mark object 2 with a special mark value so that 2 will be recognized in the next GC and the edge between 1 an 2 can be mirrored. If object 3 keeps being penalized (e.g., indicating that we are amplifying a true problem), a long mirror chain will be constructed and, hence, more complete path information will be reported.

During the graph traversal, if the object being reached happens to be the one to be penalized (lines 11–30), we first increase the object's PC by the size of object (lines 12 and 13), and then find the head object of its mirror path (line 14). If no mirror object has been created (i.e., it is the first time to penalize this object), we remember the object in set *objToBeMirrored* – since we cannot create mirror objects during a GC, we will do it later after the GC. If the head mirror object is found, we need to create a new mirror object and append it to the existing mirror path. Since this existing mirror path was built in the previous GCs, certain reference edges being mirrored may have changed. To detect such changes, we traverse the mirror path (lines 18–27) to validate if each edge in the path still mirrors a valid heap reference (line 21). When the traversal finishes, $n$ is either the last node of the mirror path or the first node at which the old reference relationship breaks. A new mirror object will be created and linked to $n$. We mark the object that $n$ mirrors (i.e., $n.o$) with *currMark* (line 29) so that $n.o$ will be recognized in the next GC and a new reference edge pointing to $n.o$ will be mirrored. $n$ is then remembered in set *mirrorObjCurrGC* and we leave the mirror object creation to the PostGC procedure.

If any child of the object being traversed has been marked (by the previous GC) (line 32), the reference edge connecting the object and this child needs to be mirrored. We remember the pair $\langle currObj, p \rangle$ in map *newRefEdges* for further processing (line 33) and then unmark object $p$ (line 34). It is important to note that the correctness of an existing mirror chain is verified when its root node is traversed during each GC (lines 25–27). If the reference path it mirrors becomes invalid, the mirror chain will be removed from the root node and the system starts mirroring the new reference path.

All mirror objects are created after the GC is done (lines 38–47). We first create a mirror object $m'$ for each object $m \in objToBeMirrored$ that does not have a mirror chain, store $m$'s allocation site information into $m'$ (line 40), and write a reference of $m'$ into $m$'s header space (line 41). Next, we check each chain-ending mirror object $p \in mirrorObjLastGC$ and see whether there exists a new heap reference edge recorded in *newRefEdges* whose target is

mirrored by $p$ (line 43). If such an edge exists, we create one more mirror object to mirror the source of the edge (lines 45–47).

All objects in list *mirrorObjCurrGC* are added to list *mirrorObjLastGC* before the mutator execution resumes, so that these objects will be used to mirror new edges in the next collection. Finally, the data structures *mirrorObjCurrGC*, *objToBeMirrored* and *newRefEdges* are cleared (line 49), and the VSO is computed (line 50) based on the penalty size *VP* (which is updated at each node traversal at line 35).

Our algorithm can mirror only one reference path for each object. We find it to be sufficient in most cases: there is often one single reference path in which an object is inappropriately used and causes a problem, although the object may be referenced by many paths at various points; if the object keeps being penalized, the penalty chain will eventually get stabilized to mirror the problematic path necessary for the developer to fix the problem. Furthermore, for many types of performance problems, reporting *any* reference path will be helpful to find their root causes. For instance, for leak detection, any reference path that holds a suspicious object and makes it reachable is problematic. As another example, for detection of under-utilized containers, PerfBlower tracks inefficiently-used arrays and the key to producing a high-quality report is to find the data structures (e.g., HashMap, ArrayList, etc.) that reference those arrays. Since an array is often *owned* by a high-level data structure, any reference path that leads to the array must pass the data structure, and hence, reporting any path will be sufficient for the developer to find the data structure and understand the problem.

**Limitations.** While PerfBlower captures commonalities of different memory-related problems and makes it easy for them to manifest, it has a few limitations that leave room for improvement. First, it can only find heap-related inefficiencies, while there are many different sources for real-world performance problems, such as idle threads, redundant computations, or inappropriate algorithms. Second, the JVM needs to be rebuilt every time a new checker is added. In practice, this is not an issue, since most modern JVMs support fast and incremental building. For example, it takes only one minute to build the Jikes RVM on which PerfBlower is implemented. One future direction would be to make PerfBlower a JVMTI-like library that can be dynamically registered during bootstrapping of the JVM without needing to build the JVM. Third, since PerfBlower piggybacks on the GC, its effectiveness may be affected by the GC frequency. PerfBlower may report a higher overhead if a program is executed with a smaller heap (due to more GCs). While this introduces uncertainties, our experimental results (in Figure 8) demonstrate, for most programs, the VSOs reported by PerfBlower are very stable across different heap sizes. Finally, PerfBlower may become less effective when a generational GC is used. Since a generational GC does not frequently perform full-heap scanning, objects in the old generation space may not be effectively amplified.

