Static Detection of Asymptotic Performance Bugs in Collection Traversals

Oswaldo Olivo
University of Texas at Austin
olivo@cs.utexas.edu

Isil Dillig
University of Texas at Austin
isil@cs.utexas.edu

Calvin Lin
University of Texas at Austin
lin@cs.utexas.edu

Abstract

This paper identifies and formalizes a prevalent class of asymptotic performance bugs called *redundant traversal bugs* and presents a novel static analysis for automatically detecting them. We evaluate our technique by implementing it in a tool called CLARITY and applying it to widely-used software packages such as the Google Core Collections Library, the Apache Common Collections, and the Apache Ant build tool. Across 1.6M lines of Java code, CLARITY finds 92 instances of redundant traversal bugs, including 72 that have never been previously reported, with just 5 false positives. To evaluate the performance impact of these bugs, we manually repair these programs and find that for an input size of 50,000, all repaired programs are at least 2.45× faster than their original code.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; C.4 [*Performance of Systems*]: Analysis

General Terms Program analysis, Performance Bugs, Experimentation

1. Introduction

A *functionality bug* occurs when a piece of software crashes or produces an incorrect result. Fortunately, research in program analysis has produced significant advances in the automated detection of such bugs [1, 4–6, 22]. By contrast, a *performance bug* arises when a program produces the correct result but a simple functionality-preserving change can provide a substantial performance improvement [25]. Performance bugs are significant because they can render a program unusable; they can also be exploited by malicious users to create denial-of-service attacks. Unfortunately, performance bugs are more difficult to detect than functionality bugs for several reasons:

- First, it is difficult to know whether a program's performance can be expected to improve, since it depends on user inputs, on the many details of the program's execution environment, and on some notion of how a "good" solution should perform.

PLDI'15 , June 13–17, 2015, Portland, OR, USA
Copyright © 2015 ACM 978-1-4503-3468-6/15/06…$15.00
DOI: http://dx.doi.org/10.1145/(to come)

```
1. public boolean render(Graphics2D g2, Rectangle2D
2.                       dataArea, int index, ...) {
3.   ...
4.   XYDataset dataset = getDataset(index);
5.   XYItemRenderer renderer = getRenderer(index);
6.   ...
7.   int sCount = dataset.getSeriesCount();
8.   int series;
9.   for (series=sCount-1; series >= 0; series--) {
10.    int first = 0;
11.    int last = dataset.getItemCount(series) - 1;
12.    ...
13.    for (item=first; item <= last; item++) {
14.     renderer.drawItem(dataset, series, item,...);
15.    }
16.    ...
17.   }
18.   ...
19. }
20. public void drawItem(XYDataSet dataset,
21.                      int series, int item, ...) {
22.   ...
23.   OHLCDataset highLowData = (OHLCDataset)dataset;
24.   itemCount = highLowData.getItemCount(series);
25.   double xxWidth = dataArea.getWidth();
26.   for(int i=0; i< itemCount; i++) {
27.    ...
28.    if(last != -1) {
29.     xxWidth=Math.min(xxWidth,Math.abs(pos-last));
30.     }
31.   }
32. }
```

Figure 1. A previously unknown performance bug in the JFreeChart application that was identified by CLARITY.

- Second, while functionality bugs can be tested using assertions or various automated testing tools [6, 18, 33], the detection of performance bugs typically requires a human to monitor the program and make a judgment call on its performance.
- Third, performance bugs often manifest themselves only with large inputs, so the *small input hypothesis* [29], which forms the basis of most software testing methodologies, does not hold.

For these reasons, performance bugs remain a nebulous and evasive problem, and most existing tools for detecting performance problems either rely on rule-based pattern matching of syntactic program constructs or on some degree of runtime analysis and human intervention.

This paper presents a new static analysis—and its implementation in a tool called CLARITY—for automatically detecting an important class of asymptotic performance bugs. We say that a

code snippet has an *asymptotic performance bug* if its computational complexity is $O(f(n))$ but the same functionality can be implemented by code with complexity $O(g(n))$ such that $g(n)$ is $O(f(n))$ but $f(n)$ is not $O(g(n))$. Although the detection of arbitrary asymptotic performance bugs is beyond the scope of program analysis[1], we have identified a restricted but prevalent class of asymptotic performance bugs that we call *redundant traversal bugs*. A redundant traversal bug arises if a program fragment repeatedly iterates over a data structure, such as an array or list, that has not been modified between successive traversals of the data structure. Since such computation can be memoized and re-used across loop iterations, redundant traversal bugs typically result in at least an $O(n)$ performance degradation, where n is the size of the data structure. Furthermore, such performance bugs are typically easy to fix and often only require the addition of a parameter to a method, the addition of a field to an object, or the use of a slightly different data structure.

Motivating Example. As an example of a redundant traversal bug, consider the program snippet shown in Figure 1. This code is taken from version 1.0.17 of the JFreeChart software and exhibits a previously unknown performance bug uncovered by CLARITY. In particular, the `render` method (lines 1–19) plots a series of data items in the form (x, y) and invokes the `drawItem` method on line 14 to draw a single point within a given series. Here, the method invocation on line 14 is a virtual call with many possible targets, one of which is the `drawItem` method of `CandlestickRenderer` (lines 20–32).

The performance problem in this example arises because the `drawItem` method iterates over *all points* within the series in order to draw a *single* data point. In particular, the code traverses all data points to compute a value called `xxWidth`, which corresponds to the minimum gap between adjacent x-coordinates in the series. However, since the data set is not modified between successive calls to `drawItem`, the recomputation of `xxWidth` in each call to `drawItem` is redundant and needlessly traverses a potentially large list of data items many times. Hence, this code fragment exhibits a serious performance bug that can be fixed either by passing `xxWidth` as an argument to `drawItem` or by storing it as a field. Not only does such a fix result in a theoretical asymptotic performance improvement of $O(n)$, but it produces an order of magnitude performance improvement in practice.[2]

While it may seem surprising that such a blatant performance bug exists in a mature software project like JFreeChart, there are several reasons why this bug could be missed during development and testing. First, the impact of this performance bug is proportional to the size of the data series and requires data points to be drawn in the shape of candlesticks. Hence, test cases that either use small data series or render objects in a different shape, such as a square, will not reveal the performance bug. Second, the heavy-use of object oriented abstractions obscures the performance bug, making it difficult to spot the problem during manual code inspection. In particular, observe that the `drawItem()` method is virtual, and the performance bug only occurs in the `CandlestickRenderer` implementation. Similarly, the collection that is traversed is hidden behind an interface, so to identify the exact data structure, another virtual method call must be resolved. Finally, there is a function call depth of three between the loop that traverses the data structure and the access of the actual item in the data structure.

[1] Observe that identifying arbitrary asymptotic performance bugs requires knowing a "best" algorithm for implementing a given functionality.

[2] For example, using the existing test harnesses from SourceForge, we observe speedups of 8× to 11× when we fix this performance bug and modify the test harness to render each data item in the shape of a candlestick.

```
boolean containsAny1(HashSet<Foo> mySet,
                     ArrayList<Foo> myList) {
  for(Foo f: mySet)
    if(myList.contains(f))
      return true;
  return false;
}
```

Figure 2. Check if `myList` contains an element from `mySet`

Contributions. This paper makes the following contributions:

- We introduce and formalize the notion of *redundant traversal bugs*, which result in serious performance problems even in mature and well-tested software.
- We show that the detection of redundant traversal bugs is a non-trivial static analysis problem, and we present a novel and sound static analysis for automatically detecting this class of performance bugs.
- We implement the proposed analysis in a tool called CLARITY and experimentally demonstrate its effectiveness on nine open source Java code bases. Our tool is able to find 92 redundant traversal bugs, 72 of which were previously unknown. We also show that these performance bugs have significant impact in terms of program performance. For example, for an input size of 50,000, the repaired versions of these programs are at least 2.45× faster than the original code.

