# Directed Test Generation to Detect Loop Inefficiencies

Monika Dhok
Indian Institute of Science, Bangalore, India
monika.dhok@csa.iisc.ernet.in

Murali Krishna Ramanathan
Indian Institute of Science, Bangalore, India
muralikrishna@csa.iisc.ernet.in

## ABSTRACT

Redundant traversal of loops in the context of other loops has been recently identified as a source of performance bugs in many Java libraries. This has resulted in the design of static and dynamic analysis techniques to detect these performance bugs automatically. However, while the effectiveness of dynamic analyses is dependent on the analyzed input tests, static analyses are less effective in *automatically* validating the presence of these problems, validating the fixes and avoiding regressions in future versions. This necessitates the design of an approach to automatically generate tests for exposing redundant traversal of loops.

In this paper, we design a novel, scalable and automatic approach that addresses this goal. Our approach takes a library and an initial set of coverage-driven randomly generated tests as input and generates tests which enable detection of redundant traversal of loops. Our approach is broadly composed of three phases – analysis of the execution of random tests to generate method summaries, identification of methods with potential nested loops along with the appropriate context to expose the problem, and test generation to invoke the identified methods with the appropriate parameters. The generated tests can be analyzed by existing dynamic tools to detect possible performance issues.

We have implemented our approach on top of the SOOT bytecode analysis framework and validated it on many open-source Java libraries. Our experiments reveal the effectiveness of our approach in generating 224 tests that reveal 46 bugs across seven libraries, including 34 previously unknown bugs. The tests generated using our approach significantly outperform the randomly generated tests in their ability to expose the inefficiencies, demonstrating the usefulness of our design. The implementation of our tool, named GLIDER, is available at http://drona.csa.iisc.ac.in/~sss/tools/glider.

## CCS Concepts

•**Software and its engineering** → *Software performance; Software testing and debugging;*

*FSE'16, November 13-19, 2016, Seattle, WA, USA*
© 2016 ACM. ISBN 978-1-4503-4218-6/16/11...$15.00
DOI: http://dx.doi.org/10.1145/2950290.2950360

## Keywords

redundant traversal bugs, performance, testing

## 1. INTRODUCTION

Performance is of significant importance for any software application. Unfortunately, underlying performance issues are hard to detect in-house during testing and usually manifest in the field [28]. Not surprisingly, these issues are found even in well tested commercial products [1, 2]. Since these bugs degrade the application responsiveness, techniques that improve the possibility of detecting these problems before deployment are helpful.

Many effective techniques are developed to detect performance bugs automatically. These techniques address various kinds of performance related issues including repetitive computations [30, 31], redundant loops [28], object bloat [27], latent performance bugs [21], and performance issues in clouds and smart phones [10, 29]. The performance problems due to repetitive and similar computations across iterations have been found in many mature codebases [19, 39]. This has resulted in the design of efficient static [31] and dynamic analysis [30, 28] techniques to detect these problems.

```
--------------------------------------------------------------
1. public class A {
2.  public boolean containsAny(Collection c1, Collection c2) {
3.      Iterator itr = c1.iterator();
4.      while(itr.hasNext())
5.          if(c2.contains(itr.next()))
6.            return true;
7.      return false;
8.    }
9.}
--------------------------------------------------------------
```

**Figure 1 Example**

Figure 1 illustrates the problem of redundant traversal. Here, class `A` has `containsAny` method which accepts collections, `c1` and `c2`, as input. The method iterates over `c1` to check whether one of its elements is present in `c2`. This innocuous looking code can result in poor performance. If `containsAny` is invoked with a non-empty hashset and arraylist as parameters respectively, repeated invocation of `contains` method (line 5) can result in a slowdown. This is because the implementation of `contains` in `ArrayList` has *linear* complexity as it traverses the list to check the presence of the element. For each iteration of the outer loop (line 4), the elements in `c2` are *unnecessarily* traversed. This redundant traversal [31] can be addressed using memoization [12], for example, by ensuring that `c2` is hashset in this case. However, the core problem is to identify their existence.

TODDLER [30] employs dynamic analysis to detect loops which perform repetitive memory accesses. It takes a set of tests as input and monitors their execution to detect repetitive accesses across loop iterations. While unit tests provided by developers can be used for this purpose, these tests can be less effective in revealing the entire gamut of defects due to redundant traversals. This is because the unit tests are usually directed towards ensuring functional correctness [13] and not necessarily to expose redundant traversals. Therefore, if the test does not cover the loops performing repetitive memory accesses, analyzing its execution becomes less useful. Manually writing the tests to help TODDLER expose the inefficiencies can be an arduous task.

In order to overcome the dependence on tests, CLARITY [31] employs static analysis to detect inefficient loops. Unfortunately, the defects reported by static analyses need to be triaged by a programmer to ensure the validity of the reported defects [24, 5]. In comparison, if tests that expose the defects are available, they can serve multiple purposes – (a) help confirm the validity of the bug based on the execution time, (b) the bug fix can be automatically validated by comparing execution times of the two program versions, and (c) can be integrated into the testsuite to prevent any regressions in future revisions. Therefore, to realize these benefits without investing manual effort, techniques for directed generation of tests to expose loop inefficiencies are desirable.

There are various challenges involved in generating tests to detect these defects. *Firstly*, due to virtual call resolution, a method invocation can be resolved to various methods depending upon the type of the receiver. Generating tests for all the possible resolutions of all the invocations is not scalable. *Secondly*, the realization of the defects can be dependent on conditions that can affect the reachability of the problematic loop. Therefore, it is essential that the generated test has the appropriate context to ensure reachability. *Finally*, the problem can manifest only when the data structure being traversed has large number of elements arranged in a *specific* order. For example, if the elements are ordered in a manner that the loop is not traversed completely, then the redundant traversal cannot be detected.

We elaborate these challenges using the example in Figure 1. To expose the underlying problem, the generated test must have the following characteristics:

- invoke the method `containsAny` with non-empty input collections so that both the loops execute sufficient number of times,
- pass an object of type ArrayList as second parameter, and
- use *distinct* elements in the input collections so that the loop does not break at line 6.

Tests that satisfy these conditions will perform redundant traversal of the list exposing the underlying problem. Our goal is to automatically generate these tests.

In this paper, we address the goal of automatically generating useful tests to detect inefficient loops in libraries. Our approach takes a library and a randomly generated [14, 32] testsuite as input and generates a testsuite that helps expose the inefficient loops in the library. Our approach is composed of three phases – *summary generation* phase which generates method summaries, *method detection* phase which identifies methods whose invocation causes the inefficiencies, and *test generation* phase which generates tests that invoke the identified methods with appropriate parameters under a suitable context. These tests can be analyzed by TODDLER [30] to expose the underlying defects.

More elaborately, in the *summary generation* phase, we generate a random test suite for the library using existing test generation tools (EVOSUITE [14], RANDOOP [32]). Our analysis instruments these tests and analyzes their execution to derive method summaries. A method summary is composed of information pertaining to the presence of loops, the object traversed in each loop, and methods (and the parameters) invoked. For example, the summary of `containsAny` method in the Figure 1 will represent the loop iterating on `c1` and the invocation of the `contains` method on `c2`. We leverage the method summaries to *direct* the test generation process so that tests exposing loop inefficiencies alone are generated. In other words, this phase prunes the large state space of method sequences that can be invoked.

In the *method detection phase*, our analysis traverses the callgraph and identifies methods that may execute inefficient loops when invoked. For example, while traversing the callgraph for the class corresponding to the example in Figure 1, this phase detects that there exist a nested loop if the method is invoked with `c2` of type ArrayList. It also keeps track of the addition of objects to collection/array fields by different methods and identifies these methods as *populator* methods. In the *test generation* phase, we generate the test to invoke the identified methods in the previous phase. We create the required objects (receiver and parameters) and ensure that the method is invoked with appropriate types. Moreover, we set the context based on the concrete execution from the initial set of tests so that the inefficient loops are reachable. Test generator also populates the collection/array fields (on which the loop traverses) with *distinct* patterns and sizes using the populator methods detected in the previous phase. The generated test will highly likely expose the redundant traversal problem in the method.