## 5 Evaluation

To implement the proposed technique, we have modified both the baseline compiler and the optimizing compiler in the Jikes Research Virtual Machine (RVM), version 3.1.3, which uses the MarkSweep GC. PerfBlower is now available on BitBucket for download. Using ISL, we have implemented three different amplifiers that target, respectively, memory leaks, under-utilized containers, and over-populated containers. In our evaluation, no application-related information was added into these descriptions, although a future user may describe a

**Table 1** Virtual space overheads (in times) reported by the amplifiers for memory leaks (section (a)), under-utilized containers (section (b)) and over-populated containers (section (c)); each section reports the real space overheads (*Inf*) and the VSOs when the size of the history $m$ is 10. Highlighted in the table are the programs whose maximal VSO is greater than 15. Notation of the form $a \rightarrow b$ means a program with a VSO $a$ has a new VSO $b$ after its performance problems are fixed. GeoMean-H and GeoMean-L report the average VSOs for the highlighted and non-highlighted programs, respectively.

| *Bench* | (a) MEM | | (a) UUC | | (b) OPC | |
|---|---|---|---|---|---|---|
| | *Inf* | $m = 10$ | *Inf* | $m = 10$ | *Inf* | $m = 10$ |
| antlr | 1.20 | 1.1 | 1.22 | 2.6 | 1.20 | 1.0 |
| bloat | 1.35 | 1.1 | 1.72 | 5.9 | 1.58 | 1.1 |
| eclipse | 1.28 | 5.6 | 1.21 | 8.7 | 1.23 | 3.5 |
| fop | 1.15 | 1.6 | 1.22 | 3.3 | 1.16 | 1.1 |
| **hsqldb** | 1.24 | **26.2**→1.3 | 1.19 | **16.6**→2.2 | 1.19 | **29.2**→1.7 |
| **jython** | 1.16 | **22.7**→6.4 | 1.06 | **48.3**→5.3 | 1.06 | **16.3**→3.8 |
| luindex | 1.18 | 1.2 | 1.16 | 1.7 | 1.13 | 1.1 |
| lusearch | 1.60 | 1.4 | 1.70 | 1.5 | 1.69 | 1.2 |
| pmd | 1.18 | 1.5 | 1.12 | 1.4 | 1.12 | 1.1 |
| **xalan** | 1.15 | **53.1**→7.1 | 1.09 | **117.7**→7.9 | 1.08 | 3.4 |
| GeoMean-H | | **31.6** | | **45.5** | | **21.8** |
| GeoMean-L | | 1.6 | | 2.9 | | 1.5 |

symptom in a way that is specific to an application. These ISL programs were easy to write and took us only a few hours. All experiments were executed on a quadcore machine with an Intel Xeon E5620 2.40GHZ processor, running Linux 2.6.32.

## 5.1 Amplification Effectiveness

The first set of experiments focuses on understanding of whether our technique is effective in exposing performance problems in a testing environment. Since the DaCapo benchmark set [2] provides three different kinds of workloads (i.e., small, default, and large), we ran each program on its small workload to simulate how a developer would use our tool during testing – it is much more difficult to reveal performance bugs under small workloads than large workloads. We make no assumption about test inputs – real-world developers can enable amplification when executing a regular test suite; whenever a test case triggers a true problem, amplification will incur a large virtual overhead.

**Indication of performance bugs.** Table 1 reports the real and virtual space overheads of the three amplifiers over the DaCapo benchmarks (2006 release) using a 500MB heap. The comparison of the amplification results under different heap sizes will be discussed shortly. We discuss our experimental data only for $m = 10$, where $m$ is the user-specified history length. Other configurations have yielded similar results; detailed results for the memory leak detector are reported in Table 2. Clearly, these VSOs indicate that leaks may exist in `hsqldb`, `jython` and `xalan`; under-utilized containers in `hsqldb`, `jython` and `xalan`; and over-populated containers in `hsqldb` and `jython`.

An important observation made here is that there is a clear distinction between the overheads of programs with and without problems, making it particularly easy for developers to determine whether manual inspection is needed during testing. For example, to find over-populated containers, there are only two programs whose memory consumption is significantly amplified. Developers only need to read reports for these two programs; they

**Table 2** Virtual space overheads (in times) of the memory leak amplifier in different configurations using a 500MB heap; section $m{=}i$ shows, for each program, the maximal VSO when the length of the execution window (i.e., field size of a history) is $i$. Highlighted in the table are the programs whose maximal VSO is greater than 15. Notation of the form $a \rightarrow b$ means a program with a VSO $a$ has a new VSO $b$ after its performance problems are fixed. GeoMean-H and GeoMean-L for a section $m = i$ report the average VSOs under $m = i$ for the highlighted and non-highlighted programs, respectively.