Organization. The remainder of this paper is organized as follows. Section 2 formally defines redundant traversal bugs and presents the core ideas for detecting such problems. After describing our algorithm for statically detecting redundant traversal bugs in Section 3, we briefly describe our implementation in Section 4 and present our empirical evaluation in Section 5. Before concluding, we compare our approach to previous work in Section 6.

2. Conceptual Foundations

This section defines the notion of redundant traversal bugs and explains the core ideas underlying our static analysis.

2.1 Defining Redundant Traversal Bugs

DEFINITION 1. **(Traversal)** *We say that a code snippet S* traverses *a data structure δ if it performs a computation whose average-case complexity is $\Omega(n)$, where n denotes the number of elements in δ.*

For instance, consider the `contains` methods provided by various data structures in the Java Collections Framework. According to Definition 1, the `contains` method of `ArrayList` performs a traversal of the data structure, as its average-case complexity is $O(n)$. On the other hand, while `HashSet`'s `contains` method has worst-case complexity $O(n)$, it is not considered a traversal because its average-case complexity is $O(1)$.

DEFINITION 2. **(Traversal Footprint)** *The* traversal footprint *of a code snippet S, written* $\mathrm{TraversalFP}(S)$*, is the set of data structures traversed by S.*

DEFINITION 3. **(Write Footprint)** *The* write footprint *of code S, written* $\mathrm{WriteFP}(S)$*, is the set of data structures that S modifies.*

DEFINITION 4. **(Redundant Traversal Bug)** *A loop L exhibits a* redundant traversal bug *if there exists a data structure δ such that:*

1. $\delta \in \mathrm{TraversalFP}(L)$ *and* $\delta \notin \mathrm{WriteFP}(L)$
2. *δ is traversed $\Omega(m)$ times in L, where m is the number of times that L executes*

```
boolean containsAny2(HashSet<Foo> mySet,
                     ArrayList<Foo> myList) {
  for(int i=0; i < myList.size(); i++) {
    Foo elem = myList.get(i);
    if(mySet.contains(elem))
      return true;
  }
  return false;
}
```

Figure 3. Different implementation of code in Figure 2.

In other words, a redundant traversal bug arises if a loop-invariant data structure is traversed a linear number of times within the loop. We believe that this definition captures the intuitive notion of redundancy, as the computation that is performed by traversing the data structure can be done once and re-used across all loop iterations.

EXAMPLE 1. *Figures 2 and 3 show the implementation of two methods called* `containsAny1` *and* `containsAny2` *that determine if the intersection of* `myList` *and* `mySet` *is non-empty. While* `containsAny1` *and* `containsAny2` *are functionally equivalent,* `containsAny1` *has a performance bug according to Definition 4 while* `containsAny2` *does not. In particular, since the* `contains` *method of* `ArrayList` *traverses the data structure,* `myList` *is part of the traversal footprint of the loop. By contrast, the* `contains` *method of* `HashSet` *does not perform a traversal; so, the traversal footprint of the loop from Figure 3 is empty. Thus,* `containsAny1` *contains an asymptotic performance bug because its average-case complexity is $O(n \cdot m)$, whereas the average-case complexity of* `containsAny2` *is $O(n)$ for a list of size n and set of size m.*

The need for the second condition of Definition 4 is illustrated by the following example.

EXAMPLE 2. *Consider the following code snippet, where the* `computeAvg` *method traverses its input:*

```
int calculate(ArrayList<ArrayList<int>> a) {
 int avgSum = 0;
 for(int i=0; i < a.size(); i++)
   avgSum += computeAvg(a.get(i));
 return avgSum;
}
```

Here, condition (1) of Definition 4 is satisfied because each element of `a` *is part of the traversal footprint but not the write footprint of the loop. However, this code does not have a performance bug because a* different *element of* `a` *is traversed in each loop iteration. Hence, condition (2) is violated.*

2.2 Core Ideas for Detecting Redundant Traversals

This section explains the key challenges underlying the static detection of redundant traversal bugs and outlines the core ideas behind our static analysis.

First, based on condition (1) of Definition 4, we see that a sound static analysis for detecting redundant data structure traversals must be able to perform the following task:

Given code snippet S, overapproximate the emptiness of the set $\theta \equiv \text{TraversalFP}(S) - \text{WriteFP}(S)$

Since θ is defined to be the difference of $\text{TraversalFP}(S)$ and $\text{WriteFP}(S)$, a sound static analysis for detecting redundant traversal bugs must *overapproximate* the traversal footprint but *underapproximate* the write footprint. Furthermore, since our analysis will track data structures in terms of program expressions, we need over- and under-approximating preconditions of program expressions with respect to a given program fragment. For this purpose, we define the following notions of *necessary* and *sufficient preconditions*:

DEFINITION 5. **(Necessary precondition)** *A set of expressions $\{e_1, \ldots, e_n\}$ is called a* necessary precondition *of an expression e with respect to a code snippet S, written $\text{pre}^+(e, S)$, if, for any constant c, the following Hoare triple is valid:*

$$\{e_1 \neq c \wedge \ldots \wedge e_n \neq c\}\ S\ \{e \neq c\}$$

In other words, for e to have value c after S, it is necessary that some element in $\text{pre}^+(e, S)$ has value c before S; hence, we refer to the set $\{e_1, \ldots, e_n\}$ as a necessary precondition of e with respect to S. Now, we also define a dual notion of *sufficient preconditions*:

DEFINITION 6. **(Sufficient precondition)** *A set of expressions E is called a* sufficient precondition *of expression e with respect to code S, written $\text{pre}^-(e, S)$, if, for all constants c and all $e' \in E$, the following Hoare triple is valid:*

$$\{e' = c\}\ S\ \{e = c\}$$

In other words, for e to have value c after S, it is sufficient that elements in $\text{pre}^-(e, S)$ have value c before S. Thus, sufficient conditions underapproximate the weakest precondition of an expression e with respect to code S.

EXAMPLE 3. *Consider the following code snippet S:*

```
if(*) x := y else x := z;
```

Here, we have $\text{pre}^+(x, S) = \{y, z\}$ and $\text{pre}^-(x, S) = \emptyset$. In particular, for `x` *to be equal to a certain value c after S, it is necessary that either* `y` *or* `z` *have value c before S. However, the sufficient precondition for* `x` *is $\emptyset$ because neither $y = c$ nor $z = c$ before S guarantees that $x = c$ after S.*

Now, given a statement S and a sub-statement π nested inside S, we will use the notation $S^-[\pi]$ to denote the code that comes before π in S. For instance, if S is the code:

```
x:=y; if(x>10) x++; y--; else x := 0
```

and π is the statement `y--`, then $S^-[\pi]$ is:

```
x:=y; assume(x>10); x++;
```

The following theorem explains why necessary and sufficient preconditions are useful for checking condition (1) of Definition 4.