We have implemented our approach on the `soot` bytecode analysis framework [41]. We perform elaborate experimentation and analyze seven popular Java libraries, including `Apache-collection` and `Guava`. Our implementation generates 224 tests that enables detection of 46 bugs, including 34 previously unknown bugs. A few bugs reported by us are confirmed as real bugs by the developers of these libraries. We fixed the bugs (in-house) and executed the generated tests on the original and fixed versions. We observed performance gains of 20 to 60% even with just 100 elements in the collection objects on which the redundant traversal occurs. Our experimental results also demonstrate that more than 95% of the generated tests help reveal a defect. The implementation of our tool, named GLIDER, is available at http://drona.csa.iisc.ac.in/~sss/tools/glider.

The paper makes the following technical contributions:

- We present a novel and effective approach to generate tests for detecting inefficient loops in Java libraries.
- Our analysis proposes a novel dynamic analysis to generate method summaries which are subsequently used to identify methods with redundant traversals. These identified methods are invoked, with appropriate parameters, as part of the tests generated by our approach.
- We implement the proposed approach on the `soot` bytecode analysis framework and present the implementation details necessary for our approach to be practical.

- We validate our approach on seven open-source Java libraries and generate 224 tests that enable the detection of 46 bugs, including 34 previously unknown bugs.

## 2. MOTIVATION

In this section, we motivate the need for our approach by using a real example from the latest version (1.0.19) of JFreeChart library, a free Java chart library that helps developers generate professional quality charts.

```
CategoryPlot.java
------------------------------------------------------------
1. public class CategoryPlot{
2. private Map<Integer, CategoryDataset> datasets;
3. public CategoryPlot() { ... }
4. public void setDataset(int index, CategoryDataset set) {
            ...
5.          this.datasets.put(index, set);
            ...
6.  }
7. public int indexOf(CategoryDataset dataset) {
8.       for (Entry entry: this.datasets.entrySet())
9.           if (entry.getValue() == dataset)
10.               return entry.getKey();
11.       return -1;
12.  }
13. private List<CategoryDataset> datasetsMapped(int idx){
14.       for (CategoryDataset set:this.datasets.values()){
               ...
15.            int i = indexOf(set);
               ...
16.      }
17.  }
18. public List getCategoriesForAxis(CategoryAxis axis) {
20.       int axisIndex = getDomainAxisIndex(axis);
21.       datasetsMapped(axisIndex))
           ...
22. }
24. }
```

**Figure 2 Motivating example.**

Figure 2 presents a simplified implementation of `CategoryPlot` class. This implementation can cause an execution slowdown when used in other applications because of the redundant computations in the method `datasetsMapped` (lines 13-17). This method traverses over `datasets` (line 14). During traversal, this method invokes another method `indexOf` which also traverses over the same data structure (line 8). This results in O($n^2$) complexity, where $n$ is the number of elements present in `datasets`. If `datasets` is populated with large number of objects, we can observe a significant slowdown. To detect this problem, a test should invoke `datasetsMapped` method while ensuring that `datasets` contains large number of elements. Apart from these requirements, there are intricate challenges in designing the test:

- The programmer needs to find appropriate callsite to invoke `datasetsMapped` which is a private method.
- The programmer should have detailed knowledge of the class hierarchy. For instance, in the generated test, objects of type `CategoryPlot`, `CategoryAxis` and `CategoryDataset` need to be instantiated.
- To add large number of elements to `datasets`, the programmer needs to identify the appropriate method.
- The programmer needs to update `datasets` with appropriate parameters. Updating a map by adding many elements at the same index will not be useful.

Our implementation overcomes these challenges and automatically generates the relevant test. To start with, the approach generates random set of tests for `CategoryPlot` class. In the *summary generation* phase, we monitor the execution of these tests and derive method summaries. Further, in the *method detection phase*, we identify methods that executes nested loops (which includes the `datasetsMapped` method) and the populator method that updates the collections (for e.g., `datasets`) of the class. Subsequently, using static analysis we identify methods to create objects that can be used to invoke the inefficient method. Finally, we use this information to synthesize the test. Figure 3 shows a sample test that exposes the aforementioned problem.

```
TestCategoryPlot.java
----------------------------------------------------
1.public class TestCategoryPlot{
2. public static void main(String[] args) {
3.  CategoryPlot categoryPlot = new CategoryPlot();
4.  for(int i = 0; i < args[0]; i++) {
5.   DefaultCategoryDataset defaultData;
6.   defaultData = new DefaultCategoryDataset();;
7.   SlidingCategoryDataset slidingData;
8.
9.   slidingData = new SlidingCategoryDataset(defaultData,i,i);
10.   categoryPlot.setDataset(i, slidingData);
11.  }
12.  CategoryAxis axis = new CategoryAxis("abc");
13.  categoryPlot.getCategoriesForAxis(axis);
14. }
15.}
```

**Figure 3 Sample Test for defect in Figure 2.**

This test invokes `getCategoriesForAxis` method (line 13) which calls `datasetsMapped`. We create the receiver of type `CategoryPlot` for this method invocation at line 3. The method `setDataset` accepts objects of type `CategoryDataset` as input. Since `categoryDataset` is an interface, required objects are created using appropriate constructors of subclasses. We invoke `setDataset` method multiple times to populate the `datasets` field (line 10). Manually designing this test is a non-trivial task. During execution, the test iterates over `datasets` with numerous invocations of `indexOf`, thereby, exposing the inefficient loop in `datasetsMapped` method. We fixed the bug by merging the loops and verified the fix by executing the generated test, which resulted in a 128× speedup between the new and old versions (when the number of elements in `datasets` is 100K).[1]

## 3. DESIGN

Figure 4 presents the overall architecture of our approach. Our approach takes a Java library as input and outputs the set of the tests that enable the detection of redundant traversals. The generated tests execute the potentially inefficient loops with appropriate parameters. Our approach is broadly composed of three phases – *summary generation*, *method detection* and *test generation*. Initially, we generate a set of random tests using coverage driven test generation tools [14, 32]. These tests and the input library are input to the *summary generation* phase. This phase generates method summaries that are input to the *method detection* phase. This phase outputs set of methods that will execute potentially redundant traversals. We also identify the *populator* methods that can be used to populate the collection/array fields on which the traversal occurs. Finally, *test*

[1]Our bug report available at https://sourceforge.net/p/jfreechart/bugs/1147/ has been acknowledged by developers and our suggested fix is incorporated in the version 1.0.20.

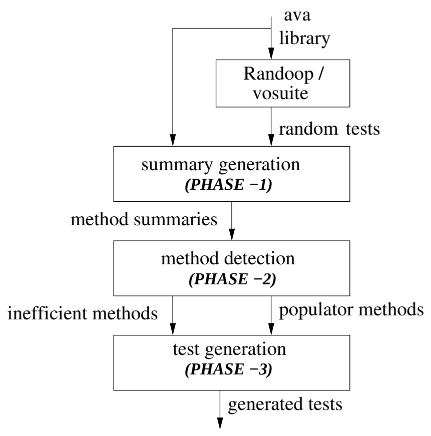

**Figure 4 Architecture diagram.**

*generation* phase generates tests that invoke the methods identified in the previous phase using relevant parameters.

## 3.1 Summary generation

In this phase, we generate method summaries which are used in later phases to identify the methods with potential inefficiencies related to redundant traversal. A method summary contains details about the loop traversals in the method, objects on which the loops are traversed, and the set of other methods invoked. We execute the randomly generated tests and analyze the execution trace to derive this information. We employ dynamic analysis here to generate precise summaries and depend on the test generation tools [14, 32] to provide an initial set of tests with good coverage.