| *Bench* | $m = 5$ | $m = 10$ | $m = 15$ | $m = 20$ |
|---|---|---|---|---|
| antlr | 1.2 | 1.1 | 1.2 | 1.2 |
| bloat | 1.1 | 1.1 | 1.1 | 1.1 |
| eclipse | 5.3 | 5.6 | 6.3 | 6.3 |
| fop | 1.6 | 1.6 | 1.7 | 1.7 |
| **hsqldb** | **26.5**→1.7 | **26.2**→1.3 | **25.9**→1.2 | **25.7**→1.2 |
| **jython** | **22.7**→6.3 | **22.7**→6.4 | **22.7**→6.4 | **24.5**→6.3 |
| luindex | 1.2 | 1.2 | 1.2 | 1.2 |
| lusearch | 1.4 | 1.3 | 1.3 | 1.3 |
| pmd | 1.5 | 1.5 | 1.5 | 1.5 |
| **xalan** | **53.1**→7.3 | **53.2**→7.1 | **40.4**→2.7 | **37.0**→1.9 |
| **mysql** | **155.5**→2.3 | **108.9**→2.3 | **70.4**→1.5 | **30.1**→1.2 |
| **mckoi** | **111.6**→6.9 | **72.4**→6.6 | **85.6**→3.3 | **42.6**→1.7 |
| GeoMean-H | **56.1** | **47.8** | **42.7** | **31.3** |
| GeoMean-L | 1.6 | 1.6 | 1.6 | 1.6 |

would otherwise have to inspect all programs and check each warning had a regular detector been used. The differences in the VSOs of programs with and without problems are clearly demonstrated by the numbers reported in the GeoMean-H and GeoMean-L rows. We have manually inspected the programs whose VSO is $> 15$[4] and found problems in all of them. Details of these problems will be discussed shortly in Section 5.2. Note that after their performance problems are fixed, the VSOs of these problems are significantly reduced.

**Space and time overheads.** Section *Inf* of Table 1 reports the real space overheads of our infrastructure, computed as $S_1/S_0$, where $S_0$ and $S_1$ are, respectively, the peak post-GC heap consumptions of the unmodified and modified RVM. The average space overheads incurred by the three amplifiers (i.e., memory leaks, under-utilized containers, and over-populated containers) under the configuration $m = 10$ are 1.23×, 1.23×, and 1.25×, respectively. The overall time overheads under the same configuration are 2.39×, 2.74×, and 2.73×, respectively. To understand the scalability of the tool, we have also run these amplifiers using the DaCapo large workloads. The time overheads for the three amplifiers are all between 2.5× and 3.2×.

**Sensitivity to the execution window length.** We ran the memory leak amplifier with $m = 5$, $m = 10$, $m = 15$ and $m = 20$ to show whether our framework is sensitive to the execution window length. Table 2 shows the results for the DaCapo benchmarks with different $m$. We added two extra programs `mysql` and `mckoi` to our benchmark set – they contain known leaks studied before and we are thus interested in how our amplifier performs on these programs.

We make two observations based on the amplification results. First, the VSOs under different parameters are generally stable; the size of an execution window does not have much impact on the amplification results. On the contrary, the effectiveness of an existing

[4] 15 is just a number in a large range (between 10 and 20) that can easily distinguish these overheads.

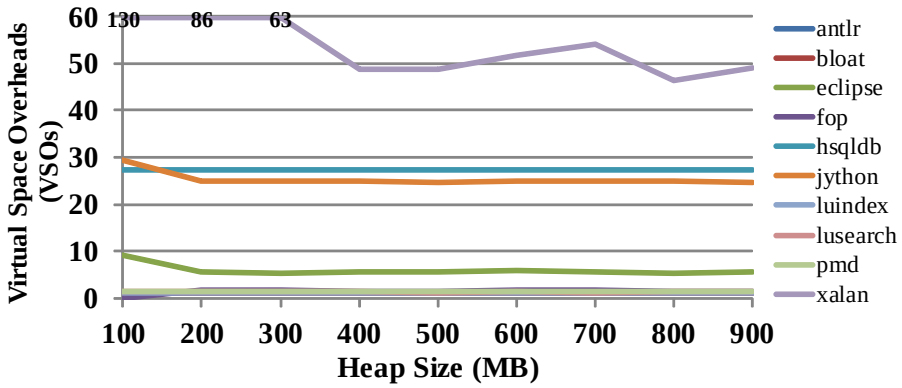



**Figure 8** VSOs for the memory leak amplifier under different heap sizes.

symptom-based performance problem detector relies heavily on such a threshold. Second, as the size of the execution window increases, the space overhead of the tool decreases in general. This is straightforward – the larger the size of the history is, the fewer objects are amplified, and the smaller effect amplification creates.

**Sensitivity to the heap size.** Are VSOs sensitive to the size of the heap? As demonstrated in Figure 8, the VSOs reported by PERFBLOWER are quite stable under different heap sizes.

