Toddler: Detecting Performance Problems via Similar Memory-Access Patterns

Adrian Nistor[1], Linhai Song[2], Darko Marinov[1], Shan Lu[2]
[1]University of Illinois at Urbana-Champaign, USA, [2]University of Wisconsin–Madison, USA
{nistor1, marinov}@illinois.edu, {songlh, shanlu}@cs.wisc.edu

***Abstract*—Performance bugs are programming errors that create significant performance degradation. While developers often use *automated oracles* for detecting functional bugs, detecting performance bugs usually requires time-consuming, manual analysis of execution profiles. The human effort for performance analysis limits the number of performance tests analyzed and enables performance bugs to easily escape to production. Unfortunately, while profilers can successfully localize slow executing code, profilers cannot be effectively used as automated oracles.**

This paper presents TODDLER, a novel automated oracle for performance bugs, which enables testing for performance bugs to use the well established and automated process of testing for functional bugs. TODDLER reports code loops whose computation has repetitive and partially similar memory-access patterns across loop iterations. Such repetitive work is likely unnecessary and can be done faster. We implement TODDLER for Java and evaluate it on 9 popular Java codebases. Our experiments with 11 previously known, real-world performance bugs show that TODDLER finds these bugs with a higher accuracy than the standard Java profiler. Using TODDLER, we also found *42 new bugs* in six Java projects: Ant, Google Core Libraries, JUnit, Apache Collections, JDK, and JFreeChart. Based on our bug reports, developers so far fixed 10 bugs and confirmed 6 more as real bugs.

I. INTRODUCTION

Software performance is critical for how end-users perceive the quality of the deployed software. Performance bugs[1] are programming errors that create significant performance degradation [1]. Even when software is mature and written by expert programmers, performance bugs have been known to cause serious and highly publicized incidents [2]–[5]. The state-of-the-art techniques for detecting and testing for performance bugs are still preliminary. As a result, performance bugs easily escape to production runs, hurt user experience, degrade system throughput, and waste computational resources [6], [7]. Because performance bugs are difficult to find, they affect even well tested software such as Windows 7's Windows Explorer, which had several high-impact performance bugs that escaped detection for long periods of time, despite their severe effects on user experience [8].

A key reason why performance bugs escape so easily to production is that testing for performance bugs cannot use the well established process of testing for functional bugs with *automated oracles*. An automated oracle detects if a test triggers a (functional or performance) bug, in which case the developer needs to inspect the test. To test for functional bugs, developers usually follow three steps: (1) write as many and as diverse tests as allowed by the testing budget, (2) run these tests and use automated oracles (e.g., crashes or assertions) to find which tests fail, and (3) inspect *only* the failing tests. To test for performance bugs, developers typically write a small number of tests, use a profiler to localize code regions that take a lot of time to execute, and then reason whether these regions can be optimized and if the effort spent for optimizing (time, added code complexity) is worth the potential speed gain (which may be difficult to ascertain before actually performing the optimization). In contrast to functional bugs, the lack of reliable automated oracles for performance bugs means that developers cannot easily find which tests fail, as in step (2). As a result, because developers need to inspect all tests/profiles in step (3), they can use only a small number of performance tests in step (1). In sum, developers follow the current process of testing for performance bugs not because it has advantages, but because developers have no reliable alternatives.

An automated oracle for performance bugs would enable developers to test for performance bugs using the well established process of testing for functional bugs. Unfortunately, profilers cannot be used as effective oracles for three reasons. First, profilers give a *report for each test*, thus running many tests results in many reports, not just a few failing tests as for a typical functional oracle. Second, profilers may miss a performance bug *even when it is executed*: if the buggy code is not slow compared to the rest of that execution, it is not ranked high by the profiler and is likely ignored by the developer. Many performance bugs manifest by significantly degrading performance only for particular input conditions [8]–[11], and the profiled inputs cannot cover all possible conditions. Third, profilers report what *takes* time but not what *wastes* time, i.e., they do not distinguish truly necessary (albeit expensive) work from the likely unnecessary computation. In other words, profilers are highly useful when the developer wants to *localize* a slow code region but are not effective when the developer needs to *decide* if a test likely exposes a performance bug and thus needs further inspection.

This paper presents TODDLER, a novel *oracle* for performance bugs. In brief, TODDLER reports tests that execute loops whose computation is repetitive and very similar across iterations. The intuition is that such loops are likely perfor-

[1]"Performance bug" is a well accepted term in some communities, e.g., Mozilla Bugzilla defines it as "A bug that affects speed or responsiveness" [1]. However, others prefer "performance problem" or "performance issue", because these problems differ from functional bugs. We take no position on this and use "performance bug" and "performance problem" interchangeably.

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

mance bugs that waste time: because the work is repetitive and similar, it could be done faster. We designed TODDLER based on two observations about performance bugs. First, many severe performance bugs (over 50% in the study in Section II) are contained by nested loops: if an inefficient code region is executed outside of a nested loop, then the inefficiency itself needs to be very severe (e.g., slow I/O) for the code region to have a real impact on the overall program performance. Second, wasted computation is often reflected by repetitive and partially similar memory accesses across loop iterations: if a group of instructions repeatedly accesses similar memory values, then those instructions probably compute similar results.

We implemented a full-blown TODDLER tool for Java and a simple prototype for C/C++. Our experiments with 11 previously known, real-world performance bugs from 9 Java projects show that TODDLER is able to find all these bugs. Our C/C++ prototype also finds 6 previously known bugs in GCC, Mozilla, and MySQL. Moreover, using TODDLER helped us identify *42 new real-world bugs* in six popular Java projects: Ant, Google Core Libraries, JUnit, Apache Collections, JDK, and JFreeChart. Based on our reports, developers so far have fixed 10 bugs and confirmed 6 more as real bugs, and the Apache Collections developers even invited the first paper author to become a project committer. Our bug reports are linked from `http://mir.cs.illinois.edu/toddler`.

II. STUDY OF PERFORMANCE BUGS

We study over 100 performance bugs from open-source projects to identify how these bugs depend on loops. We study both Java and C/C++ projects to obtain more generality of our findings. These bugs were collected independently of TODDLER in a recent study on performance bugs [9], but their relationship to loops was not analyzed in detail.

Our study shows that about 90% of performance bugs involve loops, and more than 50% of performance bugs involve at least two levels of loops. The bugs that involve nested loops can be categorized along two dimensions:

- Is the performance problem in the inner or the outer loop?
- Is the performance problem caused by redundant computation or inefficient computation? We define *redundant computation* as the same computation being unnecessarily repeated on the same set of data with the same result.

We next describe the four types of real-world performance bugs categorized along the above two dimensions, and then discuss how this understanding of real-world bugs can guide our bug-detection design. For space reasons, we will give code examples only for two categories (but covering both inner and outer loops, as well as redundant and inefficient computation).