For this purpose, we will need to track the various method invocations and the different loops that are traversed. We identify each loop in the method using a sequence of symbols. The sequence length represents the nesting depth of the loop in the method. The symbols provide the connection between the object on which the loop is traversed and the parameters that are input to the method containing the loop. We also track the method invocations along with parameters and use symbols to connect the parameters in the invoking (current) method and the parameters in the invoked method. More specifically, we use the following data structures to encode this information.

$$\begin{aligned}&loop : < S_1, S_2, \ldots, S_k >\\ &methodsInvoked : method \rightarrow params\\ &summary : loop \rightarrow methodsInvoked\\ &methodSummaries : method \rightarrow summary\end{aligned}$$

Each *loop* is distinctly identified by a sequence of symbols. The length of the sequence represents the nesting depth. The symbols are based on the parameters input to the method containing the loop. If the object on which loop iteration happens at some nesting level is local (and not reachable via the parameter passed to the method), we use $\perp$ to represent the symbol at that nesting level in *loop*. *methodsInvoked* is a map from *method* to *params* which is the list of symbols corresponding to the objects passed to *method*. The receiver of the method is encoded as the first parameter to the method invocation. If the parameter passed to the invoked method is created locally, then we represent it by $\perp$. This is because the client of the invoking method cannot influence the behavior relevant to that parameter in the invoked method. *summary* is a map from *loop* to *methodsInvoked*. This binding enables us to maintain the association between the loops and the methods invoked within the loop context. We represent the methods invoked outside a loop by mapping $\perp$ (representing absence of loops) to *methodsInvoked*. *methodSummaries* integrates *summary* of all the methods invoked from the tests.

```
1  public void foo(A a, B b, C c) {
2    HashSet set = new HashSet();
3    for (E e : a) {  // a is a collection of elements of type E
4      set.add(e);
5      for(E f: b) { c.remove(f); } // b -- collection of type E
6      a.baz(c);
7    }
8    a.update(10);
9  }
```

**Figure 5 Example for method summary generation**

More elaborately, the *methodsInvoked* structure for the simple example given in Figure 5 is:

$[(\texttt{add}_4 \rightarrow < \perp, S_a >), (\texttt{remove}_5 \rightarrow < S_c, S_b >),$
$(\texttt{baz}_6 \rightarrow < S_a, S_c >), (\texttt{update}_8 \rightarrow < S_a, \perp >)]$

Here, the element $(\texttt{add}_4 \rightarrow < \perp, S_a >)$ means that method `add` is invoked at line 4, the first parameter (receiver) is local and is therefore represented by a $\perp$, and the second parameter is given by symbol $S_a$, where $S_a$ is associated with the symbol passed as the parameter to `foo`. The other elements in the map can be constructed accordingly. The distinct loops in `foo` are $<S_a>$, $<S_a,S_b>$ representing the loops from lines 3–7 and 5–5 respectively. The summary of method `foo` is shown in Table 1.

**Table 1** ***summary*** **for method** `foo`

| Loop | methodsInvoked |
|---|---|
| $< S_a >$ | $[(\texttt{add}_4 \rightarrow < \perp, S_a >), (\texttt{baz}_6 \rightarrow < S_a, S_c >)],$ |
| $< S_a, S_b >$ | $[(\texttt{remove}_5 \rightarrow < S_c, S_b >)],$ |
| $< \perp >$ | $[(\texttt{update}_8 \rightarrow < S_a, \perp >)]$ |

Algorithm 1 presents the procedure for generating the method summaries for the input library. It takes a random set of tests as input and outputs the *methodSummaries* object $\alpha$ containing the summary of each method. In this procedure, we execute each test from the set of random tests (line 2). During the execution of these tests, we monitor three types of instructions – (a) loop start, (b) method invocations, and (c) loop finish.

**Algorithm 1** GenMethodSummary

```
Input: A random set of tests $T$
Output: methodSummaries $\alpha$
 1: for each test $t$ in $T$ do
 2:     execute($t$);
 3:     while (($I \leftarrow$ nextInstruction($t$)) $\notin \{exception, halt\}$) do
 4:         switch $I$ do
 5:             case (loop start) :
 6:                 $o \leftarrow$ getLoopTarget();
 7:                 if (isLocal($o$)) then Append $\perp$ to $loop$;
 8:                 else Append symbol($o$) to $loop$;
 9:             case (loop finish):
10:                 Remove last symbol from $loop$
11:             case (invocation of method $x$) :
12:                 $m \leftarrow$ getCurrentMethod()
13:                 $MI \leftarrow$ getMethodsInvoked($\alpha[m]$, $loop$)
14:                 $MI[x] \leftarrow$ getSymbols($x$);
15:                 $\alpha[m][loop] \leftarrow MI$;
16: return $\alpha$
```

If the current instruction is a loop start, we use the auxiliary function `getLoopTarget` to obtain the object $o$ on which the loop traversal happens (line 6). If this object is created locally, modifying the object directly is not feasible from the client invoking the method. Therefore, we append $\perp$ to *loop* (line 7). On the other hand, if $o$ is not local, then we add the associated symbol (line 8). On a loop finish, we simply remove the last element from *loop* to reflect the nesting level appropriately. When a method $x$ is invoked, we get the *methodsInvoked* (in $MI$) in the corresponding *loop* from the method summary of $m$, where $m$ contains the invocation of $x$ (line 13). We update the method summaries to appropriately represent the invocation of $x$ (lines 14-15).

Information in the *loop* is intraprocedural. For ease of presentation, we do not show the stack related operations needed to maintain the intraprocedural data. For example, if $x$ is invoked from $m$ at a nesting depth of two, the invocations of other methods in $x$ (assuming the absence of loops in $x$) is considered to happen outside the context of a loop. We merge the intra-procedural information in the method detection phase (see Section 3.2). At the end of the procedure in Algorithm 1, we return the updated *methodSummaries* object, $\alpha$, containing the *summary* of each method.

## 3.2 Method Detection

In this section, we elaborately discuss the details of two components in *method detection*. The first component identifies inefficiently implemented methods that execute nested loops. The identified methods are invoked with appropriate parameters during test generation. The second component derives *populator* methods, which adds elements to the collections or arrays and can be used to setup the context. For instance, these methods are used to populate collections with large number of elements, in specific patterns, to ensure that repetitive memory accesses are performed across iterations. Executing the populator methods is essential before invoking the inefficient methods to expose the problem.

### 3.2.1 Inefficient methods

With the help of summaries derived for each method in the previous phase, we determine if the method can potentially execute nested loops. For this purpose, we traverse the callgraph in a topologically reverse order and *merge* summaries at each node. Finally, we have a mapping from methods to the list of possible nested loops. For any method $m$ in this map, when it is invoked with appropriate objects corresponding to the symbols present in the nested loops, the execution will entail nested loop iterations. Exposing the magnitude of the inefficiency will be handled by the populator methods which is discussed in the next subsection.

The procedure to identify the inefficient methods is presented in Algorithm 2. Algorithm 2 accepts the method summaries $\alpha$ and a class $\theta$ as input and outputs the *candidates* map, which contains the map from methods to list of potential nested loops.

Initially, Algorithm 2 builds the callgraph for the input class $\theta$ (line 1), where each public method in the class is considered an entry point. $V$ is the list of methods obtained after performing a reverse topological sorting on $G$ (line 2). We break cycles in the call graph randomly. For each method in $V$, if a method is the leaf node in the callgraph, then other method invocations are not feasible. Hence, the only possible loops are the loops that exist in $m$. Therefore, we extract all loops (using `getLoops`) from $\alpha[m]$ and add it to *candidates*$[m]$ (line 5).