## 5.2 Problems Found

We have manually inspected our reference path report for each highlighted program in Table 1 and Table 2. We have not found any highlighted program to be mistakenly amplified, that is, real performance problems are identified in all of them. We have studied a total of 10 amplification reports and found 8 unknown problems that have never been reported in any prior work. Furthermore, all of these problems are uncovered under small workloads; their executions do not exhibit any abnormal behavior without amplification. In this section, we discuss our experiences with 8 unknown problems. To reduce the influence of execution noise, the performance gains shown below are obtained by running each modified program twenty times and comparing the medium running time and space consumption with those of its unmodified version. For each problem found, we have developed a fix. As shown in Table 1 and Table 2, the VSOs of these programs are significantly reduced after their fixes are applied. This demonstrates that PERFBLOWER is sensitive only to true memory issues, not false problems.

For each program, we have also run an existing leak detector, SLEIGH [3], and compared our reports with SLEIGH's reports. The reason we chose SLEIGH is because it is the only publicly available performance problem detector that does not require user involvement. While there exist other tools such as LeakChaser [36] and a few inefficient data structure detectors [26], they are either unavailable or require heavy user annotations to understand the program semantics. Hence, we compare PERFBLOWER only with SLEIGH in this paper.

**xalan-leak.** The No. 1 allocation site in the report has a very long reference path (with 10 nodes); the top 3 objects on the path are as follows:

```
#0. id: 5561 ObjectVector.java: 106
Method: org.apache.xml.utils.ObjectVector.<init>
Object Type: java.lang.Object[]
Number of suspicious objects: 27
Size: 0x00d84380
Path: ID: 5573 XPathContext.java: 920
   Method: org.apache.xpath.XPathContext.<init>
   Object Type: org.apache.xml.utils.ObjectStack
   <= ID: 5557 TransformerImpl.java: 402
      Method: org.apache.xalan.transformer.TransformerImpl.<init>
      Object Type: org.apache.xpath.XPathContext
      <= ID: 5543 StylesheetRoot.java: 212
         Method: ...templates.StylesheetRoot.newTransformer
         Object Type: org.apache.xalan.transformer.TransformerImpl
         <= ...
```

The reference path shows us that the largest penalty is added to objects created and referenced by the instances of an XML transformer class `TransformerImpl`. This immediately caught our attention, since `TransformerImpl` objects should be reclaimed after each transformation. The reference path directed us to class `StyleSheetRoot`, which implements interface `Templates` and has a reference to an `IteratorPool` object. Each `IteratorPool` object maintains a vector of `DTMAxisTraverser` objects in order to reuse those objects. However, upon the reuse of an old `DTMAxisTraverser` object, the object still keeps a reference of its previous `SAX2DTM` object, which, in turn, references the `TransformerImpl` object. Hence, each reused `DTMAxisTraverser` object in the `IteratorPool` leaks its previous `TransformerImpl`. This is a serious problem, because the size of a `TransformerImpl` object is often very large (e.g., at the megabyte level). We came up with a quick fix in which one line of code is commented out to disallow reuse of `DTMAxisTraverser` objects. **The fix has resulted in a 25.4% reduction in memory consumption and a 14.6% reduction in execution time.** A subsequent search in the `xalan` bug repository reveals that the same bug was found and fixed in version 2.5.1 [1] with a different approach, while the DaCapo xalan version in which we detected the bug is a much earlier one (2.4.1). We found that every object on this long path is necessary for understanding the cause of the bug. It would not be possible to record such complete information with a regular detector/profiler, as it uses a fixed-size space to store metadata for each object.

For `xalan`, Sleigh reported a few stale objects, including their types and numbers of instances. Most of these types are Java library classes or VM classes, such as `java.lang.String` and `com.ibm.JikesRVM.classloader.VM_Atom`, which have nothing to do with the leak. In this case, Sleigh provided neither allocation sites nor last access sites for these stale objects. This is primarily because Sleigh uses only one bit per object to encode information statistically; it needs a long execution to gather sufficient data for producing a diagnostic report, and thus does not work well for test executions, which are often very short.

**jython-leak.** From the report, it is easy to identify that most space penalties are added to stale `PyJavaPackage` objects. One use of these objects is to cache the relationships between Java classes and Jar files. When a jar file is loaded, Jython iterates all the classes and creates PyJavaPackage objects. Many classes are never used during execution, and thus, their corresponding `PyJavaPackage` objects become leaks. We fixed the problem by (1) employing lazy loading and (2) removing unnecessary jars from classpath. **The fix has led to a 24.3% peak memory reduction and a 7.4% time reduction for the warm up phase.**

In SLEIGH's report, there was only one stale allocation site. Although this allocation site is related to the cause of the leak, it is still far away from the cause and it would take a developer a fair amount of effort to find the cause from the allocation site. On the contrary, PERFBLOWER provided much more precise diagnostic information, significantly alleviating the developer's debugging burden.