A. Categories of Severe Performance Bugs

Category 1 (Redundancy in Outer Loops): Redundant computation is conducted across iterations of an outer loop. This redundant computation involves an expensive inner loop, which makes the performance problem severe. Problems of this type are usually difficult for compilers to optimize, because they involve nested loops and usually many functions. They are usually fixed by storing and reusing results from previous loop iterations.

```
1  // Simplified from the XYPlot class in JFreeChart
2  public void render(...) {
3    for (int item = 0; item < itemCount; item++) { // Outer Loop
4      renderer.drawItem(...item...); // Calls drawVerticalItem
5    }
6  }
7  // Simplified from the CandlestickRenderer class in JFreeChart
8  public void drawVerticalItem(...) {
9    int maxVolume = 1;
10   for (int i = 0; i < maxCount; i++) { // Inner Loop
11     int thisVolume = highLowData.getVolumeValue(series, i).intValue();
12     if (thisVolume > maxVolume) {
13       maxVolume = thisVolume;
14     }
15   }
16   ... = maxVolume;
17 }
```

Fig. 1. A JFreeChart bug with a redundancy in the outer loop

Figure 1 demonstrates such a bug from JFreeChart, a popular Java framework for drawing charts. This bug is particularly severe, because it causes the chart display to freeze. The outer loop iterates over all the items in a data set (line 3) and for each item calls the method `drawItem`, which in turns calls the method `drawVerticalItem`. The inner loop (line 10) in `drawVerticalItem` computes the maximum volume (line 12) of all the items in the data set. The repeated computation of maximum is redundant, because the volumes of the items do not change between calls. Thus, the inner loop can be performed only once, not in every iteration of the outer loop. Indeed, to fix this bug, the developer changed the code to cache and reuse the maximum volume.

Category 2 (Redundancy in Inner Loops): Redundant computation is conducted across iterations of an inner loop. This computation waste is amplified by outer loops that dynamically call the inner loop many times. Performance problems of this type are difficult for compilers to optimize when the redundant computation involves function calls. They are usually fixed by hoisting computation out of the loop.

Category 3 (Inefficient Outer Loops): The program has an expensive but necessary inner loop. Unfortunately, this loop is inefficiently used by an outer loop, which leads to severe performance problems. Problems of this type cannot be optimized by compilers, because they require deep understanding of code. They are usually fixed by changing outer loops so that the inner loop will execute less frequently.

Category 4 (Inefficient Inner Loops): The inner loop conducts an inefficient, but not redundant, computation. This inefficiency is amplified by an outer loop that uses each iteration to execute the inner loop on a slightly different data set. Again, problems of this type cannot be optimized by compilers, because they require deep understanding of code. Their patches need to find a more efficient or incremental algorithm to replace the inner loop, which often can be achieved with a more appropriate data structure for the data set under operation.

```
1  // SetDecorator class in Google Core Libraries contained this method call
2  set.removeAll(arrayList);
3  // Simplified from the AbstractSet class in the standard Java library
4  public boolean removeAll(Collection<?> c) {
5    if (size() > c.size()) {
6      for (Iterator<?> i = c.iterator(); i.hasNext(); )
7        remove(i.next());
8    } else {
9      for (Iterator<?> i = iterator(); i.hasNext(); ) { // Outer Loop
10       if (c.contains(i.next())) {
11         i.remove();
12       }
13     }
14   }
15 }
16 // Simplified from the ArrayList class in the standard Java library
17 public boolean contains(Object o) {
18   for (int i = 0; i < size; i++) { // Inner Loop
19     if (o.equals(elementData[i]))
20       return true;
21   }
22   return false;
23 }
```

Fig. 2. A Google Core Libraries bug with an inefficient inner loop. This was a *previously unknown* bug found by TODDLER.

Figure 2 demonstrates an example from Google Core Libraries (GCL). This is a *previously unknown bug found by* TODDLER. After we reported it, GCL developers not only fixed this bug but also searched through their entire codebase for similar code patterns and fixed 8 other classes affected by similar bugs. (We count these 9 instances as *one bug* not 9 bugs.) The GCL code called the `removeAll` method on a `Set` object, passing it an `ArrayList` object as a parameter. The `removeAll` method removes from the set `this` all the elements contained in the specified collection `c`. The method has a performance optimization that chooses whether to iterate over the set `this` or the collection `c` based on their sizes (line 5), under the assumption that the cost of `contains` and `remove` operations are similar for the set and the collection when they have similar sizes. In the `else` branch, the outer loop iterates over each element of `this` and checks if `c` contains the element (lines 9–13).

When `c` is an `ArrayList`, `contains` performs a linear search (lines 18–21), which is inefficient, so it would have been better to iterate over `c` and call `remove` on the set because it has a more efficient inner loop. Indeed, the GCL developers changed their code, replacing the call to `removeAll` by conceptually inlining the body of `removeAll` and keeping only the `then` branch from the body. In general, the solution for this category is to simplify the inner-loop computation.

B. Implications

Why do developers need automated support for performance testing? The above examples demonstrate that many performance bugs are difficult to avoid, because they involve library functions or APIs whose performance features are opaque to developers. In addition, a lot of time-consuming computation, such as many inner loops in our examples, is embedded in code written by different developers. As shown in Figure 2, GCL developers did not initially consider that the performance of the Java library method `AbstractSet.removeAll` is sensitive to the data structures used for parameters, and this information is not even stated in the documentation for `removeAll`. Tool support is needed to help developers detect these hard-to-avoid performance bugs.

Why do we focus on nested-loop performance bugs? Bugs that involve nested loops usually have severe performance impact. The reason is that the inner loop represents an expensive computation inside the outer loop, and the outer loop amplifies the performance penalty of the inner loop. For example, in the JFreeChart bug from Figure 1, the inner loop is slow, but if executed only once, it cannot have a significant effect on performance; however, if executed many times in the outer loop, it causes the chart display to freeze.

How can we detect nested-loop performance bugs? A common feature of above nested-loop performance bugs is that they often involve *repeated memory-access patterns*. Bugs from Category 1 conduct redundant computation across outer-loop iterations. A big chunk of the computation in each outer-loop iteration repeats the computation from an earlier iteration with the same input and the same result. Hence, outer-loop iterations share long sequences of memory reads that return the same values. For example, the iterations of the outer loop in Figure 1 share a long sequence of reads inside the `intValue` method (line 11). Bugs from Category 2 conduct redundant computation during every iteration of an inner loop, which results in memory reads that repeatedly return the same value. Bugs from Categories 3 and 4 have less regular patterns than bugs from Categories 1 and 2, but the memory-access similarities are still strong. The outer-loop iterations in bugs from Categories 3 and 4 often work on similar data sets. That is the reason why developers can effectively optimize these bugs. That is also the reason why there are usually memory reads that return similar sequences of values across outer-loop iterations. In sum, looking for repeated memory-access patterns is an effective way to look for performance bugs from all four categories.