For the non-leaf methods, we consider each callee $n$ of method $m$. This is to explore all possible nesting behavior. Therefore, we recursively *merge* the summary of methods $m$ and $n$ using the $\otimes$ operator. This operator accepts a *loop* corresponding to callee $n$ in $\alpha[m]$ (line 8) and all possible loops present in *candidates*$[n]$ and generates a list of merged loops (line 9). We define the $\otimes$ operator using patterns:

$$\langle S_1, ..., S_i\rangle \otimes \langle\perp\rangle = \langle S_1, ..., S_i\rangle \quad (1)$$

$$\langle\perp\rangle \otimes \langle S_1, ..., S_i\rangle = \langle S'_1, ..., S'_i\rangle \quad (2)$$

$$\langle S_1, ..., S_i\rangle \otimes \langle S_l, S_m\rangle = \langle S_1, ..., S_i, S'_l, S'_m\rangle \quad (3)$$

$$\begin{aligned}\langle S_1, ..., S_i\rangle \otimes \langle\langle S_k\rangle, \langle S_l, S_m\rangle\rangle = \langle&\langle S_1, ..., S_i\rangle \otimes \langle S_k\rangle,\\ &\langle S_1, ..., S_i\rangle \otimes \langle S_l, S_m\rangle\rangle\end{aligned} \quad (4)$$

Equation 1 and 2 imply that the merge operation on $\perp$ is identity, albeit with modifications to the symbols in equation 2 (represented by a$'$) to localize the symbols in the context of the caller. Equation 3 represents the merging of two nested loop sequences $\langle S_1, \ldots, S_i\rangle$ and $\langle S_l, S_m\rangle$ respectively to output a single nested loop sequence. Equation 4 presents a scenario where merging is done with two loops in the callee ($\langle S_k\rangle$ and $\langle S_l, S_m\rangle$). It corresponds to independently merging the two loops as shown in the equation.

**Algorithm 2** Identifying inefficient methods

**Input:** method summaries $\alpha$, Class $\theta$
**Output:** *candidates* : *method* $\rightarrow \overline{loop}$

```
1: Graph $G \leftarrow$ getCallgraph($\theta$)
2: $V \leftarrow$ reverseTopologicalSort($G$);
3: for (method $m$ in $V$) do
4:     if ($m$ is leafNode in $G$) then
5:         $candidates[m] \leftarrow$ getLoops($\alpha[m]$);
6:         goto label 3
7:     for (callee $n$ of $m$) do
8:         $loop \leftarrow$ getLoop($\alpha[m]$,n);
9:         $candidates[m] \leftarrow loop \otimes candidates[n]$
10: remove all the private methods from $candidates$
11: remove any loop from $candidates$ with length $< 2$
12: return $candidates$;
```

After updating *candidates* for each method in Algorithm 2 (lines 1-9), we remove all the private methods from *candidates*. This is because we can not invoke private methods directly. We also remove any *loop* with length less than two because these cannot expose any non-linear inefficiencies. Finally, the *candidates* map contains a list of methods and the possible nested loops within these methods. These methods can be invoked with the appropriate parameters to expose potential inefficiencies.

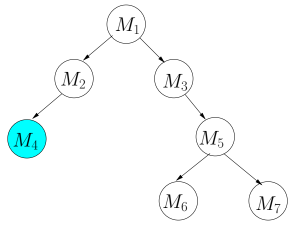

**Figure 6 Example for deriving inefficient methods**

We now illustrate the working of Algorithm 2 using the callgraph given in Figure 6. The graph has seven methods named $M_1$,...,$M_7$. The shaded node ($M_4$) represents a private method. We assume an implementation where the method summaries (including loops, methods invoked, their parameters) are as shown in Table 2. For ease of presentation, we use the notation $S_{ij}$, which represents the symbol corresponding to parameter $j$ of method $M_i$.

**Table 2 Summaries for example in Figure 6**

| method $m$ | *loop* | *method* | *params* | candidates[$m$] |
|---|---|---|---|---|
| $M_4$ | $\langle S_{41}, S_{42}\rangle$ | $\perp$ | - | $\langle S_{41}, S_{42}\rangle$ |
| $M_2$ | $\perp$ | $M_4$ | $\langle S_{23}, S_{22}\rangle$ | $\langle S_{23}, S_{22}\rangle$ |
| $M_6$ | - | - | - | - |
| $M_7$ | $\langle S_{71}\rangle$ | $\perp$ | - | $\langle S_{71}\rangle$ |
| $M_5$ | $\perp$ | $M_6$ | - | - |
| | | $M_7$ | $\langle S_{51}\rangle$ | $\langle S_{51}\rangle$ |
| $M_3$ | $\langle S_{32}\rangle$ | $M_5$ | $\langle S_{31}\rangle$ | $\langle S_{32}, S_{31}\rangle$ |
| $M_1$ | $\perp$ | $M_2$ | $\langle \perp, \perp, S_{11}\rangle$ | $\langle S_{11}, \perp\rangle$ |
| | | $M_3$ | $\langle \perp, S_{11}\rangle$ | $\langle S_{11}, \perp\rangle$ |

Algorithm 2 traverses this graph in a reverse topological order and starts from method $M_4$. According to the summary in Table 2, $M_4$ does not invoke any method. Hence, we add the existing *loop* to the *candidates* map. Subsequently, we consider $M_2$ which invokes $M_4$ with *params* $\langle S_{23}, S_{22}\rangle$ outside a loop (based on a $\perp$ for *loop*). We merge $\langle\perp\rangle$ and candidate[$M_4$] given by $\langle S_{41}, S_{42}\rangle$ to obtain $\langle S_{23}, S_{22}\rangle$. This is because the loop in $M_4$ corresponds to the first two parameters of $M_4$ and our summary connects them to the third and second parameters of $M_2$ respectively. Similarly, while calculating the candidate[$M_1$], we observe that $M_2$ is invoked outside a loop. The candidate[$M_2$] specifies the presence of a nested loop associated with traversing $\langle S_{23}, S_{22}\rangle$. Since, this corresponds to $\langle S_{11}, \perp\rangle$ based on the parameter summary for $M_2$ when invoked in $M_1$, we obtain the relevant candidate[$M_1$]. The remaining candidates shown in the table are obtained in a similar manner.

Finally, we can remove $M_4$ which is a private method and any *loop* with depth $< 2$ (e.g., in $M_7$, $M_1$ and $M_5$). The final candidate mappings are given below:

$$[M_2 \rightarrow \langle S_{23}, S_{22}\rangle,\ M_3 \rightarrow \langle S_{32}, S_{31}\rangle]$$

If we invoke $M_2$ and $M_3$ with the appropriate parameters, we can potentially expose the inefficiencies due to the implementation of $M_4$ (a private method) and the nesting that transcends method boundaries ($M_3$ and $M_7$).

### *3.2.2 Populator methods*

In this section, we discuss the procedure to derive populator methods. Given an inefficient method that needs to be invoked along with symbolic information on parameters, we need a mechanism to populate the collections corresponding to those symbols to enable traversal of the loops.

In order to derive the methods that can help populate the collection objects, we monitor the execution of methods that operate on the objects of that type. If there is an increase in the total count of the elements in the collection object after an invocation of a method, the method is identified as a possible populator method. Since there can be multiple methods which can affect the overall count, we rank the identified methods based on the behavior of the method in different contexts and select the highest ranking method.

More elaborately, we consider each test from the suite of random tests. For any method $m$ invoked from a test, we statically extract the set of class fields (of collection types), say $W$, updated in the method. We execute the test and count the number of elements present in each field $w \in W$, just before invoking the method $m$. After method exit, we recount the number of elements in the field $w$ for presence of additional elements. If additional elements are present, we consider the method as a possible populator. After repeating this procedure for each test from the set of random tests, we rank the overall populator methods for any given type and select the highest ranking method. Also, we do not consider constructors as possible populator methods because they cannot be repeatedly invoked to increase the size and will deliver independent objects.

```
Class B                                   Test
----------------------------              -------------------------
1.public class B {                        1.B b = new B();
2. HashSet set;                           2.b.initialize();
3. public B(){ set = new HashSet();}      3.b.add(new A());
4. public void add(A a) {                 4.b.add(new A());
5.  set.add(a);}
6. public void initialize() {
7.  set = new HashSet(new A()); }
8. }
```

**Figure 7 Example class $B$ and corresponding test**

We use the example in Figure 7 to illustrate our approach. For the class `B` under consideration, the constructor initializes the field `set`. Method `add` adds input object `a` to `set` (line 5) and method `initialize` assigns a new `HashSet` with one element (line 7). While monitoring the methods invoked from the test (shown on the RHS of the figure), we ignore the invocation of the constructor at line 1. Before invoking `initialize` (at line 2), we count the number of elements in `set` and obtain 0. After executing `initialize`, we observe an increase in the number of elements by one and is considered a possible populator. Similarly, we observe addition of elements by `add` and include it as a populator. Since `add` increases the overall count under different contexts, we rank it higher and select it as a populator method for `set` in `B`.