**hsqldb-leak.** Most of the reported objects are related to data values computed but not used at all in the DaCapo execution. For example, `hsqldb` maintains three maps between paths and databases. Although in DaCapo only the memory database is used, `hsqldb` also keeps maps for the file database and the resource database. One way to solve the problem is to provide configuration options that allow users to specify the databases needed. We changed these stale objects to lazy initialization, reducing the memory consumption by **15.6%**.

**hsqldb-UUC.** The report shows most under-utilized containers are due to inappropriate initial capacity. For example, the top object in our report points to arrays created during the initialization of `BaseHashMap`. The reference path quickly led us to inspect ValuePool.java. In this class, `hsqldb` caches six `ValuePoolHashMap` objects in static fields, which are used to support singleton patterns. However, all of these maps have 10000 as their initial capacity, which is way too large for many clients. We fixed the problem by making these containers grow dynamically as necessary and changing their initial capacity to 32. **This optimization has led to a 17.4% space reduction.**

**jython-UUC.** Our report indicates that the largest penalty is added to arrays created in `PyStringMap.resize` method. By inspecting `PyStringMap`, we found that the class uses a string array to store keys and a `PyObject` array to store values. Before each insertion, it checks the load factor of its key array (i.e., how full the map is allowed to get before its capacity is increased). In this case, if the load rate is higher than 0.5, its capacity is doubled. Actually, since the hash function of `String` works generally well and the key array has a very small collision rate, using 0.5 as load factor seems too aggressive. Modifying it to 0.75 resulted in a **19.1% space reduction**.

**xalan-UUC.** The top warning in the report points to `ObjectStack` objects created in `XPathContext` and `TransformerImpl` objects. The initial capacity of `ObjectStack` is defined by `XPathContext. RECURSIONLIMIT`, a static final field with a value 4096. A detailed inspection of the source code reveals that ObjectStack is implemented based on ObjectVector, which is a data structure designed to support random accesses. The initial capacity defines a tradeoff between space and time: using a smaller value saves space at the cost of increased lookup computation. Finding the sweet spot requires deep understandings of the program and empirical studies. When we use 2048 as the initial capacity, the memory consumption is reduced by **5.4%**, and the execution time is reduced by **34.1%**.

**hsqldb-OPC.** Before `hsqldb` executes an `insert` query, it invokes a method called `getNewRowData` to create an object array that represents the row to be inserted. A column in a database table often has a default value – when a new row specified by an `insert` query does not have a value for the column, this default value will be filled in. We find that method `getNewRowData` creates a large number of objects and fill them into the object array as default values. These objects are never retrieved from this array until later they

**Table 3** Comparison of results from PerfBlower and Sleigh; leaks$^+$ means the tool reports both memory leaks and useful diagnostic information; leaks$^-$ means the tool only reports leaks without useful information.

| *Bench* | PerfBlower | Sleigh |
|---|---|---|
| antlr | no leak | no leak |
| bloat | no leak | no leak |
| eclipse | leaks$^+$ | no leak |
| fop | no leak | no leak |
| hsqldb | leaks$^+$ | leaks$^+$ |
| jython | leaks$^+$ | leaks$^-$ |
| luindex | no leak | false positive |
| lusearch | no leak | fail to run |
| pmd | no leak | no leak |
| xalan | leaks$^+$ | leaks$^-$ |
| mysql | leaks$^+$ | fail to run |
| mckoi | leaks$^+$ | leaks$^-$ |
| jbb | leaks$^+$ | leaks$^+$ |
| true unknown leaks | **4** | 2 |
| true known leaks | **3** | **3** |
| false positives | **0** | 1 |
| useful information | **7** | 2 |

are replaced by the actual values. We fix the problem by performing a lazy default value assignment, resulting in a **14.9% space reduction.**

**jython-OPC.** The cause of the problem here is the same as in jython-leak. The top object in our report directed us to inspect method `PyJavaPackage.resize`, in which a large array is created to contain Java classes, most of which are never retrieved from the map. This same problem has two different manifestations.

**Comparison with Sleigh.** Table 3 summarizes the comparison of leak reports between PerfBlower and Sleigh. PerfBlower found memory leaks in seven benchmarks (including four new leaks and three known leaks), and for each memory leak, PerfBlower provided very precise diagnostic information. Sleigh reported that six programs contain memory leaks, including three known leaks, two new leaks, and a false warning. Furthermore, Sleigh did not provide any useful diagnostic information for three leaking programs. The comparison demonstrates that PerfBlower is able to find more leaks, has fewer false positives, and generates more precise diagnostic information.