III. TODDLER DESIGN AND IMPLEMENTATIONS

Motivated by the study in Section II, we have developed TODDLER, an automated oracle that finds likely performance bugs by looking for loops that read similar sequences of values across iterations. TODDLER considers such similar sequences to be a strong indication of redundant or inefficient computation and reports such loops as performance bugs.

TODDLER is a dynamic technique. It instruments the code under test, runs each test from a given test suite, and reports only the tests that contain loops with similar sequences. We first describe the instrumentation that TODDLER adds. We then describe the data structures and algorithms that TODDLER uses for storing information about reads and finding similarity among sequences. We finally discuss our two implementations of TODDLER in a full-blown tool for Java and a simple prototype for C/C++.

A. Instrumentation

To monitor loops and read instructions, TODDLER instruments the code, both the application under test and the libraries

```
1  StartLoop(L1)
2  StartIter    Read(i1, v1)
3               StartLoop(L2)
4               StartIter Read(i2, v2)
5               StartIter Read(i2, v3) Read(i3, v4)
6               StartIter Read(i3, v5)
7               StartIter Read(i2, v6) Read(i3, v7)
8               FinishLoop(L2)
9  StartIter
10              StartLoop(L2)
11              StartIter Read(i2, v8)
12              StartIter Read(i2, v9) Read(i3, v10)
13              StartIter Read(i2, v11) Read(i3, v12)
14              StartIter Read(i3, v13)
15              FinishLoop(L2)
16              Read(i4, v14) Read(i5, v15)
17 FinishLoop(L1)
```

Fig. 3. Example events produced by running instrumented code

```
1  // Instruction pointer and its dynamic context
2  class IPCS { int IP; CallStackHash cs; }
3  // Value of a memory location
4  class Val { long val; }
5  // IPCS-sequence of values read by an IPCS in one iteration
6  class Seq { List⟨Val⟩ list; }
7  // Dynamic loop record
8  class DynLoop {
9    int id; // static id of the loop
10   CallStackHash cs; // calling context
11   int iterations; // number of iterations
12   // map each IPCS encountered during loop execution...
13   // ...to values read by the IPCS in the iterations
14   Map⟨IPCS , List⟨Seq⟩⟩ map;
15 }
```

Fig. 4. Data structures for storing and processing IPCS-sequences

it depends on, because many performance bugs are caused by the misuse of libraries. For loops, the instrumentation is straightforward: TODDLER analyzes the code, assigns a unique ID for each static loop, and inserts in the code three types of method calls that inform the TODDLER runtime whenever a loop starts, a loop iteration starts, or a loop finishes. For read instructions, the instrumentation itself is also simple: for each instruction that reads object fields or array elements from the heap (e.g., Java bytecode instructions `GETFIELD` or `AALOAD`), TODDLER inserts a method call that informs the TODDLER runtime about the value read by the instruction and the call stack within which the instruction is executed. Note that TODDLER identifies a read instruction by both the static occurrence of the instruction in the code *and* the dynamic context (i.e., the call stack) in which the instruction executes. We use the term *IPCS* (instruction pointer + call stack) to refer to a static instruction with its dynamic context.

B. Collecting IPCS-Sequences

We use the term *IPCS-sequence* to refer to the sequence of values read by all dynamic instances of an IPCS I during an iteration of a loop L. Note that, when I is inside an inner loop of L, the IPCS-sequence for the outer loop L is likely to contain more than one element. Also note that TODDLER builds *one IPCS-sequence per IPCS* rather than one IPCS-sequence per the entire loop iteration, and thus a loop iteration has as many IPCS-sequences as it has IPCSs.

To illustrate, Figure 3 shows an example stream of events produced when some instrumented code is executed; $\mathtt{i}_N$ represents an IPCS, and $\mathtt{v}_M$ represents a value read. From these events, TODDLER creates IPCS-sequences of values read by the same IPCS during a loop iteration. For example, for the *outer* loop `L1`, TODDLER would create IPCS-sequences $\mathtt{i}_1{:}[\mathtt{v}_1]$, $\mathtt{i}_2{:}[\mathtt{v}_2,\mathtt{v}_3,\mathtt{v}_6]$, and $\mathtt{i}_3{:}[\mathtt{v}_4,\mathtt{v}_5,\mathtt{v}_7]$ for the first iteration and $\mathtt{i}_2{:}[\mathtt{v}_8,\mathtt{v}_9,\mathtt{v}_{11}]$, $\mathtt{i}_3{:}[\mathtt{v}_{10},\mathtt{v}_{12},\mathtt{v}_{13}]$, $\mathtt{i}_4{:}[\mathtt{v}_{14}]$, and $\mathtt{i}_5{:}[\mathtt{v}_{15}]$ for the second iteration.

Note that the IPCS-sequences for the innermost loops have length 1, e.g., for the first dynamic instance of the *inner* loop `L2`, the IPCS-sequences would be just $\mathtt{i}_2{:}[\mathtt{v}_2]$, $\mathtt{i}_2{:}[\mathtt{v}_3]$, and $\mathtt{i}_2{:}[\mathtt{v}_6]$ for $\mathtt{i}_2$ and similar for $\mathtt{i}_3$. Also note that an IPCS need not occur in every iteration of a loop (e.g., $\mathtt{i}_2$ does not occur in the third iteration of the first dynamic instance of `L2`). In that case, TODDLER still creates an IPCS-sequence (for `L1`) of consecutive values read for the same IPCS even if these values are not read in consecutive loop iterations (of `L2`).

While this example illustrates TODDLER only on the loop nesting depth of two, TODDLER handles larger nesting depths in the same manner, by appending IPCS-sequences for the same IPCS. For example, if one iteration of some loop `L0` had the events shown in Figure 3, then for that iteration of `L0`, TODDLER would create $\mathtt{i}_1{:}[\mathtt{v}_1]$, $\mathtt{i}_2{:}[\mathtt{v}_2,\mathtt{v}_3,\mathtt{v}_6,\mathtt{v}_8,\mathtt{v}_9,\mathtt{v}_{11}]$, $\mathtt{i}_3{:}[\mathtt{v}_4,\mathtt{v}_5,\mathtt{v}_7,\mathtt{v}_{10},\mathtt{v}_{12},\mathtt{v}_{13}]$, $\mathtt{i}_4{:}[\mathtt{v}_{14}]$, and $\mathtt{i}_5{:}[\mathtt{v}_{15}]$.