## 3.3 Test generation

In this section, we discuss our approach to generate tests that execute nested loops in the library with the appropriate objects. Algorithm 3 takes the methods identified as inefficient and populator, from the previous phase, as input. We represent the set of methods using $I$ and $P$ respectively. The goal then is to generate tests that invoke methods in $I$ and ensure that the objects corresponding to the parameters of the method have large number of elements and cover the code region that contain the inefficiencies.

**Algorithm 3** GENPERFTESTS

**Input:** Class ($\theta$), Inefficient methods ($I$), Populator methods ($P$)

```
 1: for (each $m$ in $I$) do
 2:     for (each $loop$ in $I[m]$) do
 3:         $receiver$ = CREATEOBJECT($\theta$)
 4:         for (each parameter $i$ in $m$) do
 5:             if ($i$ is in $loop$) then
 6:                 $type \leftarrow$ getType($i$)
 7:                 $\mu_i \leftarrow$ CREATEOBJECT(type)
 8:                 $m_p \leftarrow P[type]$
 9:                 for (j = 0; j < COUNT; j++) do
10:                     for (each parameter $p$ to $m_p$) do
11:                         $c_j$ = CREATEOBJECT(getType($p$))
12:                     Invoke $m_p$ on $\mu_i$ with params $(c_1, c_2, ..., c_n)$
13:             else
14:                 $\mu_i \leftarrow$ Reuse parameter from test invoking $m$
15:         Invoke $m$ with $receiver$ on params $(\mu_1, \mu_2, ..., \mu_n)$
```

Algorithm 3 traverses all methods in $I$ map (line 1). For each method $m$, we iterate over each problematic nesting behavior (line 2). We instantiate the method $m$ using CREateobject, an auxiliary function, which takes the type of the object as input.

We perform a simple static analysis to enable instantiation of the object. If the class is public, we extract the set of public constructors. If the class has only private constructors, we derive the appropriate callsites within the class and use them for instantiation. If the class is private, abstract or an interface, we traverse the *class hierarchy* graph [25] and check all the public subclasses and identify public constructors. If the constructor requires further objects, we recursively perform a similar procedure. After collecting all the required constructors, we instantiate the objects appropriately to obtain the receiver for invoking the method $m$.

After creating the receiver object, we need to create parameters to invoke the method $m$. Moreover, we need to ensure that parameter objects on which loop traversals happen in the implementation of $m$ are suitably populated. Therefore, if the parameter corresponds to an object on which a loop traversal can happen, it will be present in the sequence of symbols represented by the loop (line 5). Otherwise, we simply reuse the parameter from the original random test invoking $m$ (line 14). This is because if the inefficient region under consideration is dependent on the value of the parameter, there will be minimal changes and the possibility of the generated test reaching the inefficient regions is higher.

If the parameter is part of the loop sequence, then we create the object of the appropriate type after obtaining its type (lines 6-7). Further, we populate the constructed object (lines 8-12). We obtain the populator method $m_p$ based on the type of the object (line 8). Subsequently, we invoke the method $m_p$ based on a parameterized number of times ( `COUNT`). To obtain parameter objects for the invocation, we employ createobject (line 11) to instantiate the objects.

We now illustrate the working of Algorithm 3 using an example. Consider the input $I$ and $P$ as follows:

$I$: [(A.foo<B $b_1$, B $b_2$, bool flag> $\rightarrow$ <$S_1$, $S_2$>)]
$P$: [B, update<C>]

Based on the data in $I$, we need to generate a test that invokes `foo` with appropriate objects for first and second parameters to `foo`. The populator method for object of type `B` is `update` which takes objects of type `C`. Figure 8 presents the test generated. It invokes method $foo$ that will help expose the redundant traversal defect. For the first two parameter objects on which the loop traversal happens, the objects are created (at line 4) and populated subsequently using the information present in $P$. Because the third parameter to `foo` need not be modified, we simply obtain the value used in the concrete run using `getRuntime` method.

## 4. IMPLEMENTATION

We have implemented the algorithms discussed earlier as part of the soot bytecode analysis framework [41]. Our implementation generates tests for Java libraries. We now discuss the tradeoffs considered to make our implementation practical.

### 4.1 Populating the collections with patterns

In Algorithm 3, we discussed the procedure to instantiate the objects that can be used as parameters for the various method invocations. We now discuss how the parameter objects can be populated. While a straight-forward approach

```
1. public class Test {
2.  public static void main(String[] args) {
3.    A a = new A();
4.    B b1 = new B(); B b2 = new B();
5.    for(int i=0; i < args[0]; i++){
6.      C c = new C(random(Int));
7.      b1.update(c);
8.    }
9.    for(int i=0; i < args[0]; i++){
10.     C c = new C(random(Int));
11.     b2.update(c);
12.   }
13.   a.foo(b1, b2, getRuntime(flag));
14. }
15.}
```

**Figure 8 Generated test**

to populate collection objects is to provide *random* objects, it may not always be useful in exposing potential problems. For instance, if there is a search of an element in a traversal and the element being searched and the first element in the traversal coincidentally match, then the underlying problem may not be exposed. This is also necessitated as our analysis is light-weight and does not track path constraints. Therefore, to broaden the possibility of identifying the inefficiencies, we use the patterns based on size, similarity among the elements across collections and the type of elements.

1. **Size**: To handle multiple scenarios where the implementation makes choices dependent on the size of the collection objects on which the traversal happens, we use two variants for all pairs of nested loops – (a) collection sizes are equal, and (b) size of first collection is less than the second and vice versa.
2. **Similarity**: A few iterations are dependent on the similarity of elements across the collection objects under consideration. Therefore, we use three variants to populate the collections with elements – (a) all elements are distinct, (b) all elements are the same, and (c) first collection is a subset of the second collection and vice versa.
3. **Type of elements**: If the collection has a specific type, and is extended by multiple other types, we ensure that the original collection can take all possible types. This is to ensure that any path condition that is dependent on the type of the collection is satisfied, thereby, exploring the code covered by the condition.

Because of the three different patterns, the number of possible combinations among them (e.g., same size – different elements – same type, same size – same elements – same type, etc.) can be significant. Our current prototype handles a limited combination of these patterns.

### 4.2 Handling multiple method summaries

The input random tests can invoke the same method multiple times under different contexts. Therefore, we need to choose a summary among the possible summaries in the process of generating candidate methods. We prioritize the summaries based on the nesting depth of the loop, methods invoked from the loop, and number of methods invoked.

More specifically, for two summaries $\alpha_1$ and $\alpha_2$, we prioritize them as follows. If $\alpha_1$ has nesting loop depth greater than that of $\alpha_2$, we use the former. If both have the same nesting loop depth, then we prioritize the summary that contains more method invocations within a loop context. If this

**Table 3 Benchmarks**

| Benchmark | ID | Version | KLoC | #Classes analysed | #Methods | #Static nested loops | #Randomly generated tests | Analysis time (S) |
|---|---|---|---|---|---|---|---|---|
| Apache collection | B1 | 4.4.1 | 117 | 10 | 364 | 45 | 621 | 753 |
| PDFBox | B2 | 1.8.10 | 219 | 2 | 80 | 8 | 124 | 275 |
| Groovy | B3 | 2.4.6 | 197 | 2 | 69 | 10 | 108 | 59 |
| Guava | B4 | 18 | 2517 | 4 | 585 | 21 | 89 | 241 |
| JFreeChart | B5 | 1.0.19 | 233 | 2 | 280 | 42 | 62 | 214 |
| Ant | B6 | 1.8.4 | 187 | 3 | 73 | 9 | 105 | 141 |
| Lucene | B7 | 5.2.1 | 320 | 2 | 168 | 42 | 60 | 85 |

value is also the same, we choose the summary that contains more method invocations (outside loops).