THEOREM 1. *Let S be a code snippet containing two sets of statements Π_1 and Π_2 such that:*

1. *Each statement $\pi_i \in \Pi_1$ traverses a data structure referred to by program expression e_i*
2. *Each $\pi'_j \in \Pi_2$ modifies a data structure referred to by e'_j*

Then, $\text{TraversalFP}(S) - \text{WriteFP}(S) = \emptyset$ *if:*

$$\Big(\bigcup_{\pi_i \in \Pi_1} \text{pre}^+(e_i, S^-[\pi_i]) - \bigcup_{\pi'_j \in \Pi_2} \text{pre}^-(e'_j, S^-[\pi'_j])\Big) = \emptyset \quad (*)$$

PROOF 1. *Suppose* $\text{TraversalFP}(S) - \text{WriteFP}(S) \neq \emptyset$ *but $(*)$ holds. Then there must be some statement π that traverses a data structure δ that is referred to by expression e and δ is not modified in S. Let $\text{pre}^+(e, S^-[\pi]) = \{e_1, \ldots, e_n\}$. By definition of necessary precondition, this implies that $e_1 = \delta \vee \ldots \vee e_k = \delta$ before S. Now, since condition $(*)$ holds, every e_i is in the set*

$\text{pre}^-(e'_j, S^-[\pi'_j])$ *for some statement* π'_j *modifying data structure referred to by expression* e'_j*. By definition of sufficient precondition, this means that* $\{e_i = \delta\}\ S^-[\pi'_j]\ \{e'_j = \delta\}$*. But this implies that* δ *must also be modified, i.e., a contradiction.*

Theorem 1 is useful because it provides a method for statically checking condition (1) of Definition 4. In particular, to determine whether $\text{TraversalFP}(S) - \text{WriteFP}(S)$ *may* be empty, we compute necessary preconditions E of all program expressions that are traversed and sufficient preconditions E' of all expressions that are modified. If $E - E'$ is empty, then Theorem 4 implies that all expressions that are traversed are also modified; hence, we can rule out a potential redundant traversal bug. This is a key insight underlying our static analysis, and we will compute necessary preconditions of data structures that are traversed and sufficient preconditions for expressions that are modified in Section 3.

We now turn to the problem of statically checking condition (2) from Definition 4. That is, given a loop-invariant data structure δ that is traversed within the loop, is δ traversed at least a linear number of times? In our analysis, we will check this linearity requirement by over-approximating the following slightly stronger condition:

> Given a loop L and a data structure δ, is δ traversed in all iterations of L?

The above criterion is stronger than checking whether δ is traversed $\Omega(m)$ times in the loop. However, since our static analysis is path-insensitive, soundly answering the above question overapproximates condition (2) of Definition 4 for all practical purposes.

EXAMPLE 4. *Consider the following code snippet, where* `n` *is a positive integer and* `traverse` *performs list traversal:*

```
for(i=0; i<n; i++) {if(i%2 == 0) traverse(myList);}
```

Here, `myList` *is not traversed in all iterations, but it is traversed* $\Omega(n)$ *times. However, a sound static analysis that treats the test* `i%2 == 0` *as a non-deterministic choice will conclude that* `myList` *may be traversed in all iterations.*

The following theorem is useful in determining whether a data structure δ may be traversed in all loop iterations:

THEOREM 2. *Let* e *be a program expression, and let* E *be a necessary precondition of* e *with respect to code snippet* S*. Then, the following Hoare triple is valid for any constant* c*:*

$$\{e = c \ \wedge \bigwedge_{e_i \in E} e \neq e_i\}\ S\ \{e \neq c\}$$

PROOF 2. *First, note that the following implication is valid:*

$$e = c \wedge (\bigwedge_{e_i \in E} e \neq e_i) \Rightarrow (\bigwedge_{e_i \in E} e_i \neq c)$$

Now, by definition of necessary precondition, we have:

$$\{(\bigwedge_{e_i \in E} e_i \neq c)\}\ S\ \{e \neq c\}$$

Hence, the theorem holds by precondition strengthening.

Simply put, this theorem states that the value of an expression e has different values before and after executing S *provided that* e is distinct from every $e_i \in \text{pre}^+(e, S)$. To see the relevance of this theorem, suppose that e is a program expression that may be traversed in the loop. Now, if the value of e changes between any two consecutive loop iterations, then two different data structures δ and δ' are traversed; hence, δ is not traversed in all loop iterations. Thus, the key question to answer is whether the value of e can change between different loop iterations. Fortunately, we can answer this question using Theorem 2. Specifically, let E be the necessary precondition of e with respect to the loop body S. Based on Theorem 2, if we can prove that e is distinct from every $e_i \in E$, then we know that the same data structure is not traversed in all loop iterations.

Program P	:=	$\tau_1\ v_1; \ldots\ \tau_n\ v_n;\ S$
Type τ	:=	$\text{Int} \mid \text{Collection}\langle\tau_1, \tau_2\rangle$
Statement S	:=	$\text{skip} \mid v := e \mid v.\text{traverse}()$
		$\mid v_1 := v_2.\text{get}(v_3) \mid v_1.\text{put}^\rho(v_2, v_3)$
		$\mid S_1; S_2 \mid \text{if}(\star)\ \text{then}\ S_1\ \text{else}\ S_2$
		$\mid \text{while}(\star)\ \text{do}^\rho\ S$
Expression e	:=	$\text{int} \mid v \mid e_1 \oplus e_2 \quad (\oplus \in \{+, -, \times\})$

Figure 4. Language used for formal development.

EXAMPLE 5. *Consider again the code from Example 2, where* `computeAvg` *traverses the input array. Here, the loop traverses program expression* `a[i]`*, but the necessary precondition of* `a[i]` *with respect to the loop body is* {`a[i+1]`}*. Thus, assuming* `a[i]` *and* `a[i+1]` *do not alias, we can determine that condition (2) of Definition 4 is violated.*

3. Static Analysis

We now use the ideas introduced in Section 2 to describe our static analysis for detecting performance bugs.

3.1 Language

To formally describe our analysis, we use the small imperative language shown in Figure 4. This language contains two types of variables, namely, scalars of type Int and references of type Collection. We model collections as key-value stores that support insertion and retrieval of values associated with a given key. Hence, a variable of type $\text{Collection}\langle\tau_1, \tau_2\rangle$ models a key-value store where keys are of type τ_1 and values are of types τ_2. Observe that both keys and values may be of type Collection; hence, it is possible to nest an arbitrary number of data structures within another one.

In the language shown in Figure 4, statements include skip, assignments of the form $v := e$, and the following three collection-manipulating operations:

- A statement $v.\text{traverse}()$ traverses collection v, where traversal encompasses any operation that is consistent with Definition 1.
- A statement $v_1 := v_2.\text{get}(v_3)$ retrieves the value v_1 of key v_3 in the data structure pointed to by variable v_2.
- A statement $v_1.\text{put}^\rho(v_2, v_3)$, where ρ denotes a program point, associates value v_3 with key v_2 in the data structure referenced by variable v_1.

In addition, statements also include sequences $S_1; S_2$, if statements, and while loops. Since our analysis does not interpret conditionals (i.e., is path-insensitive), we model conditionals using non-deterministic choices indicated as $\star$ in Figure 4. Furthermore, we assume that while loops are annotated with a program point ρ which denotes the program location right before the first instruction in the loop body.

To simplify the formalization of our analysis, we omit function calls from this language and assume that the only way to traverse a data structure is by calling $v.\text{traverse}()$. In Section 4, we explain the inference of methods that traverse data structures as well as our interprocedural analysis.

Symbolic exp π	:=	$c \mid v \mid \pi_1\langle\pi_2\rangle$
Read footprint Φ	:=	2^π
Write footprint Ψ	:=	2^π
Alias environment $\mathcal{E}$	:=	$\rho \times \pi \to (2^\pi, 2^\pi)$

Figure 5. Summary of notation used in formalization

3.2 Computing Traversal and Write Footprints

We now describe our static analysis for over- and under-approximating each statement's traversal and write footprints. Our analysis is a backwards dataflow analysis and is presented in Figure 6 using judgments of the form:

$$\mathcal{E}, \Phi, \Psi \vdash S : \Phi', \Psi'$$

This judgment means that under the aliasing relations given by environment $\mathcal{E}$, if Φ and Ψ denote the traversal and write footprints after statement S, then Φ' and Ψ' over- and under-approximate the traversal and write footprints before S respectively. That is, assuming the correctness of $\mathcal{E}, \Phi$ and Ψ, the set Φ' over-approximates all collections that are traversed in or after S in terms of program expressions *before* S. Similarly, the set Φ' under-approximates all collections that must be modified in terms of program expressions before S.