C. Data Structures

Figure 4 shows the data structures that TODDLER uses to store information about loops. `IPCS` has an `IP` that statically determines the instruction (e.g., its class, method, and bytecode offset within the method in Java) and the call stack that represents the dynamic context in which the instruction executes. (Call stacks can be efficiently computed using hashing [12].) `Val` represents a value read by an instruction, which is either a primitive value or an object ID (obtained with `System.identityHashCode()` in Java). Note that the ID is of the object being *returned* by the read, not of the object being *dereferenced*. For example, in `e.next`, the ID is of `e.next` not of `e`. `Seq` is an IPCS-sequence of values read by the same `IPCS` in one loop iteration. `DynLoop` records information about one dynamic loop instance: the static loop ID (its class, method, and bytecode offset within the method in Java), the call stack in which the loop executes, the number of loop iterations, and the IPCS-sequences across all iterations for each `IPCS`. For example, for $\mathtt{i}_2$, the two IPCS-sequences of the *outer* (L1) loop are $\mathtt{i}_2{:}[[\mathtt{v}_2,\mathtt{v}_3,\mathtt{v}_6],[\mathtt{v}_8,\mathtt{v}_9,\mathtt{v}_{11}]]$.

D. Algorithm for Finding Performance Bugs

Figure 5 shows the pseudo-code of the top-level function. TODDLER checks for potential performance bugs in each dynamic loop that had more than a few iterations (by default, `minIter=10`; this threshold is a configurable parameter of our algorithm, and Section IV-D discusses the impact of the parameters). For each `DynLoop`, TODDLER finds all `IPCS`s that have similar IPCS-sequences across loop iterations. If there is any such `IPCS`, TODDLER reports a performance bug.

Given a test suite, TODDLER runs each test, collects `DynLoop` objects, and reports a set of static loops that have

```
1  // One parameter for loops
2  int minIter; // absolute number of loop iterations
3
4  // Input: the record of a dynamic loop
5  // Output: whether this loop has performance bugs
6  boolean hasPerformanceBug(DynLoop loop) {
7    return !(computeSimilarIPCSs(loop).empty());
8  }
9
10 // Input: the record of a dynamic loop
11 // Output: IPCSs that read similar values across iterations
12 Set⟨IPCS⟩ computeSimilarIPCSs(DynLoop loop) {
13   Set⟨IPCS⟩ similarIPCSs = new Set⟨IPCS⟩();
14   // ignore very small loops
15   if (loop.iterations < minIter) return similarIPCSs;
16   for (curIPCS : loop.map.keyset())
17     // compare IPCS-sequences for iterations in which curIPCS occurs
18     if (areSimilarIterations(loop.map.get(curIPCS)), loop.iterations)
19       similarIPCSs.add(curIPCS);
20   return similarIPCSs;
21 }
```

Fig. 5. The top-level function for TODDLER

```
1  // Two parameters for loop iterations
2  float minSeqRatio; // relative number of IPCS-sequences in the loop
3  float minSimRatio; // relative number of similar iterations
4
5  // Input: IPCS-sequences for all iterations of a loop
6  // Output: whether IPCS reads similar values across iterations
7  boolean areSimilarIterations(List⟨Seq⟩ seqs, int iterations) {
8    // ignore IPCS that occurs in a small fraction of iterations
9    if ((seqs.size() / iterations) < minSeqRatio) return false;
10   int similar = 0;
11   for (int i = 0; i < seqs.size()−1; i++)
12     if (areSimilarSequences(seqs[i], seqs[i+1])) similar++;
13   return (similar / (seqs.size()−1)) >= minSimRatio;
14 }
```

Fig. 6. Checking the similarity throughout a loop

```
1  // Two parameters for IPCS-sequences of values
2  int minLCS; // absolute length of the longest common substring
3  float minLCSRatio; // relative length of the longest common substring
4
5  // Input: two IPCS-sequences
6  // Output: whether two IPCS-sequences are similar
7  boolean areSimilarSequences(Seq S1, Seq S2) {
8    lcs = longestCommonSubstring(S1, S2).size();
9    lcsRatio = lcs / min(S1.size(), S2.size());
10   return (lcs >= minLCS) && (lcsRatio >= minLCSRatio);
11 }
```

Fig. 7. Checking the similarity of two IPCS-sequences

similar IPCS-sequences for at least one test. For each static loop, TODDLER generates a set of records that help in understanding and debugging the problem. Each record contains the test that executes the loop, the call stack for the loop, the static IP of the instruction that reads similar values, the call stack for that instruction, and statistics about similarity.

Note that TODDLER can find the same loop to be repetitive for multiple tests. Rather than printing a report for each test and each loop, TODDLER *clusters* these reports based on the *static* outer loop. Clustering is commonly used for grouping failure reports in testing [13], [14].

E. Measuring Similarity

Figure 6 shows the pseudo-code for finding similar IPCS-sequences across loop iterations. TODDLER compares consecutive IPCS-sequences for the same `IPCS`. As mentioned in Section III-A, an IPCS may not be executed in every iteration of a loop. TODDLER computes the ratio of the number of IPCS-sequences to the number of loop iterations and ignores `IPCS`s that occur in a small ratio of iterations, because even if the computation at these `IPCS`s is similar and could be optimized, they may not be an expensive part of the entire loop. (By default, `minSeqRatio`=45%.)

To compare the IPCS-sequences of an IPCS inside a loop L, TODDLER determines whether these IPCS-sequences are similar *throughout* L based on the relative number of similar *consecutive* IPCS-sequences. The IPCS-sequences are considered similar throughout loop L if and only if the ratio is larger than the threshold. (By default, `minSimRatio`=70%.)

Redundant and inefficient computation can be reflected not only by IPCS-sequences that are exactly the same across iterations, such as the IPCS-sequences from `intValue()` in Figure 1, but also by IPCS-sequences that are slightly different across iterations, such as the IPCS-sequences for `elementData[i]` in Figure 2. Thus, we need to judge whether two IPCS-sequences are *similar* enough to represent potential performance problems.

Figure 7 shows the pseudo code of this algorithm. TODDLER uses the *longest common substring* [15] to measure the similarity between two IPCS-sequences. (Note that *substring* refers to the *consecutive* occurrences of values in the IPCS-sequences, while *subsequence* would refer to the *potentially non-consecutive* occurrences of values.) The longest common substring can be computed in $O(nm)$ time where n and m are the lengths of the two IPCS-sequences [15]. We define two IPCS-sequences to be similar if both the absolute and relative length of their longest common substring are above thresholds. (By default, `minLCS`=7 and `minLCSRatio`=70%.)

F. Filtering Reads

TODDLER can filter reads that have repetitive values but are unlikely to indicate performance bugs. First, TODDLER ignores IPCS-sequences that repeat only one value. For example, an inner loop of the form `for (int i = 0; i < this.size; i++)` repeatedly reads the value for `this.size` but does not contain a performance bug. Note that this heuristic may cause TODDLER to lose some Category 2 bugs. For example, if `this.size` is returned by a synchronized getter method, which is slower than just reading `this.size`, one may want to pull the getter method call out of the loop. TODDLER considers all operations to take an equal amount of time, and therefore does not report the repeated getter method calls as a performance bug. Future implementations can add timing information to TODDLER.