## 4.3 Parameterizing random test generation

Our approach accepts a randomly generated set of tests as input. We use EvoSuite [14] and Randoop [32] for this purpose. A key goal is to invoke more number of methods so that inter-procedural summary information can be updated suitably. Therefore, we guide the random test generation tools to invoke more methods by setting the relevant parameters in these tools.

## 4.4 Optimizing virtual call resolution

We statically analyze types of all the fields in the class and track the type bindings performed in each constructor. This is to optimize the virtual call resolution and reduce the overall possible set of methods that can be invoked.

```
1. public class A {
2.  public B order;
3.  public A() { order = new C(); }
4.  public void foo(){
5.    ...
6.    order.baz();
7.    ...
8. }
```

We illustrate this using a simple example as shown above. `A` contains a field `order` of type `B` (line 2). The constructor (at line 3) restricts the type to field `C` (under the assumption that `B` is a superclass of `C`). In the invocation of `baz` from `foo`, we use this information to restrict the possible types on which `baz` is invoked.

## 5. EXPERIMENTAL EVALUATION

In this section, we report the evaluation of our implementation and demonstrate the effectiveness of our approach. We have applied our approach to many popular Java libraries. Our selection of benchmarks was guided by earlier bug reports on these benchmarks [30, 31]. We performed our experiments on an `Ubuntu-14.04` desktop machine with a 3.5Ghz Intel Core i7 processor with 16GB of RAM.

Table 3 provides the details of the benchmarks used for our experiments. `Apache collection` provides many powerful data structures that are used to build Java applications; `PDFBox` is a Java tool for working with pdf documents; `Groovy` is a optionally typed dynamic language that has static compilation capabilities; `Guava` is the Google Core Libraries for Java; `JFreeChart` is a Java chart library to display professional quality charts; `Ant` is a Java library and command-line tool to help build software; and `Lucene` is a high-performance, full-featured text search engine library. For brevity, as indicated in the table, we refer to benchmarks as `B1` through `B7`.

The table presents the version of the different benchmarks used for our experiments. The lines of code varies from 117Kl for `Apache collection` to 2.5MC for `Guava` libraries. We select the classes in these benchmarks on which bugs are reported in other papers [30, 31]. The cumulative number of methods in all these classes varies from 69 for `Groovy` to 585 for `Guava`. We also count the static nested loops present in the analyzed code which ranges from 8 to 45. The total method and loop count indicate, without even considering the parameters to method invocations and the complexities due to virtual calls, that analyzing even few classes manually is a nontrivial task.

We use EvoSuite [14] and Randoop [32] to generate the initial tests of random tests that is used as input to our implementation. We restrict the number of tests generated by these tools to 200 for each class under consideration. The overall time to generate the random tests ranges from a minute to 12 minutes. Essentially, these tests invoke the various methods in the class with random objects. We designed our experiments to answer the following research questions:

1. *RQ1*: Is our implementation effective in generating bug-revealing tests?
2. *RQ2*: Is our approach useful for practical adoption?
3. *RQ3*: Are randomly generated tests sufficient to expose the redundant traversal problem?
4. *RQ4*: How many elements in the collection object will be necessary to expose the underlying performance issue?

## 5.1 RQ1: Effectiveness of test generation

**Table 4 Information on generated tests, detected bugs and analysis time.**

| ID | #Generated tests | #Bugs | #New bugs | #False positives | Analysis time (Min) |
|---|---|---|---|---|---|
| B1 | 80 | 16 | 9 | 1 | 45 |
| B2 | 30 | 6 | 6 | 0 | 12 |
| B3 | 20 | 5 | 4 | 0 | 7 |
| B4 | 50 | 9 | 10 | 1 | 28 |
| B5 | 15 | 3 | 1 | 4 | 24 |
| B6 | 24 | 6 | 3 | 1 | 15 |
| B7 | 5 | 1 | 1 | 0 | 16 |
| Total | 224 | 46 | 34 | 8 | 147 |

Table 4 presents the information on the tests generated using our approach when the initial test suite consists of tests from various test generators [32, 14]. The number of generated tests varies from 1 for `B7` to 16 for `B1`. This is significant reduction from the total number of randomly generated tests that is input to our implementation. Ideally, these tests will be input to Toddler [30] to detect bugs. However, since the implementation of Toddler is unavailable[2], we manually analyzed the generated tests. This helped us reveal 46 bugs in these benchmarks, including 34 previously unknown performance issues .

The number of bugs detected also depends upon the generated tests. We observe that 36 bugs are detected when

[2]Personal communication – Adrian Nistor

the set of patterns proposed in the Section 4.1 are disabled. Other bugs are missed because the test with default pattern either generates objects of inappropriate types or do not meet the conditions to execute the loop.

Our approach is able to generate useful tests by analyzing random tests. The generated tests detect performance problems even in well-tested open-source Java libraries.

## 5.2 RQ2: Practicality of our approach

Table 4 also gives information on the practicality of our approach. There are two key issues involved w.r.t practicality of program analysis tools – false positives and analysis time [5]. A few tests generated by our approach do not reveal any defects. Our approach generates eight tests that do not reveal any defect.[3] Based on industry standards [24], the false positive rate of less than 5% is negligible.

On closer examination, the false positive tests are mainly due to two reasons – generated tests take a different path compared to the original test, and absence of redundancy during nested loop traversal. For example, the implementation of the library in `Apache-collection` compares the size of objects present in the two input collections and follow a path based on the result. Since, our generated test did not use this constraint (recall that our current prototype explores a limited combination of patterns for parameters), we generate a test where a different path is taken. In a few other cases (e.g., `JFreeChart`, `ant`), there is no redundancy in the traversal (e.g., 2-dimensional table, checking for duplicate elements in list). This set of false-positive can be eliminated when their executions are analyzed using TODDLER.

The overall time taken to generate the tests for all the benchmarks is around 2.5 hours. On average, this corresponds to 6 minutes per analyzed class. This time depends on the number of tests considered in the beginning and also the length of method sequences in those tests.

More than 95% of the tests generated by our approach help in bug detection and the time taken is less than five minutes per class. These numbers indicate the potential for seamless integration of our implementation in the software development process.

## 5.3 RQ3: Comparison with randomly generated tests

We now discuss the usefulness of the tests generated by our approach as compared to the random tests. The first author created new versions of the libraries by fixing the bugs appropriately by removing redundant traversals. The generated testsuite (with 10K elements populated in the collection objects) and the (input) random testsuite were executed on the original and fixed versions of the libraries. Figure 9 presents the percentage performance improvement of the fixed version over the original version for the two testsuites across all the benchmarks. The time taken to execute the original version is shown on top of the bar (e.g., 21.11 seconds to execute randomly generated tests on B1).

The figure clearly demonstrates the huge performance gains observed on the generated tests as compared to the random tests. This is because the generated tests are directed towards exposing the problems that are addressed by

[3]Many defects are detected by multiple generated tests.