As summarized in Figure 5, our analysis tracks traversal and write footprints using sets of symbolic expressions π. Symbolic expressions π can be constants c, variables v, or expressions of the form $\pi_1\langle\pi_2\rangle$, which represents the value associated with key π_2 in a data structure represented by expression π_1. For example, if the traversal footprint Φ of some statement S includes an expression $v\langle 3\rangle$, then the data structure stored at index 3 of the collection referenced by variable v may be traversed by statement S.

Since variables of type Collection are references in our language, our analysis must take possible aliasing relations into account if it is to soundly compute traversal and write footprints. Thus, our analysis rules utilize an *aliasing environment* $\mathcal{E}$ which maps each expression to its set of aliases. However, since our goal is to under-approximate write footprints, we need *must alias* facts as well as *may alias* information, so the aliasing environment $\mathcal{E}$ has signature $\rho \times \pi \to (2^\pi, 2^\pi)$, which maps each expression π and program point ρ to π's may- and must-aliases at program point ρ. In what follows, we will assume that such an aliasing environment $\mathcal{E}$ has been computed by performing may- and must-alias analyses prior to our footprint computation.

Let us now consider the analysis rules shown in Figure 6. In particular, rule (2) describes the analysis of assignments of the form $v := e$. Here, we replace any variable v used in Φ and Ψ by expression e because e is both a necessary and sufficient precondition for v with respect to statement S. For instance, for a statement $v_1 := v_2$, traversal footprint $\Phi = \{v_1, y\}$ and write footprint $\Psi = \{x\langle v_1\rangle\}$, our analysis computes $\Phi' = \{v_2, y\}$ and $\Psi' = \{x\langle v_2\rangle\}$.

Rule (3) describes the analysis of traversals of the form $v.\text{traverse}$. In this case, we simply add variable v to the traversal footprint Φ; the write footprint Ψ remains unchanged.

Rule (4) describes the analysis of retrieval (i.e., load) operations of the form $v_1 := v_2.\text{get}(v_3)$. Similar to the assignment rule, we replace variable v_1 in the traversal and write footprints with the expression $v_2\langle v_3\rangle$, which denotes the value associated with key v_3 in the collection referenced by v_2. Observe that $v_2\langle v_3\rangle$ is both a necessary and sufficient precondition for v_1 with respect to the statement $v_1 := v_2.\text{get}(v_3)$: In particular, the value of v_1 is equal to constant c after this statement if and only if $v_2\langle v_3\rangle = c$ before executing $v_1 := v_2.\text{get}(v_3)$.

(1) $$\frac{}{\mathcal{E}, \Phi, \Psi \vdash \text{skip} : \Phi, \Psi}$$

(2) $$\frac{\Phi' = \Phi[e/v] \quad \Psi' = \Psi[e/v]}{\mathcal{E}, \Phi, \Psi \vdash v := e : \Phi', \Psi'}$$

(3) $$\frac{\Phi' = \Phi \cup \{v\}}{\mathcal{E}, \Phi, \Psi \vdash v.\text{traverse}() : \Phi', \Psi}$$

(4) $$\frac{\begin{array}{c}\Phi' = \Phi[v_2\langle v_3\rangle/v_1] \\ \Psi' = \Psi[v_2\langle v_3\rangle/v_1]\end{array}}{\mathcal{E}, \Phi, \Psi \vdash v_1 := v_2.\text{get}(v_3) : \Phi', \Psi'}$$

(5) $$\frac{\begin{array}{c}\mathcal{E}(\rho, v_1) = (\mathcal{A}_1^+, \mathcal{A}_1^-) \quad \mathcal{E}(\rho, v_2) = (\mathcal{A}_2^+, \mathcal{A}_2^-) \\ \Phi' = \Phi[v_3/\mathcal{A}_1^+\langle\mathcal{A}_2^+\rangle] \cup (\Phi \ominus \mathcal{A}_1^-\langle\mathcal{A}_2^-\rangle) \\ \Psi' = \Psi[v_3/\mathcal{A}_1^-\langle\mathcal{A}_2^-\rangle] \cup (\Phi \ominus \mathcal{A}_1^+\langle\mathcal{A}_2^+\rangle)\end{array}}{\mathcal{E}, \Phi, \Psi \vdash v_1.\text{put}^\rho(v_2, v_3) : \Phi', \Psi' \cup \{v_1\}}$$

(6) $$\frac{\begin{array}{c}\mathcal{E}, \Phi, \Psi \vdash S_2 : \Phi', \Psi' \\ \mathcal{E}, \Phi', \Psi' \vdash S_1 : \Phi'', \Psi''\end{array}}{\mathcal{E}, \Phi, \Psi \vdash S_1; S_2 : \Phi'', \Psi''}$$

(7) $$\frac{\begin{array}{c}\mathcal{E}, \Phi, \Psi \vdash S_1 : \Phi_1, \Psi_1 \\ \mathcal{E}, \Phi, \Psi \vdash S_2 : \Phi_2, \Psi_2\end{array}}{\mathcal{E}, \Phi, \Psi \vdash \text{if}(\star) \text{ then } S_1 \text{ else } S_2 : \Phi_1 \cup \Phi_2, \Psi_1 \cap \Psi_2}$$

(8) $$\frac{\begin{array}{c}\Phi' \supseteq \Phi \quad \Psi \supseteq \Psi' \\ \mathcal{E}, \Phi', \Psi' \vdash \widetilde{S} : \Phi', \Psi'\end{array}}{\mathcal{E}, \Phi, \Psi \vdash \text{while}(\star) \text{ do } S : \Phi', \Psi'}$$

Figure 6. Analysis rules for computing traversal and write footprints. The notations $\mathcal{A}_1\langle\mathcal{A}_2\rangle$ and $\Phi[v/\mathcal{A}]$ are defined in Equations 1 and 2, and operator $\ominus$ is defined in Equation 3.

The most involved part of the analysis is Rule (5) for analyzing insertion (i.e., store) operations. To build intuition, let us first consider the statement $S = v_1.\text{put}(v_2, v_3)$ and an expression $x\langle y\rangle \in \Phi$. There are two cases to consider:

- If x must alias v_1 and y must alias[3] v_2, then the necessary precondition for $x\langle y\rangle$ is just $\{v_3\}$ since, for any value c, condition $v_3 \neq c$ before S guarantees $x\langle y\rangle \neq c$ after S.
- On the other hand, if x may alias v_1 and y may alias v_2 (but either may-alias relation is not also a must-alias relation), then the necessary precondition for $x\langle y\rangle$ is the set $\{x\langle y\rangle, v_3\}$. Observe that neither condition $x\langle y\rangle \neq c$ nor $v_3 \neq c$ before S on its own guarantees $x\langle y\rangle \neq c$ after S, as the value of $x\langle y\rangle$ may—but does not have to– be affected by S. However, if we know $x\langle y\rangle \neq c \wedge v_3 \neq c$ before S, we can guarantee that $x\langle y\rangle \neq c$ after S.

Now, let us also consider the analogous case where $x\langle y\rangle \in \Psi$. Again, there are two cases to consider:

- If x must alias v_1 and y must alias v_2, then the sufficient precondition for $x\langle y\rangle$ is just $\{v_3\}$ since, for any value c, condition $v_3 = c$ before S guarantees $x\langle y\rangle = c$ after S.
- Otherwise, if x may alias v_1 and y may alias v_2, then the sufficient precondition for $x\langle y\rangle$ is the empty set, as there is no

[3] Here, since y and v_2 may be scalars, we overload the term "alias" to also mean equality for scalars.

program expression whose value before S is guaranteed to be the same as the value of $x\langle y\rangle$ after S.