Second, TODDLER for Java ignores reads that happen in the class initializer methods because these are executed only once per class loading, so even if the code contains a bug, developers may not want to change it. Third, TODDLER allows the users to specify a set of fields and methods to be ignored, when the users do not expect them to be indicative of performance bugs. TODDLER ignores IPCSs that either read a specified field or execute in a context where a specified method

ID	Application	Description	LoC	Known Bugs	New Bugs
#1	Ant	build tool	109,765	1	8
#2	Apache Collections	collections library	51,416	1	20
#3	Groovy	dynamic language	136,994	1	0
#4	Google Core Libraries	collections library	156,004	2	10
#5	JFreeChart	chart framework	64,184	1	1
#6	JMeter	load testing tool	86,549	1	0
#7	Lucene	text search engine	320,899	2	0
#8	PDFBox	PDF framework	78,578	1	0
#9	Solr	search server	373,138	1	0
	JDK standard library				2
	JUnit testing framework				1
SUM				**11**	**42**

Fig. 8. Applications used in experiments, previously known bugs, and new bugs found with TODDLER.

is on the call stack. For example, some fields are used as *indexes* and can appear in an inner loop as `for (...) { ... this.cursor++; ... }`; if the outer loop resets `cursor`, the IPCS-sequence would repeat, but repeatedly reading the index itself does not indicate inefficient or redundant computation. As another example, appending strings in a loop often leads to repeated work, and in fact, it is an anti-pattern in Java to append many `String` objects. However, to simplify coding, many times developers do append strings in loops, and may not want to be bothered with reports of such coding patterns. By default, TODDLER ignores *only three fields and four `toString`/`append` methods* from the standard JDK library `java.util` classes. Note that specifying these library fields and methods is done *only once* for all applications that use the library.

G. Implementations

We implemented the TODDLER technique in a full-blown tool for Java, which we also call TODDLER, and a simple prototype for C/C++. Our Java implementation is based on static Java-bytecode instrumentation, using Soot 2.4.0 [16]. TODDLER uses Soot to instrument every instruction that reads an object field or an array element, the start of each loop, the start of each loop iteration, and the exit of each loop. The implementation closely follows the pseudo-code algorithms presented earlier. It performs similarity checks *online*, i.e., collects IPCS-sequences of values read in a `DynLoop` object and, whenever the program exits a loop, calls the `hasPerformanceBug` function from Figure 5 to process the `DynLoop` object and decide if there is a performance bug. Section IV-E discusses our C/C++ prototype.

IV. EXPERIMENTAL RESULTS

Our evaluation focuses on the Java version of TODDLER and uses 9 popular Java codebases. Figure 8 lists basic information about these codebases. We first evaluated TODDLER on 11 previously known real-world performance bugs and on over 173,000 existing functional tests from these codebases. We then *settled on the values for the* TODDLER *parameters* and evaluated it on newly written performance tests. Our experiments found *42 real-world performance bugs* in these codebases (39 in the application code and 3 in the libraries they use).

The rest of this section first presents our experiments with the 11 previously known bugs. It then presents our experiments with performance tests and the new bugs that we found. It next presents the evaluation with the existing functional tests. It finally presents a sensitivity analysis of the parameter values. Unless otherwise specified, all the experiments use the following default values: `minIter`=10, `minSeqRatio`=45%, `minSimRatio`=70%, `minLCS`=7, `minLCSRatio`=70%.

We conduct all experiments where time is measured on an AMD Athlon machine with 2.1GHz CPU, 3GB memory, and Oracle/Sun JVM version 1.6.0. We also conduct experiments where time is not measured on a cluster of machines; while TODDLER does not need a cluster for regular use, we needed it for our extensive experiments.

A. Experiments with Previously Known Bugs

To evaluate bug-detection coverage, accuracy, and overhead of TODDLER, we first used 11 known real-world bugs from the 9 codebases. We searched the respective bug-tracking databases to collect these bugs; they were reported by the users of these applications and the bug description clearly marks them as performance bugs.

We run TODDLER on a performance test related to the bug report for each of the 11 bugs. Because each test is supposed to reveal a bug, we effectively evaluate if TODDLER has *false negatives* that miss some bugs. We compare the results of TODDLER with the results of a traditional profiler ran on the same tests. As explained in Section I, profilers are not designed to detect performance bugs, but are the only traditional tool that developers could use without TODDLER. Specifically, we use HPROF [17], the standard Java profiler. It outputs a ranked list of methods (more precisely, calling contexts) that consume the most time. We measure how highly HPROF ranks the buggy method (that contains the buggy code region). Additionally, for these 11 tests, we compare the runtime overheads of TODDLER and HPROF.

1) Bug Detection Results: Figure 9 summarizes the results for the 11 bugs. TODDLER finds all the bugs (no false negatives) and produces only one false positive. Specifically, for bug #8, TODDLER produces two reports: one showing the real bug and one being a false positive. (Section IV-C discusses false positives.) TODDLER finds these 11 bugs because they involve at least two levels of loops and have similar sequences of values read across loop iterations. In fact, most of these bugs have so strongly similar sequences that TODDLER can detect them under a wide range of threshold settings. (Section IV-D discusses sensitivity to threshold settings.)

Figure 9 also shows the results for HPROF. We use it with the `cpu=times` option as it gives more accurate results than `cpu=samples`, though at a higher overhead. However, even with `cpu=times`, the results of HPROF for the same code and same input can vary from one run to another. Therefore,

Known Bug	Bug Detected?		False P.	Rank	Slowdown	
	TODD.	HPROF	TODD.	HPROF	TODD.	HPROF
#1	✓	-	0	19.3	13.7	4.2
#2	✓	✓	0	1.0	10.0	2.1
#3	✓	✓	0	3.7	15.5	3.7
#4.1	✓	✓	0	1.8	9.0	3.8
#4.2	✓	-	0	5.3	7.5	3.2
#5	✓	-	0	53.7	13.4	8.8
#6	✓	-	0	10.3	8.5	1.9
#7.1	✓	-	0	7.7	6.8	2.5
#7.2	✓	✓	0	3.1	25.4	3.1
#8	✓	-	1	18.8	51.8	12.1
#9	✓	-	0	178.3	114.2	7.1
SUM	**11**	**4**	**1**		**15.9X**	**4.0X**

Fig. 9. Comparison of TODDLER and HPROF for bug-triggering tests.

we ran each test under HPROF 10 times and show the mean ranking of the buggy method.

The developer is unlikely to inspect more than a handful of methods reported by a profiler. If we consider that HPROF correctly detects a bug when the buggy method ranks in top 5, then HPROF detects only 4 out of 11 cases that TODDLER detects. On the positive, HPROF ranks bug #2 consistently as number one. On the negative, for 5 out of 11 bugs, HPROF does not rank the buggy method even in the top ten. For example, bug #9 comes from a text-search server, Solr. The method with the performance bug constructs a set of strings that represent filter keywords. Under normal server setting, this set is small, and the method consumes only about 0.1% of the total search-query time. As a consequence, it ranks only about 178th in the profiling results.