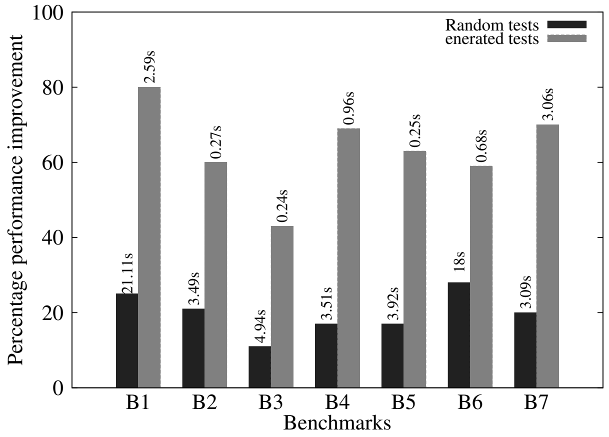

**Figure 9 Comparison with randomly generated tests.**

our fixes. Moreover, the time taken to execute the generated tests on the original version is significantly less than that of the random tests across all benchmarks. This suggests that the generated tests can be used as part of a regression testsuite. Moreover, the magnitude of performance gains implies that the use of existing dynamic tools like TODDLER will be more successful with the generated tests.

The generated tests take less time to execute and are more suitable to expose the magnitude of the underlying performance problem.

## 5.4 RQ4: Size of the collection objects

In order to use our implementation for practical purposes, we wanted to find the number of elements in the collections to help expose the inefficiencies with the loop. Therefore, we consider the original and fixed versions of the benchmark. Further, we modified the number of elements that are populated in the collection (based on the `COUNT` parameter) from 100 to 100K. Then we execute the generated tests on the original and fixed versions and compare the percentage performance improvement in time between the two versions. Figure 10 presents the corresponding results.

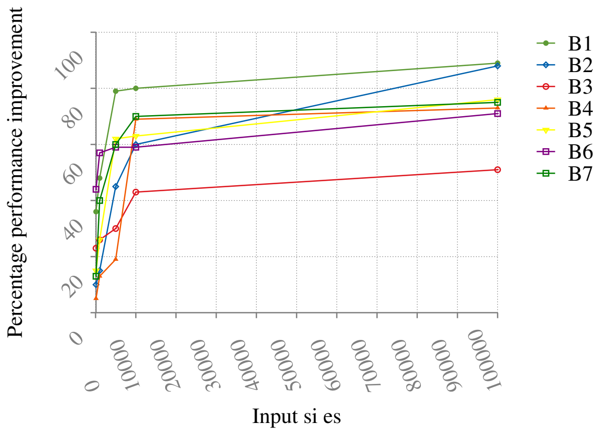

**Figure 10 Percentage performance improvement.**

According to the figure, the performance improvement saturates after 10K elements. In other words, populating the collection objects with 10K elements will yield sufficient repetition of memory accesses leading to the detection of the underlying problem. Populating the collection with fewer elements (e.g., less than 1K elements) may not always demonstrate a substantial difference.

Populating the collection objects with 10K elements will enable detection of performance issues.

## 5.5 Threats to validity

Our approach is sensitive to the initial set of random tests used. If these tests do not cover problematic code regions, our approach will not be able to generate the necessary tests, a drawback that is shared with other dynamic analyses. Our ability to identify populator methods can also be hampered due to this reason. Also, as we do not explicitly track the path conditions, it is possible that the generated tests may cover a different path, which can reduce the effectiveness of our approach. Although, benchmarks used in our experimental results are representative of most codebases, it is possible to have custom codebases where our analysis may be less effective.

## 6. RELATED WORK

*Detection of redundant traversal bugs.*

Elegant techniques to detect redundant traversal bugs have been designed recently [30, 31]. TODDLER [30] is a dynamic analysis technique that analyzes execution of input tests to detect repetitive memory accesses. The effectiveness of the technique is dependent on the ability of the input tests to expose the problem. Manually writing these tests is nontrivial and random test generation is not effective. Our approach is complementary to TODDLER, as we generate the tests that can be used by it. CLARITY [31] analyzes the source code of the library to detect redundant nested loop traversals. However, as we discuss earlier, it is difficult to automatically confirm the bug, validate any fix and use the information to avoid future regressions. In contrast, our approach generates tests to serve these purposes.

*Detection of performance bugs.*

Many useful techniques to detect a variety of performance bugs have been designed [27, 21, 10, 43, 12]. Dynamic analyses have been proposed to detect problems pertaining to object bloat in Java programs [38, 44, 45, 46, 26]. Mudduluru *et al* [26] propose an approach to efficiently instrument object flow profiles and use them to detect object bloats. RESURRECTOR [43] uses dynamic analysis to detect problematic code regions, which can be fixed to reduce GC pressure. Bhattacharya *et al* [6] propose a hybrid approach to detect object bloat due to unnecessary creation of temporary container and string objects. Techniques that propose static identification of locations where dead objects can be reclaimed are also proposed [17]. Other proposed techniques include identifying wasteful use of temporary objects [38], incorrect use of data structures [37], latent performance bugs [21] and performance issues in clouds and smartphones [10]. EventBreak [33] uses a random testsuite and identifies event handlers whose execution time may gradually increase while using applications. This approach generates more tests based on its execution. Our approach differs by doing white-box analysis and by focusing on unit-level tests. All these techniques are geared towards detecting other kinds of performance problems. Our approach generates tests to expose redundant traversal bugs.

*Automated test generation.*

Automatic test-generation tools [32, 14] for Java utilize feedback from generated tests. We use the tests generated by these random test generators as input to bootstrap the process of generating tests necessary for exposing loop inefficiencies. SEEKER [40] combines static and dynamic analyses to synthesize method sequences that are necessary for high coverage testing. Concolic testing that integrates symbolic execution with concrete trace information enhancing path coverage has been effective in detecting bugs [16, 15, 8, 9]. Our approach is inspired by these techniques and combines concrete traces with symbols to generate the necessary tests. Our approach will benefit from the tests generated by this approach as it increases coverage, thereby, the possibility of exploring the problematic code regions containing the loops.

*Test generation for detecting concurrency bugs.*

Automatically generating multithreaded tests to detect data races [36], deadlocks [34], and atomicity violations [35] have been found to be effective in detecting rare bugs in well-tested and thread-safe Java libraries. These techniques analyze the execution of a random set of sequential tests to generate the required tests. Our current approach is inspired by the successes of the test generators in the context of detecting concurrency bugs. Our approach also operates by analyzing a random set of tests and uses this concrete information to generate relevant tests.

*Profiling.*

Profiling is a common technique to detect performance problems in programs. Ball and Larus [4] propose a numbering scheme to get statistics on the control flow paths traversed in an execution with minimum overhead. Many extensions to this technique have been proposed [42, 11, 22]. In [11], the imprecision of path profiling due to loop iterations is addressed. Improving garbage collection by profiling data due to dynamically created objects is also proposed [7, 20, 23, 3, 18]. Our work is orthogonal to these approaches as we enable detection of unnecessary traversals of loops to improve the performance of libraries.

## 7. CONCLUSIONS

Redundant traversal of loops under nested conditions can affect the overall performance of Java libraries. The usefulness of existing techniques to detect these problems can be significantly enhanced in the presence of a directed test generator that generates tests to help expose these inefficiencies. In this paper, we designed a novel, scalable and automatic dynamic analysis technique that analyzes the execution of randomly generated tests to construct targeted tests, which can serve as input to existing dynamic analyses. The evaluation of our implementation on many Java libraries demonstrate the efficacy of our design. Our tool generated 224 tests that enabled detection of 46 bugs, including 34 previously unknown bugs.

## 8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their useful feedback. We are grateful to Sridhar Gopinath for help in setting up the initial infrastructure for experimental evaluation. The first author is supported by a Google travel grant and the second author is supported by a Google faculty research award.

## 9. ARTIFACT DESCRIPTION

### 9.1 Introduction

The artifact is a 4GB tar.gz file, and is available at http://drona.csa.iisc.ac.in/˜sss/tools/glider. The uncompressed folder consists of a VM image, and a README file which describes the instructions along with snapshots for clear understanding of the artifact. The virtual image provided has the required execution environment set up to run our analysis. Packages like maven, ant and gnuplot are preinstalled for building benchmarks and generating graphs. We have installed both Java7 and Java8 in the virtual machine. This is necessary since random test generators like Evosuite work with Java8 and soot works with Java7.