## 5.3 PerfBlower Completeness

Our amplifiers are able to find bugs in all of the highlighted programs, but do they miss bugs? To answer this question, we performed an additional experiment. In this experiment, the benchmark set includes 14 programs (shown in Table 4), which contain known performance problems reported in the literature [14, 3, 5, 36, 39] except Chameleon [26]. Chameleon is a dynamic technique that can detect inefficiently-used containers. However, it is based on the commercial J9 VM and not publicly available. In addition, the description of the bugs

**Table 4** Completeness of PERFBLOWER: for each existing Java-based performance problem detector, the table reports the bugs studied in its original paper and whether they are found by PERFBLOWER.

| *Work* | *Bugs studied* | PERFBLOWER |
|---|---|---|
| Cork [14] | SPECjbb's leak | Reported |
| | Eclipse #115789 | Reported |
| Sleigh [3] | SPECjbb's leak | Reported |
| | Eclipse #115789 | Reported |
| Container Profiler [38] | SPECjbb's leak | Reported |
| | JDK #6209673 | Cannot execute |
| | JDK #6559589 | Cannot execute |
| LeakChaser [36] | SPECjbb's leak | Reported |
| | Eclipse #115789 | Reported |
| | Eclipse #155889 | Reported |
| | MySQL's leak | Reported |
| | Mckoi's leak | Reported |
| Leak Pruning [5] | SPECjbb's leak | Reported |
| | Eclipse #115789 | Reported |
| | Eclipse #155889 | Reported |
| | MySQL's leak | Reported |
| | JbbMod's leak | Reported |
| | Mckoi's leak | Reported |
| | Delaunay's leak | Reported |
| | List leak | Reported |
| | Swap leak | Reported |
| | Dual leak | Reported |
| Static Bloat Finder[39] | Bloat's UUC | Not Triggered |
| | Chart's OPC | Cannot execute |

in [26] is at a very high level and does not contain detailed location information (such as classes, methods, and line numbers). Hence, those bugs were not considered.

Among these benchmarks, three programs could not be executed. Two of them are about memory leaks in Sun JDK library, which is not used by Jikes RVM. Another benchmark is `chart`, which could not be executed due to the missing of certain AWT libraries. For the remaining bugs, all but one were captured by PERFBLOWER. The missing one is an under-utilized container in the DaCapo benchmark `bloat`, reported by a static analysis-based container problem detector [39]. Because PERFBLOWER is based on dynamic analysis, its ability of finding bugs depends on the coverage of test cases. We inspected the source code of benchmark `bloat` and concluded that this bug could not be triggered by the DaCapo input. Overall, PERFBLOWER did not miss any bug that could be triggered by a test case.

## 5.4 False Positive Elimination

In order to understand if our technique successfully eliminates false warnings of a regular performance problem detector, we measured the number of objects that have experienced both amplification and deamplification. These objects are amplified because they satisfy the symptom specification for a sufficiently long period; they are deamplified later when the symptom disappears. Although they are not true causes of performance problems, a symptom-based detector would report all of them.

**Table 5** False warnings that are eliminated by deamplification but would have been reported by a memory leak detector; for each program under each history size $i$, we report four numbers *FO*, *FS*, *TO*, and *TS*; *FO* is the number of such objects that their staleness exceeds $i$ but they are used afterwards, *FS* is the number of allocation sites creating objects in *FO*, *TO* is the total number of objects whose staleness exceeds $i$, and *TS* is the number of allocation sites creating objects in *TO*. Although objects under *FO* are not true leaks, a regular leak detector would report all of them.

| *Bench* | $m = 5$ | | $m = 10$ | | $m = 15$ | | $m = 20$ | |
|---|---|---|---|---|---|---|---|---|
| | *FO/FS* | *TO/TS* | *FO/FS* | *TO/TS* | *FO/FS* | *TO/TS* | *FO/FS* | *TO/TS* |
| antlr | 26/2 | 103/68 | 10/2 | 87/68 | 3/2 | 80/68 | 2/1 | 79/68 |
| bloat | 19/3 | 72/29 | 4/3 | 56/29 | 3/3 | 54/29 | 2/2 | 53/29 |
| eclipse | 5992/498 | 12062/1279 | 4697/246 | 10572/1030 | 4648/235 | 9747/1018 | 4339/201 | 8852/958 |
| fop | 27/2 | 823/723 | 9/1 | 806/722 | 4/1 | 797/722 | 0/0 | 793/722 |
| hsqldb | 386/4 | 1866/105 | 188/4 | 1236/105 | 68/3 | 793/105 | 13/3 | 493/105 |
| jython | 28/3 | 17864/1072 | 21/3 | 17709/1034 | 11/2 | 17634/951 | 7/2 | 17410/830 |
| luindex | 9/2 | 94/72 | 5/2 | 90/72 | 2/2 | 90/72 | 2/2 | 88/72 |
| lusearch | 183/15 | 1713/113 | 32/14 | 206/67 | 5/4 | 178/66 | 3/2 | 174/64 |
| pmd | 28/2 | 238/156 | 6/2 | 210/156 | 1/1 | 204/156 | 1/1 | 203/156 |
| xalan | 120/12 | 19795/637 | 65/11 | 14657/637 | 17/8 | 11304/636 | 10/7 | 10540/628 |
| mysql | 4457/3 | 54937/192 | 1933/3 | 40844/192 | 2/2 | 26547/192 | 2/2 | 12877/192 |
| mckoi | 116/76 | 12004/298 | 116/76 | 11547/292 | 111/76 | 10133/292 | 1/1 | 5058/236 |
| jbb | 8192/55 | 54358/59 | 27/8 | 49018/55 | 5/4 | 27913/55 | 4/4 | 25081/53 |