A careful reader may wonder if an easier approach would suffice to find the bugs that TODDLER finds: could we simply report all nested loops as potentially buggy? We added code to count nested loops during an execution, more precisely static outer loops that dynamically execute at least one inner loop. For the 11 tests, the number of such outer loops ranges from 1 to 12, and the total number of such loops is 38. Thus, a naïve technique that reports every nested loop as a performance bug would have 27(=38-11) false positives for just these 11 bugs. In contrast, TODDLER can identify truly performance-wasting nested loops by analyzing memory-access patterns and reports only one false positive for these 11 cases.

2) Performance Results: The last two columns of Figure 9 show the slowdown that TODDLER and HPROF have over an execution with no tool for the 11 bug-triggering tests. TODDLER causes, on average, a 15.9X slowdown that comes from monitoring read accesses and comparing IPCS-sequences. Our current implementation of TODDLER is about 4 times slower than HPROF. In the future we plan to further reduce the overhead of TODDLER through sampling techniques and static analysis.

B. Experiments with New Bugs and Performance Tests

We further evaluate bug-detection coverage and accuracy of TODDLER by applying it on *performance tests*, which is the intended usage scenario for TODDLER. To avoid the bias of us as tool authors manually writing tests, we use three sets of tests not written by us: (1) automatically generated tests, (2) tests manually written by an undergraduate student *familiar* with performance testing ("expert"), and (3) tests manually written in a controlled experiment by 8 graduate and undergraduate students *unfamiliar* with performance testing ("novices"). We use these different sets to assess how TODDLER works for tests with various characteristics.

We focus our efforts on collection classes because they are widely used and make both automated generation [18] and manual writing of tests easier than domain-specific applications such as Groovy or Lucene. Ant, Apache Collections, and Google Core Libraries (GCL) implement collection classes. The performance tests for collections follow a simple pattern: create some empty collection(s), insert several elements into the collection(s), and finally call a method under test. (Note that performance tests need not necessarily check the functional results of the methods.) The collections for performance tests should not be very small, e.g., when testing `Collection.removeAll(Collection c)`, both `this` and `c` should have a reasonable number of elements, say, over 20 each; if they had a very small number, say, 2 each, it is unlikely the test would be useful for performance testing.

We wrote a simple library to automate generation of performance tests for collections. Our library can generate individual collections of various types, sizes, element types, and element values, e.g., generate an `ArrayList<Integer>` with elements `1-50`. Moreover, our library can generate multiple collections with various relationships in terms of types (collections of same or different types), sizes (collections of same, smaller, larger sizes), and intersection of elements (collections that are disjoint, equal, or partially intersect), e.g., generate a set with elements `1-50` and a list with elements `1-75`. Our library supports exhaustive and random selection of combinations of these relationships. The design goal for the library was not to extensively cover all the cases but to provide some reasonable tests for TODDLER.

We collected two types of manually written tests. We asked the "expert" to write tests for any methods in GCL and Apache Collections. We asked each "novice" to spend an hour writing tests for a given set of 10 methods in a class from Apache Collections; one of these 10 methods contained a known performance bug, and we wanted to check if the students would write tests that find this bug.

Figure 10 shows the number of tests generated/written for each codebase, the number of dynamic loops executed, and the number of reports that TODDLER produces. We examined all these reports to identify if they are real bugs or false positives.

We found 35 new, previously unknown performance bugs in Ant, Apache Collections, GCL, and even in a JDK class called from these projects; based on our reports, developers so far have fixed 8 of these bugs and confirmed 6 more as real bugs. TODDLER was highly effective in finding performance bugs using both automatically generated and manually written tests. Both types of tests found bugs, and sometimes found the same bugs. (Our study used older versions of GCL and Apache

Who	App	Tests	#Dyn. Loops	Bugs	Bugs in Test	False Pos.	Sum
Auto	#1	691	13,748	5	0	1	6
	#2	3,375	342,821	18	1	2	21
	#4	1,703	423,406	9	0	0	9
Expert	#2	60	6,761	10	0	1	11
	#4	60	6,319	2	0	0	2
Novice	#2	14	2,057	1	6	0	7
	#2	20	3,043	2	0	0	2
	#2	5	1,868	1	0	0	1
	#2	18	3,269	1	0	0	1
	#2	5	606	0	0	0	0
	#2	28	4,502	2	0	0	2
	#2	30	3,810	1	0	0	1
	#2	5	1,996	1	0	0	1
Unique Bugs Found:				**35**	**FPs:**	**4**	

Fig. 10. Experiments with performance tests. Note that the same bug may be found by different automatically generated and manually written tests.

Collections, without the fixes for the bugs we reported.) Surprisingly, some "novice"-written tests found two bugs in a class that we expected to have only one bug.

We also found 7 performance bugs where the test code itself is unnecessarily slow. For example, the "novice"-written tests had assertions that check the method results, and the assertions themselves use rather slow code, e.g., nested loops that search in lists but could have searched in sets. If such loops appeared in the code under test, they would be definite bugs that should be changed.

C. Experiments with Functional Unit Tests

The first two sets of experiments used tests written for performance, which is the intended usage scenario for TODDLER. To further evaluate TODDLER, we run it on the *functional* JUnit tests that come with the 9 codebases used in our evaluation. Note that this is *not the intended usage scenario*: a developer would *not use functional tests for performance testing* and thus would not use TODDLER on the functional JUnit tests. We perform these experiments *only* to stress-evaluate TODDLER.

Our experiments use 173,439 tests shown in Figure 11. These tests execute 24,810–3,526,496 dynamic loops (and 1,181,628–54,054,728 dynamic iterations) per codebase, a challenge for the run-time monitoring scalability. The tests also cover 21–919 unique static loops that contain nested loops per codebase, a challenge for the bug-detection accuracy.

TODDLER successfully ran for this extensive evaluation and reported 43 static loops as having similar memory accesses and thus potential performance bugs. We examined all these reports and found 7 real bugs. For JFreeChart (#5), one bug is in the JFreeChart code itself and the other in the standard JDK library. For Apache Collections (#2), one bug we reported is already fixed, and the other three bugs are similar to three bugs we previously reported and developers resolved by changing the Javadoc documentation to clarify the performance problems. For Ant (#1), all three bugs have been already fixed in the latest release. (Our experiments use older versions of the codebases.)

App	# Tests	# Dynamic Loops	Bugs	Bugs in Test	False Pos.	Sum
#1	675	877,362	3	0	3	6
#2	31,105	3,526,496	4	7	1	12
#3	464	281,596	0	0	0	0
#4	138,997	2,574,756	0	0	4	4
#5	332	514,824	2	0	1	3
#6	164	88,548	0	0	1	1
#7	675	1,488,977	0	0	4	4
#8	42	24,810	0	0	0	0
#9	985	1,395,494	0	0	13	13
Unique Bugs Found:			**7**	**FPs:**	**27**	

Fig. 11. Experiments on JUnit *functional* tests. Note that this is *not the intended usage scenario for* TODDLER; a developer would *not use functional tests for performance testing*.