As this example illustrates, the computation of traversal and write footprints for store operations requires aliasing information for pointers (and equality information for scalars). With this intuition in mind, we now explain Rule (5) from Figure 6. As expected, we first need to look up the set of may- and must-aliases $(\mathcal{A}_1^+, \mathcal{A}_1^-)$ of v_1 as well as those of v_2 $(\mathcal{A}_2^+, \mathcal{A}_2^-)$. Now, any expression of the form $x\langle y\rangle$ may be affected by the statement $v_1.\mathrm{put}(v_2, v_3)$ if x is an alias of v_1 and y is an alias of v_2. Given set of symbolic expressions $\mathcal{A}$ and $\mathcal{A}'$, we use the notation $\mathcal{A}\langle\mathcal{A}'\rangle$ to represent:

$$[\![\mathcal{A}\langle\mathcal{A}'\rangle]\!] = \bigcup_{\pi\in\mathcal{A}} \bigcup_{\pi'\in\mathcal{A}'} \pi\langle\pi'\rangle \quad (1)$$

Hence, in Rule (5), $\mathcal{A}_1^+\langle\mathcal{A}_2^+\rangle$ yields the set of expressions that *may* be affected by the update, while $\mathcal{A}_1^-\langle\mathcal{A}_2^-\rangle$ represents expressions that *must* be overwritten.

Now, let us focus on the computation of the traversal footprint Φ' in the second line of Rule (5). Here, for a set $\mathcal{A} = \{\pi_1, \ldots, \pi_n\}$, we use the notation $\Phi[v/\mathcal{A}]$ as shorthand for:

$$\Phi[v/\{\pi_1, \ldots, \pi_n\}] = \Phi[v/\pi_1, \ldots, v/\pi_n] \quad (2)$$

Hence, the set $\Phi[v_3/\mathcal{A}_1^+\langle\mathcal{A}_2^+\rangle]$ is the same as Φ except that every (sub-) expression that may correspond to $v_1\langle v_2\rangle$ has been replaced with v_3. However, since we want to over-approximate the footprint, Φ' must also contain any expression π such that (i) $\pi \in \Phi$ and (ii) no prefix of π is in the set $\mathcal{A}_1^-\langle\mathcal{A}_2^-\rangle$, because such an expression π is not guaranteed to be killed by the store operation. To capture all expressions in Φ that are preserved by the statement $v_1.\mathrm{put}(v_2, v_3)$, we define an $\ominus$ operation on expression sets as follows:

$$\pi \in (\mathcal{A}_1 \ominus \mathcal{A}_2) \Leftrightarrow \pi \in \mathcal{A}_1 \wedge \forall\pi' \in \mathcal{A}_2.(\pi' \neq \mathrm{prefix}(\pi)) \quad (3)$$

In other words, $\mathcal{A}_1 \ominus \mathcal{A}_2$ preserves exactly those expressions π in $\mathcal{A}_1$ where π is not an extension of some expression in $\mathcal{A}_2$. Hence, the overall effect is two-fold:

- Φ' contains v_3 if Φ contains some expression $x\langle y\rangle$ where x and y may alias v_1 and v_2 respectively
- Φ' contains any expression $\pi \in \Phi$ such that π is not guaranteed to be modified by the statement $v_1.\mathrm{put}(v_2, v_3)$

We now explain the computation of the write footprint Ψ', which is described in the third line of Rule (5). First, observe that Ψ' contains v_3 iff there exists some $x\langle y\rangle \in \Psi$ such that x and y must alias v_1 and v_2. Furthermore, we kill all expressions in Ψ of the form $x'\langle y'\rangle$ where x' and y' may alias v_1 and v_2, respectively. Finally, since the statement $v_1.\mathrm{put}(v_2, v_3)$ modifies collection v_1, the write footprint before this statement includes variable v_1.

EXAMPLE 6. *Consider the following code snippet where x and y are variables of type* $\mathrm{Collection}\langle\mathrm{Int}, \mathrm{Collection}\langle\mathrm{Int}\rangle\rangle$ *and z, w are variables of type* $\mathrm{Collection}\langle\mathrm{Int}, \mathrm{Int}\rangle$*:*

1.	$y.\mathrm{put}(0, w);$	$\Phi_1 = \{x\langle 0\rangle, w\}$	$\Psi_1 = \{y, w\}$
2.	$z := x.\mathrm{get}(0);$	$\Phi_2 = \{x\langle 0\rangle\}$	$\Psi_2 = \{w\}$
3.	$w.\mathrm{put}(2, 5);$	$\Phi_3 = \{z\}$	$\Psi_3 = \{w\}$
4.	$z.\mathrm{traverse}();$	$\Phi_4 = \{z\}$	$\Psi_4 = \emptyset$

The annotations Φ_i and Ψ_i show the traversal and write footprints right before statement S_i under the assumption that x and y may alias (but are not guaranteed to).

We now consider the last two rules in Figure 6. When analyzing if statements in Rule (7), we take the union of the traversal footprints Φ_1 and Φ_2 obtained from the two branches. On the other hand, since we need to underapproximate the write footprint, we take the intersection of Ψ_1 and Ψ_2. rather than their union.

The final rule (8) describes the analysis of while loops[4]. In this rule, we use the notation $\widetilde{S}$ to denote the resulting statement when all traversal statements in S are replaced by skip. In particular, to avoid reporting the same warning for both inner and outer loops, our analysis ignores traversals in nested loops when computing the traversal footprint associated with an outer loop.

Now, continuing with rule (8), the traversal and write footprints Φ' and Ψ' must satisfy the following properties:

- Φ' must be a superset of Φ since any expression that is traversed after the loop may also be traversed before the loop (observe that the loop may execute zero times).
- Ψ' must be a subset of Ψ since only those expressions that are modified after the loop are guaranteed to be modified before the loop.
- Φ' and Ψ' must be inductive with respect to loop body $\widetilde{S}$.

EXAMPLE 7. *Consider the following code snippet:*

1. $\mathrm{while}(\star)\ \mathrm{do}\ i = i + 1;\ \mathrm{if}(\star)\ \mathrm{then}\ a = b\ \mathrm{else}\ b = a;$
2. $a.\mathrm{traverse}();\ b.\mathrm{put}(2, 5);$

Right before line 2, we have the traversal and write footprints $\Phi_2 = \{a\}, \Psi_2 = \{b\}$. On the other hand, the traversal and write footprints before line 1 are $\Phi_1 = \{a, b\}$ and $\Psi_1 = \emptyset$.

3.3 Detecting Redundant Traversal Bugs

We now explain how the computed traversal and write footprints are used to detect redundant traversal bugs. Figure 7 summarizes our performance bug detection algorithm using the judgments from Figure 6. As expected, our analysis only reports warnings when analyzing loops. Specifically, as shown on the first line of the ERR rule, we first compute the traversal and write footprints Φ, Ψ associated with the loop body. Now recall from Section 2 that the loop contains a redundant traversal bug if there exists a $\pi \in \Phi$ such that (i) π is traversed in all loop iterations, and (ii) π is loop invariant (i.e., is not modified within the loop).

To determine if condition (i) holds, we use Theorem 2 from Section 2 to check whether π is distinct from every expression $\pi' \in \mathrm{pre}^+(\pi, S)$. More specifically, in the ALL rule, $\mathcal{NC}$ corresponds to $\mathrm{pre}^+(\pi, S)$, and the check $\mathcal{A}^+ \cap \mathcal{NC} = \emptyset$ stipulates that π does not alias any expression in $\mathrm{pre}^+(\pi, S)$. Hence, by Theorem 2, if the predicate $\mathrm{traversal\_all}(S^\rho, \pi)$ evaluates to true, then π may be traversed in all loop iterations.