We focus only on memory leak amplification in this subsection; our results are shown in Table 5. The setup for this experiment is exactly the same as the one for the experiment reported in Table 2: the DaCapo small workload is used and the number of iterations for each program is such that the execution can have at least 20 GCs. *FO* and *FS* are, respectively, the numbers of falsely amplified objects and their allocation sites; these false warnings are eliminated by amplification but would have otherwise been reported by a regular memory leak detector.

We make three observations based on these numbers. First, the number of false warnings is reduced as we increase the threshold $i$. This is straightforward because true leaking objects often have larger staleness than false leaks. However, increasing $i$ can easily lead to *false negatives* because the staleness of objects created late during execution may not reach $i$ before the execution finishes. This can be observed from the fact that the total numbers of suspicious objects *TO* and suspicious allocation sites *TS* drop significantly, especially for `lusearch`, `xalan`, `mysql`, and `mckoi`. Our algorithm frees developers from choosing a perfect threshold $i$: one can always use a small $i$ (such as 5) to ensure no true problem is missing without worrying about false positives – most of them are automatically eliminated by deamplification. Finally, PerfBlower has eliminated not only false leaking objects (*FO*), but also false leaking allocation sites (*FS*). Since a leak detector often ranks and reports allocation sites, elimination of false leaking allocation sites leads directly to more precise reports as well as reduced manual inspection effort.

## 6 Related Work

**Finding performance bugs.** Jin et al. [13] study a number of performance bugs in real-world programs and develop a pattern-based approach to find bugs in the program source code. Song and Lu [28] propose a statistical approach to finding performance bugs in real-world software. Xu et al. propose static [39, 40] and dynamic [37] techniques to detect various kinds of performance problems. Recent work from [23] proposes a static analysis that can find redundant container traversals. Mitchell et al. propose heap analysis techniques [20, 19] that can correlate system metrics with program behaviors. Nistor et al. [22] propose a runtime technique to detect performance problems by looking for code loops that exhibit similar memory-access patterns. Han et al. [12] identify performance problems by mining large

numbers of stack traces. Stewart et al. propose EntomoModel [29], a dynamic framework that uses decision tree classification and a design-driven performance model to identify and avoid performance anomaly. Caramel [21] detects loop-related performance bugs that can be easily fixed by adding conditional breaks. These existing works are postmortem debugging techniques, focused on finding the root cause of a performance problem that already manifests in a user-provided test execution or actual production run, while our work provides a general framework to describe and amplify performance problems during testing.

Xiao et al. [34] develop a delta inference technique that predicts workload-dependence performance bugs by using models with respect to workload size to infer iteration counts. Yu et al. [42] propose a technique that collects large numbers of traces to measure performance impacts and identify their root causes. WuKong [44] is an automated technique that builds a feature-based behavioral model to predict bugs that manifest at large-system scale. WuKong falls into the category of *learning-based bug detection* where small-scale training runs are employed to build the model, while our approach uses developer-provided symptom specifications to amplify performance problems.

**Test amplification techniques.** Test amplification is a notion wherein a single test execution can be used to learn much more information about a program's behavior than is directly exhibited by that particular execution. Zhang et al. [43] proposes a test amplification technique that exhaustively explores the space of exceptional behaviors to validate exception handling code. Work from [16] uses a static information flow analysis to amplify the result of a single test execution over the set of all inputs and interleavings for verifying properties of data parallel GPU programs. Dynamic test generation techniques such as [7, 9, 25] can also be considered as test amplification techniques which generate many tests from one test execution to achieve path coverage.

**Domain-specific languages.** The Broadway compiler project [11] contains an annotation language and a compiler that can customize a library implementation for specific application requirements. Other domain-specific languages and compilers attempt to either incorporate application semantics into compilation to improve performance (such as [8]) or generate application code from declarative specifications (such as [27]). Annotations such as DyC [10] have been used to direct dynamic optimization. Vandevoorde [32] proposes specifications based on Larch [33] to enable performance optimizations. Work closest to the proposed ISL language is [24] that develops a declarative language called DEAL to describe assertions checkable within one single GC run. Unlike DEAL that aims to inform the GC how to check assertions, ISL is an event-based instrumentation language that specifies (1) how a program should be instrumented, (2) what data need to be collected, and (3) how the mutator and the GC should collaborate.