For Apache Collections (#2) we also found 7 performance bugs in tests, where the test code is unnecessarily slow and would need to be fixed had it been in the application code.

The remaining 27 reports are false positives due to three causes. First, in 10 reports, the test input itself contains a lot of repetition and similar values, so TODDLER detects similarity due to the specific input provided, not because the computation is repetitive in general. Such false positives could be eliminated by using less repetitive test inputs. Second, in 3 reports, the code performs some computation on all possible pairs of values from two data sets. Such code is naturally repetitive, but the repetitions are useful computation, not performance bugs. Such false positives may be eliminated by analyzing the data flow of computation results, but such an analysis is beyond the scope of this paper. Third, in 14 reports, the computation is truly repetitive, but removing the repetition would be too complex or would not provide clear speedup, so a developer is unlikely to change the code.

D. Parameter Sensitivity

The false-positive and false-negative rates of TODDLER are affected by the values for the five parameters described in Section III. All these parameters provide the minimum threshold that loops/iterations/sequences need to satisfy to be deemed indicative of performance bugs. Hence, larger thresholds could lead to fewer false positives but more false negatives, while smaller thresholds could lead to more false positives but fewer false negatives. We experimented with various threshold values to understand their impact.

Figure 12 shows the results for several configurations. For each configuration, we change only one threshold value and keep the other four at the default setting. To evaluate the impact on false negatives, we apply TODDLER on the 11 bug-triggering tests for previously known bugs (Section IV-A) and count the number of bugs found. To evaluate the impact on false positives, we cannot use TODDLER in the intended scenarios from sections IV-A and IV-B, because they have few false positives. We thus use the functional tests (Section IV-C). The default configuration finds all 11 known bugs in the experiments from Section IV-A and reports 27 false positives in the experiments from Section IV-C. Figure 12 plots the

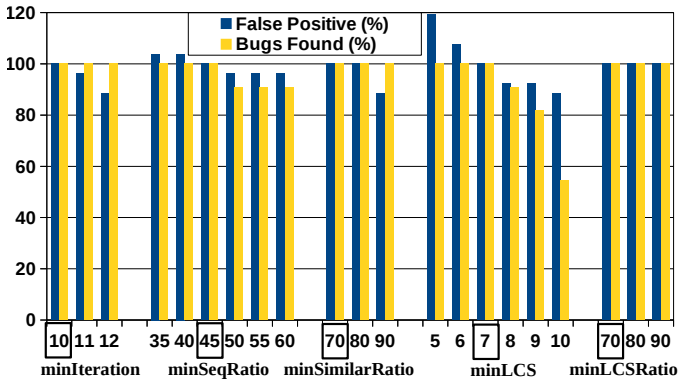

Fig. 12. Parameter-sensitivity experiments. Each configuration changes only one threshold, with its value shown on the X-axis. The default values are boxed. Two sets of experiments are conducted for each configuration: the left/dark bar shows false positives on JUnit tests, and the right/light bar shows bugs found on bug-triggering inputs. The Y-axis shows the numbers normalized to the results under default setting.

number of bugs found (light/yellow bars) and false positives (dark/blue bars) normalized to the values for the default configuration. For bugs found, higher is better, and for false positives, lower is better.

Impact on False Negatives: We increased the threshold value for each parameter, and for only two of them such increase has caused false negatives. The most sensitive is `minLCS`, which measures the absolute length of the longest common substring between two consecutive IPCS-sequences for an instruction. When `minLCS` increases from the default 7 to 10, the number of bugs found steadily decreases from 11 to 6. The longest common substring is usually shorter than the total number of inner-loop iterations, which is often determined by the input scale. Therefore, when the input scale is small, a high `minLCS` setting could miss many bugs.

The other parameter whose increase caused false negatives is `minSeqRatio`, which measures the ratio of loop iterations that have executed the particular memory-read instruction. The inner loop of bug #7.1 is buried inside an `if` statement that is executed by about half of the outer-loop iterations. As a result, this bug is missed once `minSeqRatio` gets over 50%. We believe that this type of if/then/else situation is common enough to have the default value under 50%. Note that, except for this type of bugs, `minSeqRatio` can be increased to 60% and beyond without losing any bugs.

Impact on False Positives: For the two parameters that caused false negatives above, `minLCS` and `minSeqRatio`, we both increased and decreased the threshold values. For the other three parameters, we only increased the values. We can see that increasing `minIter` and `minSimRatio` over the default values decreases the number of false positives by about 12% *without loosing any bugs*. In practice, one may want to indeed increase these parameters, but we chose conservative parameter values. In contrast, `minLCSRatio` is the least sensitive: increasing it from 70% to 90% changes neither false positives nor false negatives.

Choosing the Threshold Values: As seen from the discussion above, TODDLER can work well in a large range of threshold values. Note that *we did not choose the default threshold values for* TODDLER *to obtain the best results for false positives and false negatives*. For example, we could increase `minIter` and `minSimRatio` to get fewer false positives without missing any bug. Rather, we chose the default values based on our intuition about the values that could give reasonable results. Moreover, we settled on these values *before* running TODDLER on performance tests (Section IV-B).

E. TODDLER *for C/C++ Code*

The performance bugs that TODDLER finds do not exist only in Java code; as already mentioned in Section II, such bugs also exist in C/C++ code. To further evaluate our technique, we implemented a prototype TODDLER tool for C/C++ code. Our prototype uses Pin [19] to automatically instrument memory reads but currently does not automatically instrument loops; we manually added loop events for six real-world bugs (three from GCC, two from Mozilla, and one from MySQL). The prototype logs values read and loop events, and computes similarity *offline* by processing these logs using Python. The results show that this prototype can find all these six bugs. Because we do not instrument all the loops, we cannot measure false positives for this prototype.

V. DISCUSSION

Loop Nesting: TODDLER misses bugs that are not in nested loops. We intentionally focused on nested loops, because they create more severe performance hits. However, non-nested loops can also be slow, e.g., loops that contain I/O. TODDLER can be extended to look for bugs in such loops by *modeling* the native, I/O methods in Java [20] to make their loops explicit.

Other Performance Bugs: TODDLER misses several categories of performance bugs, including (1) performance bugs specific to multi-threaded code such as lock contention [21], load imbalance [22], or false sharing [23], (2) bugs related to idle time [24], and (3) object bloat [25]. TODDLER finds performance bugs involving loops, which the existing techniques miss, so TODDLER complements these techniques.