Now, we still need to check that π is not modified within the loop. For this purpose, we use the INV rule, which checks if some must-alias of π is in the write footprint Ψ. If not (i.e., $\mathcal{A}^+ \cap \Psi = \emptyset$), this implies that π may be loop invariant, so our analysis reports a potential performance bug.

EXAMPLE 8. *Consider the following code snippet:*

$$\mathrm{while}(\star)\ \mathrm{do}\ \ t := a.\mathrm{get}(i);\ t.\mathrm{traverse}();\ i := i + 1;$$

Here, the traversal footprint Φ for the loop body is $\{a\langle i\rangle\}$, and the write footprint Ψ is $\emptyset$. Since the necessary precondition of $a\langle i\rangle$ is $\Phi' = \{a\langle i + 1\rangle\}$, the ALL *rule fails (assuming $a\langle i\rangle$ and $a\langle i + 1\rangle$ do not alias), so our analysis does not report a performance bug.*

[4] This rule is only needed when detecting performance bugs in loops that contain at least one nested loop.

$$\frac{\mathcal{E}, \{\pi\}, \_ \vdash \widetilde{S} : \mathcal{NC}, \_ \quad \mathcal{E} \vdash (\rho, \pi) : (\mathcal{A}^+, \mathcal{A}^-) \quad \mathcal{A}^+ \cap \mathcal{NC} \neq \emptyset}{\mathcal{E} \vdash \text{traverse\_all}(S^\rho, \pi)} \ (\textsc{All})$$

$$\frac{\mathcal{E} \vdash (\rho, \pi) : (\mathcal{A}^+, \mathcal{A}^-) \quad \mathcal{A}^- \cap \Psi = \emptyset}{\mathcal{E}, \Psi \vdash \text{loop\_inv}(S^\rho, \pi)} \ (\textsc{Inv})$$

$$\frac{\mathcal{E}, \emptyset, \emptyset \vdash S : \Phi, \Psi \quad \pi \in \Phi \quad \mathcal{E} \vdash \text{traverse\_all}(S^\rho, \pi) \quad \mathcal{E}, \Psi \vdash \text{loop\_inv}(S^\rho, \pi)}{\mathcal{E} \vdash \text{while}(\star)\ \text{do}^\rho\ S \rightsquigarrow \textsc{Error}} \ (\textsc{Err})$$

Figure 7. Summary of performance bug detection. As before, $\widetilde{S}$ denotes traverse statements in S replaced by skip.

EXAMPLE 9. *Consider the following code snippet:*

1. $\text{if}(\star)$ then $b := a$; else skip;
2. $\text{while}(\star)$ do $a.\text{traverse}()$; $b.\text{put}(2, 4)$;

Here, we have $\Phi = \{a\}$ and $\Psi = \{b\}$. The necessary precondition of a with respect to the loop body is $\{a\}$, so using the ALL *rule, we determine that a may be traversed in all loop iterations. Furthermore, since a and b are not guaranteed to alias, the* INV *rule succeeds, so our analysis reports a redundant traversal bug. However, if we changed line 1 to $b := a$, then our analysis would not report a performance bug, since a and b are now guaranteed to alias.*

4. Implementation

We have implemented our proposed analysis in a tool called CLARITY, which is written in Java and consists of approximately 8,000 lines of code. CLARITY is implemented in the Soot compiler framework [39] and uses the Jimple intermediate representation. CLARITY relies on Soot for CFG and callgraph construction and issues a warning if any possible target of a virtual method call contains a performance bug. We have also implemented our own summary-based pointer analysis for computing may and must aliases by adapting the ideas described by Dillig et al. [14].

In this section, we focus on two important aspects of the implementation: (1) the identification of data structure traversals and (2) interprocedural analysis.

Identifying Data Structure Traversals. Our implementation uses a combination of automatic inference and a small amount of manual annotations to identify data structure traversals. In particular, we manually annotate 28 methods that (a) are implemented by the Java collections library and (b) have average-case complexity that is at least linear in the size of the data structure.[5] For instance, as we saw in Section 2, the `contains` method of `ArrayList` is annotated as a traversal, while `HashSet`'s `contains` method is not.

In addition to such manual annotations, CLARITY performs automated inference to identify code snippets that traverse data structures through iterators or loops. However, our current implementation does not detect data structure traversals in recursive functions. Hence, if a Java application implements its own tree traversal algorithm, CLARITY will not be able to detect performance bugs that arise from redundant traversals of this custom tree data structure.

[5] The annotated methods represent a tiny fraction of the analyzed methods.

To detect data structures that are traversed in loops, our static analysis also maintains a so-called *read footprint*. In particular, a data structure δ is added to the read footprint for code snippet S if S may access δ. For instance, using the notation from Section 3, if a code snippet accesses an element of array `a`, we simply add `a` to the read footprint $\mathcal{R}$ and compute $\mathcal{R}$'s necessary precondition during our backwards analysis. When we encounter a loop, we then check whether an element $a \in \mathcal{R}$ is accessed multiple times using Theorem 2. This analysis is similar to the check that tests whether a data structure is traversed multiple times (see the ALL rule in Figure 7). If a given data structure may be accessed in multiple loop iterations, we then add it to the traversal footprint Φ. The following example illustrates this analysis:

EXAMPLE 10. *Consider the following code snippet:*

$$\text{while}(\star)\ \text{do}\ t := l.\text{get}(i);\ \text{sum} := \text{sum} + t;\ i := i + 1;$$

While analyzing the loop body, we add variable l to $\mathcal{R}$. Then, when analyzing the while loop, we check whether l may be accessed in all loop iterations by testing whether the necessary precondition for l includes itself. Since it does in this case, l is added to the traversal footprint Φ of the while loop.

Interprocedural Analysis. Since the key idea underlying our technique is to compute traversal and write footprints, our analysis is amenable to modular reasoning. In particular, our interprocedural analysis computes a *procedure summary* for each method that tracks its read, traversal, and write footprints as well as transfer functions that give the necessary and sufficient preconditions for each data structure used in that method.

When we encounter a call to method m, we simply retrieve m's summary and *instantiate* its read, write, and traversal footprints by applying the appropriate formal-to-actual mapping. The instantiated callee footprints are then added to the corresponding footprints of the caller. In addition, the summary for each procedure also contains transfer functions that describe side effects of the callee. These transfer functions correspond to necessary and sufficient preconditions of program expressions with respect to the callee's body. Hence, for each expression e in the caller's read, write, and traversal footprints, we apply the appropriate transfer function to obtain the new footprints before the method call.

5. Experimental Evaluation

We evaluate CLARITY by applying it to nine mature and widely-used open source code bases (see Table 1). We conduct our experiments on an x86_64 Ubuntu 12.04 desktop machine with 8 GB of RAM and a dual-core 3 GHz processor. We evaluate our approach using the following metrics: (1) the number of performance bugs detected by CLARITY, (2) the number of false positives reported, and (3) the impact of fixing these bugs.

Table 1 summarizes the results of our experimental evaluation, with the benchmarks listed in order of increasing code size. Our benchmarks range from 21,396 to 226,623 lines of Java source code and contain between 715 and 12,338 methods (including external library calls). Since CLARITY also analyzes the external libraries called by each application, the actual number of lines of code analyzed by the tool can be much larger (see third column of Table 1). We see that even though CLARITY performs a non-trivial interprocedural static analysis, the running time of the tool is quite reasonable, with LWJGL taking the longest at 26.4 minutes. Note that the reported times include pointer analysis as well as the time required to analyze library code.