## 7 Conclusions

This paper presents the first technique that can expose a class of performance problems during testing by amplifying their effects. We first design a language to describe the symptom of a performance problem if the symptom can be expressed by logical statements over a history of heap state updates. Next, we develop compiler and runtime system support that compiles an ISL program, performs the instrumentation, and amplifies the target problem when the symptom is observed at run time. We have implemented this technique on Jikes RVM and used it to successfully amplify three types of heap data-related performance problems.

**Acknowledgements.** We would like to thank Michael Bond, Brian Demsky, David Liu, Shan Lu, and Feng Qin for their helpful comments on an early draft of the paper. We also thank the anonymous reviewers for their valuable and thorough comments. This material is based upon work supported by the US National Science Foundation under grant CNS-1321179 and CCF-1409829, and by the Office of Naval Research under grant N00014-14-1-0549.

## References

1 Apache JIRA issue tracker. `https://issues.apache.org/jira/browse/XALANJ-796`.
2 S. M. Blackburn and *et al.* The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.
3 Michael D. Bond and Kathryn S. McKinley. Bell: Bit-encoding online memory leak detection. In *ASPLOS*, pages 61–72, 2006.
4 Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. In *OOPSLA*, pages 97–112, 2007.
5 Michael D. Bond and Kathryn S. McKinley. Leak pruning. In *ASPLOS*, pages 277–288, 2009.
6 Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley, 2010.
7 Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *CCS*, pages 322–335, 2006.
8 Dawson R. Engler. Incorporating application semantics and control into compilation. In *DSL*, pages 9–9, 1997.
9 Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.
10 Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theor. Comput. Sci.*, 248(1-2):147–199, October 2000.
11 Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *DSL*, pages 39–52, 1999.
12 Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, pages 145–155, 2012.
13 Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.
14 Maria Jump and Kathryn S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *POPL*, pages 31–38, 2007.
15 JUnitPerf. `http://www.clarkware.com/software/JUnitPerf.html`, 2003.
16 Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying GPU kernels by test amplification. In *PLDI*, pages 383–394, 2012.
17 Load Test Tools. `http://www.softwareqatest.com/qatweb1.html`, 2015.
18 Microbenchmarking framework for Java. `https://code.google.com/p/caliper/`, 2013.
19 Nick Mitchell and Gary Sevitsky. The causes of bloat, the limits of health. In *OOPSLA*, pages 245–260, 2007.
20 Nick Mitchell, Gary Sevitsky, and Harini Srinivasan. Modeling runtime behavior in framework-based applications. In *ECOOP*, pages 429–451, 2006.
21 Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*, 2015.
22 Adrian Nistor, Lintao Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571, 2013.

23 Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, 2015.

24 Christoph Reichenbach, Neil Immerman, Yannis Smaragdakis, Edward Aftandilian, and Samuel Z. Guyer. What can the GC compute efficiently? A language for heap assertions at GC time. In *OOPSLA*, pages 256–269, 2010.

25 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *FSE*, pages 263–272, 2005.

26 Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: Adaptive selection of collections. In *PLDI*, pages 408–418, 2009.

27 Yannis Smaragdakis and Don Batory. DiSTiL: a transformation library for data structures. In *DSL*, pages 20–20, 1997.

28 Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *OOPSLA*, pages 561–578, 2014.

29 C. Stewart, Kai Shen, A. Iyengar, and Jian Yin. Entomomodel: Understanding and avoiding performance anomaly manifestations. In *MASCOTS*, pages 3–13, 2010.

30 The Grinder Java load testing framework. `http://grinder.sourceforge.net/`, 2013.

31 The SmartBear distributed testing framework. `http://support.smartbear.com/articles/testcomplete/distributed-testing-tutorial`, 2013.

32 M. Vandevoorde. *Exploiting Specifications to Improve Program Performance*. PhD thesis, Massachusetts Institute of Technology, 1994.

33 Jeannette M. Wing. Writing larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

34 Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ISSTA*, pages 90–100, 2013.

35 Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *PLDI*, pages 419–430, 2009.

36 Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. LeakChaser: Helping programmers narrow down causes of memory leaks. In *PLDI*, pages 270–282, 2011.

37 Guoqing Xu, Nick Mitchel, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *PLDI*, pages 174–186, 2010.

38 Guoqing Xu and Atanas Rountev. Precise memory leak detection for Java software using container profiling. In *ICSE*, pages 151–160, 2008.

39 Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, pages 160–173, 2010.

40 Guoqing Xu, Dacong Yan, and Atanas Rountev. Static detection of loop-invariant data structures. In *ECOOP*, pages 738–763, 2012.

41 YourKit Java Profiling. `http://www.yourkit.com`, 2015.

42 Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: A device-driver case. In *ASPLOS*, pages 193–206, 2014.

43 Pingyu Zhang and Sebastian Elbaum. Amplifying tests to validate exception handling code. In *ICSE*, pages 595–605, 2012.

44 Bowen Zhou, Jonathan Too, Milind Kulkarni, and Saurabh Bagchi. WuKong: automatically detecting and localizing bugs that manifest at large system scales. In *HPDC*, pages 131–142, 2013.