Dynamic Technique: Just like profilers, TODDLER requires a test input. Fortunately, developers already write some performance benchmarks but typically measure only real time and look for regressions. TODDLER provides an oracle to identify performance bugs and encourages developers to write performance tests. As our evaluation shows, performance tests are relatively easy to write manually even by developers who are not familiar with performance testing, and one can sometimes even use automated test-generation techniques for performance tests. Future work can focus on developing specialized test-generation techniques for performance bugs.

Similarity Measures: Because the longest common substring worked quite well for comparing similarity of IPCS-sequences, we did not evaluate any other approach. Future work could, for example, use edit distance to compare IPCS-sequences or, even further, capture the memory accesses not as IPCS-sequences of values but as execution trees that encode loop iterations and then measure tree similarity.

VI. Related Work

Profiling, Visualization, and Computational Complexity: Profiling and performance-visualization tools are critical for developers to understand the performance features of different software components. A lot of recent progress was made to provide more accurate and efficient profiling [8], [10], [26]–[32]. However, as discussed in Section I, profilers have fundamental limitations in detecting performance bugs. Several tools estimate the worst-time computational complexity of code [33]–[35], but like profilers, these tools report that some computation takes time, not if it wastes time. TODDLER complements these techniques to find performance bugs.

Performance-Bug Detection: Several techniques detect the excessive use of temporary objects, a common performance problem in object-oriented software [36], [37]. Xu et al. use a run-time analysis to detect low-utility data structures where the effort to construct member fields outweighs the usage of these fields [25]. Jin et al. study efficiency rules in performance-bug patches and detect performance bugs that are similar with previously patched ones [9]. Other techniques detect performance problems caused by idle time [24], multi-thread false sharing [23], or error recovery in distributed systems [38]. The success of these tools demonstrates the potential of performance-bug detection, but the existing work only covers a small portion of real-world performance bugs. TODDLER focuses on performance bugs caused by inefficient or redundant computation across nested loops. Many of these bugs, such as the real-world example bugs discussed in the paper, cannot be detected by the existing performance bug detectors. Therefore, TODDLER well complements these techniques.

VII. Conclusions

Performance testing would greatly benefit from automated oracles for performance bugs. We presented TODDLER, a novel oracle that detects performance bugs by identifying repetitive memory read sequences across loop iterations. TODDLER found 42 new bugs in 6 popular codebases: Ant, Google Core Libraries, JUnit, Apache Collections, JDK, and JFreeChart. So far developers have already fixed 10 of these bugs and confirmed 6 more as real bugs. We also evaluated TODDLER with 11 previously known, real-world performance bugs, and the experiments show TODDLER can effectively detect performance bugs with a much higher accuracy than profiling tools. TODDLER can help expose more performance bugs before software release by discovering problems even before they manifest as slow computation found by profilers. While these results are highly promising, TODDLER is just a starting point in addressing loop-related performance bugs.

Acknowledgments

We thank Caius Brindescu, Mihai Codoban, Hassan Eslami, Lyle Franklin, Mert Guldur, Alex Győri, Stas Negara, Francesco Sorrentino, and Loránd Szakács for help with experiments. This material is based upon work partially supported by NSF under Grant Nos. CCF-1054616, CNS-0958199, and CCF-0746856; and a Clare Boothe Luce faculty fellowship.

References

[1] Bugzilla@Mozilla, "Bugzilla keyword descriptions," https://bugzilla.mozilla.org/describekeywords.cgi.
[2] P. Kallender, "Trend Micro will pay for PC repair costs," 2005, http://www.pcworld.com/article/120612/article.html.
[3] G. E. Morris, "Lessons from the Colorado benefits management system disaster," 2004, www.ad-mkt-review.com/public_html/air/ai200411.html.
[4] T. Richardson, "1901 census site still down after six months," 2002, http://www.theregister.co.uk/2002/07/03/1901_census_site_still_down/.
[5] D. Mituzas, "Embarrassment," 2009, http://dom.as/2009/06/26/embarrassment/.
[6] I. Molyneaux, *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, 2009.
[7] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*. Addison-Wesley, 2010.
[8] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *ICSE*, 2012.
[9] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, 2012.
[10] M. Jovic, A. Adamoli, and M. Hauswirth, "Catch me if you can: Performance bug detection in the wild," in *OOPSLA*, 2011.
[11] S. Zaman, B. Adams, and A. E. Hassan, "A qualitative study on performance bugs," in *MSR*, 2012.
[12] M. D. Bond and K. S. McKinley, "Probabilistic calling context," in *OOPSLA*, 2007.
[13] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *ICSE*, 2003.
[14] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *ISSTA*, 2009.
[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.
[16] Soot, "Soot: A Java optimization framework," http://www.sable.mcgill.ca/soot/.
[17] Sun Microsystems, "HPROF JVM profiler," http://java.sun.com/developer/technicalArticles/Programming/HPROF.html.
[18] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov, "Testing container classes: Random or systematic?" in *FASE*, 2011.
[19] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[20] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," *ASE-J*, 2003.
[21] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, "Analyzing lock contention in multithreaded applications," in *PPOPP*, 2010.
[22] J. Oh, C. J. Hughes, G. Venkataramani, and M. Prvulovic, "LIME: A framework for debugging load imbalance in multi-threaded execution," in *ICSE*, 2011.
[23] T. Liu and E. D. Berger, "Precise detection and automatic mitigation of false sharing," in *OOPSLA*, 2011.
[24] E. Altman, M. Arnold, S. Fink, and N. Mitchell, "Performance analysis of idle programs," in *OOPSLA*, 2010.
[25] G. H. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky, "Finding low-utility data structures," in *PLDI*, 2010.
[26] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in *ISCA*, 2011.
[27] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of Java profilers," in *PLDI*, 2010.
[28] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. C. Mozer, "Automating vertical profiling," in *OOPSLA*, 2005.
[29] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-sensitive profiling," in *PLDI*, 2012.
[30] D. Zaparanuks and M. Hauswirth, "Algorithmic profiling," in *PLDI*, 2012.
[31] A. Diwan, M. Hauswirth, T. Mytkowicz, and P. F. Sweeney, "TraceAnalyzer: A system for processing performance traces," *Softw. Pract. Exper.*, March 2011.
[32] D. C. D'Elia, C. Demetrescu, and I. Finocchi, "Mining hot calling contexts in small space," in *PLDI*, 2011.
[33] J. Burnim, S. Juvekar, and K. Sen, "WISE: Automated test generation for worst-case complexity," in *ICSE*, 2009.
[34] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *USENIX Security Symposium*, 2003.
[35] S. Gulwani, K. K. Mehra, and T. M. Chilimbi, "SPEED: Precise and efficient static estimation of program computational complexity," in *POPL*, 2009.
[36] G. H. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky, "Go with the flow: Profiling copies to find runtime bloat," in *PLDI*, 2009.
[37] B. Dufour, B. G. Ryder, and G. Sevitsky, "A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications," in *FSE*, 2008.
[38] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala, "Finding latent performance bugs in systems implementations," in *FSE*, 2010.