Most significantly, we see that CLARITY reports a total of 92 performance bugs, with only five of these being false positives. Furthermore, among the 92 true positives, 72 are previously unre-

Name	LoC	LoC w/ libraries	Number of Methods	Analysis Time (sec)	Reported Bugs	Previously Unknown Bugs	False Positives
Charts4j	21,396	28,667	715	122	0	0	0
Janino	39,832	305,660	7,149	556	3	3	0
Apache Collections	58,186	58,186	3,759	180	19	10	4
Ode4J	83,207	83,207	4,048	128	0	0	0
Java3D	115,859	146,376	1,875	335	0	0	0
Guava (Google Core)	129,745	129,745	12,338	338	55	44	1
LWJGL (Game Library)	202,902	267,248	10,740	1,584	10	10	0
Apache Ant	205,371	265,828	10,029	1,325	2	2	0
JFreeChart	226,623	362,584	9,162	602	3	3	0
Total	1,083,121	1,647,501	59,815	5,470	92	72	5

Table 1. Experimental results: Performance Bugs Detected.

ported according to the SourceForge development logs. The numbers in this table include only unique performance bugs; that is, performance bugs that are inherited by a sub-class are not counted multiple times.

Finally, to evaluate CLARITY's performance impact, we repair each of the identified performance bugs, for example, by adding additional fields to classes, passing extra parameters to methods, or transforming data structures (e.g., lists into sets). We then compare the execution time of the original and repaired programs on input sizes ranging from a few thousand to a few hundred thousand elements. In this evaluation, we observe speedups in the range 2.5-548.2× on inputs sizes of 50,000 and speedups between 6.6-3,350× on input sizes of 100,000. This empirical comparison between the original and modified programs confirms that the redundant traversals identified by CLARITY indeed correspond to asymptotic performance problems.

5.1 Discussion

We now explain the most commonly reported performance bugs and share some observations about the nature of the performance bugs detected by CLARITY.

RetainAll Performance Bug. The *RetainAll* bug occurs in code that removes all elements in a collection A that are not also present in another collection B (often passed as a parameter). The inefficiency occurs when B has a slow containment checking method that is invoked a linear number of times. This bug can typically be fixed by converting the parameter collection B to a set, either within the algorithm or at its call site, or by more complicated means such as keeping an iterator on the parameter collection and advancing it accordingly to avoid redundant checks. In addition to appearing in doubly-nested loops, this bug can also appear in other ways, such as the pruning of multi-maps, removal of data points from a plot, and intersection of build resources while compiling a Java application. In fact, we observe some variant of the *RetainAll* bug in four code bases, namely, Apache Ant, Guava, JFreeChart, and Apache Collections.

ContainsAll Performance Bug. The *ContainsAll* bug is similar to *RetainAll* and occurs in code that checks whether a collection contains all the elements in some other collection, which is often a method parameter. As in the *RetainAll* bug, we see that the flexibility of generic types for collection parameters can lead to severe performance degradation. We found several instances of this performance bug in Guava and Apache Collections.

FilterPredicate Bug. This bug occurs when a containment predicate P is attached to a collection C, and elements can only be added to C if they satisfy P. Typically, the root cause of the performance problem is an inefficient data structure used in the implementation of the predicate. We see several instances of this bug in the Guava libraries.

TransformPredicate Bug. This type of performance bug is similar to the *FilterPredicate* bug. It appears when an inefficient predicate is used to identify elements that should be removed from a collection. Again, we find several occurrences of this type of performance bug in the Guava libraries.

ExtremeVal Bug. The *ExtremeVal* performance bug occurs when the maximum or minimum value of a list of elements is computed multiple times, even though the list does not change. An instance of this type of bug is the JFreeChart example from Figure 1.

PatternMatching Bug. This bug occurs when checking a set of elements against a set of patterns. The redundant traversals could be avoided by combining the set of patterns into one pattern, simplifying it, and then checking the elements against this pattern. An instance of this bug arises in Apache Ant when testing if files in a directory satisfy a regular expression describing an include-path.

False Positives. Four of the false positives in our experiment arise from imprecise virtual method call resolution. For example, in some cases, CLARITY believes that the target of a virtual method call may be `LinkedList::contains` even though it can only be `HashMap::contains`. The fifth false positive arises when CLARITY believes that a data structure is traversed multiple times in a loop that can be traversed only once. In this case, the code has a non-trivial loop invariant that CLARITY cannot reason about.

Nature of Performance Bugs. We conclude this section with some observations about the nature of performance bugs uncovered by CLARITY.

First, while the majority of the bugs detected by CLARITY are *conceptually* quite simple, they are not easily identifiable through either manual code review or simple static analysis. Due to the heavy use of abstraction in Java, the collection that is traversed is often hidden behind an interface, so identifying data structures that are accessed requires virtual method call resolution. Furthermore, the loop where the performance bug arises is typically defined in a different class or method from the code that actually traverses the data structure. Hence, the detection of such bugs requires fairly sophisticated interprocedural static analysis.

Second, even though there are conceptual similarities between the performance bugs identified by CLARITY, different code snippets exhibiting conceptually similar bugs can be syntactically *very different*. For example, a performance bug that is classified in the *RetainAll* category appears in a method called `createConsolidatedPieDataset`, which tries to create a new dataset with keys above a certain threshold. Meanwhile, another instance of the *RetainAll* bug appears in the Apache collections in a method called `retainAll` and looks very different from the

JFreeChart instance of the *RetainAll* bug. Thus, despite conceptual similarities among various performance bugs detected by CLARITY, these bugs cannot be detected by performing syntactic pattern matching on program constructs.

Finally, our empirical evaluation sheds light on the difficulty of detecting these performance bugs during testing. First, several performance bugs identified by CLARITY arise in rarely executed program paths. For instance, recall the performance bug from Figure 1, which is triggered when the user renders items in the shape of a candlestick. Since this shape is unlikely to be a popular choice, it is unlikely to be triggered during testing. Second, as observed in our empirical performance comparison between the original and repaired programs, the true cost of a performance bug may not be apparent unless tested with large inputs. Since most test suites are designed with the *small input hypothesis* in mind, they are unlikely to reveal these performance problems.

6. Related Work

Automated Performance Bug Detection. Several recent projects use program analysis to automatically detect performance bugs. Some of these detect wasteful use of temporary objects [16, 35, 43, 44], others focus on inefficient or incorrect usage of collection data structures [34, 41, 42], and some use dynamic profiling to identify expensive computation that can be memoized [31].

The Toddler tool [32] uses dynamic instrumentation to identify "likely redundant" computation by monitoring repetitive and partially-similar memory access patterns. Like Toddler, our work builds on the observation that repetitive traversal of collections likely constitutes a performance bug. Unlike Toddler, our method is purely static, so it incurs no run-time overhead and does not require that the programmer provide representative performance tests.

The PerfChecker tool [28] statically analyzes Android applications to identify common performance bugs. Unlike CLARITY, PerfChecker detects performance bugs related to GUI lagging, energy leaks, and memory bloat.

The X-ray tool [3] helps users diagnose performance problems related to configuration settings. X-ray uses a technique called *performance summarization*, which couples performance costs with dynamic information flow analysis. Unlike CLARITY, X-ray performs dynamic analysis and focuses on performance problems caused by user rather than developer error.

Trace analysis is a technique for identifying root causes of performance anomalies [15, 45]. For example, the TraceAnalyzer tool [15] constructs performance traces that capture the time-varying performance of program runs. Another approach [45] performs impact and causality analysis on traces to discover patterns that are correlated with performance problems. These techniques can shed light on a wide variety of performance anomalies, but they are not fully automated.

Classification and Impact of Performance Bugs. Jin et al. present a comprehensive study of performance bugs and propose a variety of rules to detect and repair likely performance bugs [25]. These rules, which are inspired from existing patches, perform pattern-matching over syntactic program constructs and require domain-specific knowledge about the classes of performance bugs that exist in a given application. A pattern-matching technique is also proposed in the context of databases [8].