### 9.2 Hardware dependencies

1) Virtual box with version 5.0.18
https://www.virtualbox.org/wiki/Downloads
2) System with at least 12GB of main memory.
3) System with at least 8 cores (preferably).

### 9.3 Installation

To execute virtual machine :
1) Download and uncompress the artifact (around 10GB).
2) Open VirtualBox.
3) Create a new VM by clicking machine –> new.
4) Give a name to the new VM.
5) Select Linux type and Ubuntu (64bit) version.
6) Click Next, select 4GB of RAM and select Next again.
7) In the hard drive selection screen, select the option to "use an existing hard drive file", then select artifact.vdi file containing the VM image that you just downloaded. Go to settings->system->processor and assign 4 cores to VM. Start the VM with `Username : artifact Password : 123`

### 9.4 Description

The tool is present in the folder named artifact on the desktop which is organized as:

- *versions* : original and modified versions of benchmarks.
- *randoop, evosuite* : contains scripts to generate and parse randomly generated tests.
- *sootAnalysis* : static and dynamic analysis.
- *perfTests* : tests generated to detect loop inefficiencies.
- *expected-perfTests* : expected generated tests to detect loop inefficiencies.
- *figures* : displays generated figures along the dimensions of figure9 and figure10 in the paper.
- *benchmark* : contains the program under analysis.
- *bugs* : a file with detailed description of the bugs detected.

The artifact folder contains a copy of the paper named paper.pdf.

### 9.5 Experimental setup

We describe three steps that might be followed as part of evaluation. We have used the environment variable `$fsehome` in the subsequent discussion which points to the artifact folder on the desktop. The first step shows the working on a simple example. This step describes the summary generated, and how it is leveraged further for synthesizing tests. In the second step, we show a demonstration on a class. The third step runs all the benchmarks together and generates data for Figure 9 and Figure 10 as shown in the paper. [You can skip STEP 1 and 2 to directly run benchmarks].

#### 9.5.1 STEP1 : Analyzing a simple example

We have added examples in $fsehome/sootAnalysis/test folder so that one can get familiar with the tool.
1) To run these examples, `cd $fsehome/sootAnalysis`
2) Execute `./sampleTest.sh` ; outputs the summaries for each method in Type.java and the generated tests. [Run other examples using instructions at the header of sampleTest.sh]

#### 9.5.2 STEP2 : Analyzing a class from benchmarks

Consider the class *CollectionBag* from collections which is placed in the package *org.apache.commons.collections4.bag*
1) Change the working directory to evosuite

```
cd $fsehome/evosuite
```

2) Remove contents of benchmark and perfTests

```
rm -rf $fsehome/benchmark/*
rm -rf $fsehome/perfTests/*
```

3) Copy class files of collections to benchmark

```
cp -r $fsehome/versions/collections/bin/* $fsehome/benchmark/
```

4) ./Puffer.sh -R <packageName> <className> for our analysis
For this example, the command is :

```
./Puffer.sh -R org.apache.commons.collections4.bag CollectionBag
```

This step takes around 10 minutes depending upon the initial number of tests generated and the performance tests are stored in $fsehome/perfTests/ folder. It outputs 5 tests with different patterns for one redundant loop traversal. One of which fails during compilation because of invalid type (Section 4.1). This step may not generate any tests if the randomly generated input tests do not execute inefficient loop.

#### 9.5.3 STEP3

Running all the benchmarks together to generate data for Figure 9 and Figure 10 :
1) Change the working directory to $fsehome. `cd $fsehome`
2) Execute `./getData.sh.`
This script performs the following operations.
i) Deletes pregenerated graphs for figure 9 and 10.
ii) Execute ./run.sh reuseSummary to analyse all the benchmarks using pregerated summaries
iii) Execute ./generateFigure9.sh to generate data for Figure9.
iv) Generate figure9 and figure10 in $fsehome/figures.
This script takes around 1.5 to 2 hours to finish execution. During execution of the scripts, you may observe some test failures. You can safely ignore them as those errors correspond to inconsistency of java versions.

The generated tests are stored in $fsehome/perfTests folder and methods with bugs are listed in the file TestsOutput.log. The generated figures (fig9.eps and fig10.eps) are saved in $fsehome/figures folder. Please note that, the results obtained will be roughly proportional to the performance ratios mentioned in the Figure9 and Figure10 of the paper. Figure 9 shows that the percentage performance improvement achieved using directed tests is more than random tests, i.e., grey bars should rise above black bars. Figure 10 shows that percentage performance improvement increases with size of the input collections.

## References

[1] https://answers.acrobatusers.com/Performance-Issues-Acrobat-Reader-11-0-0-2-secure-mode-enabled-q91247.aspx.

[2] https://bugs.eclipse.org/bugs/show_bug.cgi?id=394078.

[3] O. Agesen and A. Garthwaite. Efficient object sampling via weak references. In *Proceedings of the 2Nd International Symposium on Memory Management*, ISMM 2000.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29. IEEE Computer Society, 1996.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2).

[6] S. Bhattacharya, M. G. Nanda, K. Gopinath, and M. Gupta. Reuse, recycle to de-bloat software. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP 2011.

[7] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinely, and J. E. B. Moss. Pretenuring for java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA 2001.

[8] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI 2008.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS 2006.

[10] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC 2014.

[11] D. C. D'Elia and C. Demetrescu. Ball-larus path profiling across multiple loop iterations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA 2013.

[12] L. Della Toffola, M. Pradel, and T. R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015.

[13] D. D. Dunlop and V. R. Basili. A comparative analysis of functional correctness. *ACM Comput. Surv.*, 14(2), June 1982.

[14] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE 2011.

[15] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2007.

[16] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2005.

[17] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-me: A static analysis for automatic individual object reclamation. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2006.

[18] W. huang, W. Srisa-an, and J. M. Chang. Dynamic pretenuring schemes for generational garbage collection. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS 2004.

[19] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2012.

[20] M. Jump, S. M. Blackburn, and K. S. McKinley. Dynamic object sampling for pretenuring. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM 2004.

[21] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2010.

[22] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI 1999.

[23] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6), June 1983.

[24] S. McPeak, C.-H. Gros, and M. K. Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013.

[25] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[26] R. Mudduluru and M. K. Ramanathan. Efficient flow profiling for detecting performance bugs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2016.

[27] 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*, ESEC/FSE 2013.

[28] A. Nistor, P. C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE 2015.

[29] A. Nistor and L. Ravindranath. Suncat: Helping developers understand and predict performance problems in smartphone applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014.

[30] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE 2013.

[31] O. Olivo, I. Dillig, and C. Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015.

[32] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA 2007.

[33] M. Pradel, P. Schuh, G. Necula, and K. Sen. Eventbreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA 2014.

[34] M. Samak and M. K. Ramanathan. Multithreaded test synthesis for deadlock detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA 2014.

[35] M. Samak and M. K. Ramanathan. Synthesizing tests for detecting atomicity violations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015.

[36] M. Samak, M. K. Ramanathan, and S. Jagannathan. Synthesizing racy tests. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015.

[37] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2009.

[38] A. Shankar, M. Arnold, and R. Bodik. Jolt: Lightweight dynamic analysis and removal of object churn. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA 2008.

[39] L. Song and S. Lu. Statistical debugging for real-world performance problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA 2014.

[40] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA 2011.

[41] R. Vallee-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *In International Conference on Compiler Construction, LNCS 1781*, 2000.

[42] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: Compactly numbering interesting paths. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2007.

[43] G. Xu. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA 2013.

[44] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2009.

[45] G. Xu, M. D. Bond, F. Qin, and A. Rountev. Leakchaser: Helping programmers narrow down causes of memory leaks. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2011.

[46] D. Yan, G. Xu, and A. Rountev. Uncovering performance problems in java applications with reference propagation profiling. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE 2012.