Song and Lu use five open-source applications to study the use of statistical debugging for finding performance bugs [36]. They find that two kinds of statistical models involving branch predicates can help pinpoint root causes of performance problems. The idea is to use existing bug reports to gather similar efficient and inefficient computations and compute statistically significant predicates. While quite general, this approach relies on existing bug reports and on user-provided test parameters.

Zaman et al. find that, for Mozilla Firefox and Google Chrome, developers typically spend more time fixing performance bugs than functionality bugs [46].

Loop-Invariant Code Motion. The removal of redundant traversal bugs bears some similarity to loop invariant code motion (LICM), but the problems are quite different. LICM is typically applied to individual assignment statements, and it uses a low-level notion of loop-invariance that is based on reaching definitions. Thus, LICM is not capable of identifying redundant data structure traversals that are detected by our analysis. Of course, LICM also performs the actual optimization, rather than simply detecting the inefficiency.

Techniques for Estimating Computational Complexity. Recent work uses sophisticated static analyses to automatically estimate worst-case resource usage—such as running time—of programs [20, 21, 23, 26], and the TrendProfiler tool uses profiling and dynamic analysis to estimate *empirical computational complexity* [19]. While these approaches can help programmers debug and understand performance problems, they do not automatically pinpoint them.

Necessary and Sufficient Preconditions. To detect redundant traversal bugs, our algorithm constructs dual over- and under-approximations of the program. Other static analyses that involve negation (or set complement) also make simultaneous use of necessary and sufficient conditions. In particular, the interplay between over- and under-approximations has been explored in path-sensitive static analysis [11], in precise reasoning for unbounded data structures [12, 13], in the construction of method summaries [9], for analysis of confidentiality properties [7], and in typestate analysis [17].

May and Must Alias Analysis. *May alias* analysis [10, 27, 37, 38, 40] underlies almost any compiler optimization, bug detection, and verification technique. While not quite as common as may-alias analysis, *must-alias* analysis is also considered in several papers [2, 24, 30]. Our work simply utilizes may- and must-alias information and does not make contributions in this area.

7. Conclusions

Modern software is written in languages such as Java and C# and makes heavy use of abstractions that allow difficult-to-detect performance bugs to creep into mature code bases. While modern compilers perform many types of sophisticated optimizations, they are ill-equipped to deal with performance bugs that cross many procedure and abstraction boundaries.

In this paper, we have introduced CLARITY, a purely static tool that effectively detects redundant traversal bugs. Our application of CLARITY to nine open source Java code bases identifies a large number of previously unknown performance bugs with a small number of false alarms. Once CLARITY identifies a redundant traversal bug, the repair is typically straightforward. Furthermore, as confirmed by our empirical evaluation, the repaired programs enjoy a significant performance benefit over the original programs.

Although CLARITY targets a restricted class of performance problems, it is nonetheless a significant step towards automated static detection of performance bugs. In future work, we plan to expand the class of performance bugs that can be automatically detected. We also plan to explore techniques for automatically repairing the performance bugs identified by CLARITY.

Acknowledgments. We thank Thomas Dillig and Jia Chen for their valuable comments on early versions of this paper. This work

was supported by NSF grants CNS-1138506 and DRL-1441009, and Air Force Research Laboratory under agreement number FA8750-12-2-0020

References

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn Project. In *PASTE*, pages 43–48, 2007.

[2] R. Z. Altucher and W. Landi. An extended form of must alias analysis for dynamic allocation. In *PLDI*, pages 74–84. ACM, 1995.

[3] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, pages 307–320, 2012.

[4] T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[5] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30 (7):775–802, 2000.

[6] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[7] P. Černỳ and R. Alur. Automated analysis of Java methods for confidentiality. In *CAV*, pages 173–187. Springer, 2009.

[8] T. Chen, W. Shang, Z. Jiang, A. Hassan, M. Nasser, and P. Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *ICSE*, pages 1013–1024. ACM, 2014.

[9] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.

[10] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI*, pages 230–241. ACM, 1994.

[11] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI*, volume 43, pages 270–280, 2008.

[12] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, pages 246–266. 2010.

[13] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *ACM SIGPLAN Notices*, volume 46, pages 187–200. ACM, 2011.

[14] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–577. ACM, 2011.

[15] A. Diwan, M. Hauswirth, T. Mytkowicz, and P. F. Sweeney. Traceanalyzer: a system for processing performance traces. *Software: Practice and Experience*, 41(3):267–282, 2011.

[16] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *FSE*, pages 59–70, 2008.

[17] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):9, 2008.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *ACM SIGPLAN Notices*, volume 40, pages 213–223. ACM, 2005.

[19] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *FSE*, pages 395–404, 2007.

[20] S. Gulwani. Speed: Symbolic complexity bound analysis. In *Computer Aided Verification*, pages 51–62. Springer, 2009.

[21] S. Gulwani, K. K. Mehra, and T. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *ACM SIGPLAN Notices*, volume 44, pages 127–139. ACM, 2009.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Model Checking Software*, pages 235–239. Springer, 2003.

[23] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In *ACM SIGPLAN Notices*, volume 46, pages 357–370. ACM, 2011.

[24] S. Jagannathan, P. Thiemann, S. Weeks, and A. Wright. Single and loving it: Must-alias analysis for higher-order languages. In *PLDI*, pages 329–341. ACM, 1998.

[25] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47 (6):77–88, 2012.

[26] J. Knoop, L. Kovács, and J. Zwirchmayr. Symbolic loop bound computation for WCET analysis. In *Perspectives of Systems Informatics*, pages 227–242. Springer, 2012.

[27] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, 1992.

[28] Y. Liu, C. Xu, and S. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, pages 1013–1024, 2014.

[29] D. Marinov and S. Khurshid. TestEra: a novel framework for automated testing of Java programs. In *16^{th} IEEE Conference on Automated Software Engineering*, page 22, 2001.

[30] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. *ACM SIGPLAN Notices*, 42(1):327–338, 2007.

[31] K. Nguyen and G. Xu. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 268–278. ACM, 2013.

[32] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571. IEEE, 2013.

[33] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference*, pages 263–272, 2005.

[34] O. Shacham, M. Vechev, and E. Yahav. Chameleon: adaptive selection of collections. In *ACM SIGPLAN Notices*, volume 44, pages 408–418. ACM, 2009.

[35] A. Shankar, M. Arnold, and R. Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In *ACM SIGPLAN Notices*, volume 43, pages 127–142. ACM, 2008.

[36] L. Song and S. Lu. Statistical debugging for real-world performance problems. In *OOPSLA*, NY, USA, 2014. ACM.

[37] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. *ACM SIGPLAN Notices*, 41(6):387–400, 2006.

[38] B. Steensgaard. Points-to analysis in almost linear time. In *PLDI*, pages 32–41. ACM, 1996.

[39] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[40] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. *ACM SIGPLAN Notices*, 30(6):1–12, 1995.

[41] G. Xu. CoCo: sound and adaptive replacement of Java collections. In *ECOOP 2013–Object-Oriented Programming*, pages 1–26. Springer, 2013.

[42] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. *ACM SIGPLAN Notices*, 45(6):160–173, 2010.

[43] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *ACM SIGPLAN Notices*, volume 44, pages 419–430. ACM, 2009.

[44] G. Xu, M. Arnold, A. Rountev, and G. Sevitsky. Finding low utility data structures. In *PLDI*, pages 174–186. ACM, 2010.

[45] X. Yu, S. Han, D. Zhang, and T. Xie. Comprehending performance from real-world execution traces: A device-driver case. In *ASPLOS*, pages 193–206. ACM, 2014.

[46] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *Mining Software Repositories*. ACM, 2012.