CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes

Adrian Nistor[1], Po-Chun Chang[2], Cosmin Radoi[3], Shan Lu[4]
[1]Chapman University, [2]University of Wisconsin–Madison, [3]University of Illinois, Urbana-Champaign, [4]University of Chicago
[1]anistor@chapman.edu, [2]pchang9@cs.wisc.edu, [3]cos@illinois.edu, [4]shanlu@cs.uchicago.edu

***Abstract*—Performance bugs are programming errors that slow down program execution. While existing techniques can detect various types of performance bugs, a crucial and practical aspect of performance bugs has not received the attention it deserves: how likely are developers to fix a performance bug? In practice, fixing a performance bug can have both benefits and drawbacks, and developers fix a performance bug only when the benefits outweigh the drawbacks. Unfortunately, for many performance bugs, the benefits and drawbacks are difficult to assess accurately.**

This paper presents CARAMEL, a novel static technique that detects and fixes performance bugs that have non-intrusive fixes likely to be adopted by developers. Each performance bug detected by CARAMEL is associated with a loop and a condition. When the condition becomes true during the loop execution, all the remaining computation performed by the loop is wasted. Developers typically fix such performance bugs because these bugs waste computation in loops and have non-intrusive fixes: when some condition becomes true dynamically, just break out of the loop. Given a program, CARAMEL detects such bugs statically and gives developers a potential source-level fix for each bug. We evaluate CARAMEL on *real-world applications*, including 11 popular Java applications (e.g., Groovy, Log4J, Lucene, Struts, Tomcat, etc) and 4 widely used C/C++ applications (Chromium, GCC, Mozilla, and MySQL). CARAMEL finds *61 new performance bugs* in the Java applications and *89 new performance bugs* in the C/C++ applications. Based on our bug reports, *developers so far have fixed 51 and 65 performance bugs* in the Java and C/C++ applications, respectively. Most of the remaining bugs are still under consideration by developers.

I. INTRODUCTION

Software performance is critical for the success of a software project. Performance bugs[1] are programming errors that slow down program execution [7]. Performance bugs create poor user experience, affect the user-perceived software quality, degrade application responsiveness, waste computational resources, and lower system throughput [6], [39]. Even expert programmers can introduce performance bugs, which have already caused serious and highly publicized incidents [24], [25], [38]. Well tested commercial products such as Internet Explorer, Microsoft SQLServer, Visual Studio, and Acrobat Reader are also affected by performance bugs [2], [37].

Several techniques [5], [11], [18], [20]–[22], [26], [43], [46], [47], [50], [56], [58], [60]–[62] have been proposed to help detect various types of performance bugs. However, there are still many performance bugs that cannot be detected by existing techniques. Furthermore, a crucial and practical aspect of performance bugs has not received the attention it deserves: *how likely are developers **to fix** a detected performance bug?*

In practice, when developers decide if they should fix a performance bug, developers face a difficult choice between the potential drawbacks and the potential benefits of the fix.

On one hand, similar to fixing functional bugs, fixing performance bugs can have drawbacks. First, fixing performance bugs may introduce severe functional bugs, which may lead to program crash or data loss. The risk of having such negative and noticeable effects can make developers very cautious about improving performance. Second, fixing performance bugs may break good software engineering practices, making the code difficult to read, maintain, and evolve. For example, fixing performance bugs may require breaking encapsulation, code cloning, or specialization. Third, fixing performance bugs takes time and effort, especially if the fix involves several software modules or requires a complex implementation. Fourth, fixing performance bugs, which manifest for some inputs, may slow down other code, for some other inputs, and developers must decide which of these slowdowns—the slowdown caused by the performance bug for some inputs, or the slowdown caused by the fix for some other inputs—is preferable.

On the other hand, fixing performance bugs has benefits, i.e., it speeds up code. However, unlike fixing functional bugs, the benefits of fixing a performance bug *are often difficult to assess accurately*, especially when the fix is complex. First, the speedup offered by the fix depends on the input, and many inputs may not be sped up at all, because performance bugs manifest only for certain inputs. Therefore, developers need to estimate which inputs have which speedups, and how frequent or important are these inputs in practice. Second, the exact speedup offered by the fix for an input is difficult to estimate without executing the code, and speedups of orders of magnitude—i.e., speedups for which accurate estimates are not necessary—are rare. Unfortunately, developers often have access to only a few real-world inputs triggering a bug—or none at all, if the bug was detected during development using benchmarks, static tools, or code inspection—, and can find it difficult to estimate the expected speedup for the rest of the real-world inputs that may trigger the bug.

In practice, developers fix performance bugs when the benefits outweigh the drawbacks. Specifically, developers are likely

[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" [7]. However, others believe "bug" requires specifications, and prefer to use the terms "performance problem" or "performance issue". We feel this is just a naming issue and use the Mozilla Bugzilla term of "performance bug".

to fix performance bugs that have simple and non-intrusive fixes. Such fixes are unlikely to introduce new functional bugs, do not increase code complexity and maintenance costs, are easy to understand and implement, and are unlikely to degrade performance for other inputs. In other words, the choice between benefits and drawbacks is made easy for developers: because the fixes are simple and non-intrusive, fixing the bugs brings only benefits.

This paper makes the following contributions:

Novel Perspective: Compared with previous work, this paper has a unique perspective towards detecting performance bugs: *we focus on detecting bugs that are very likely to be fixed by developers*. Following the above discussion, we propose to detect performance bugs whose fixes *clearly* offer more benefits than drawbacks to developers.

New Family of Performance Bugs: This paper identifies a family of performance bugs that developers are very likely to fix. Every bug in this family is associated with a loop[2] and a condition. When the condition becomes true during the loop execution, all the remaining computation performed by the loop is wasted. In the extreme case when the condition is true at the start of the loop execution, the entire loop computation is wasted. Developers typically fix performance bugs in this family because (1) these performance bugs waste computation in loops, and (2) these performance bugs have simple and non-intrusive fixes: when some condition becomes true, just break out of the loop. Typically, these bugs are fixed by adding one line of code inside the loop, i.e., `if (cond) break`, which we call a *CondBreak fix*. We call `cond` a *L-Break condition*.

Important Performance Bugs: The performance bugs in this family are important: developers of real-world applications typically fix these bugs. Developers are the ultimate arbiters for what is useful and what is not useful for their project, and developers typically think these bugs must be addressed. Furthermore, these bugs are not all the performance bugs that have non-intrusive fixes, and our work is a promising first step in this important research direction.

Technique: This paper proposes CARAMEL, a novel static technique for detecting performance bugs that have CondBreak fixes. CARAMEL takes as input a program and outputs loops that can be fixed by CondBreak fixes, together with a potential fix for each buggy loop. The fixes proposed by CARAMEL can be directly applied to source-code and are easy to read by developers. CARAMEL works on intermediate code representation and analyzes each loop in five steps. First, CARAMEL identifies the loop instructions that may produce results visible after the loop terminates. Second, for each such instruction, CARAMEL detects the condition under which the instruction can be skipped for the remainder of the loop without changing the program outcome. We call this condition an *I-Break condition*, similarly to the L-Break condition described earlier for the entire loop. Third, CARAMEL checks if all instructions from step two can be skipped *simultaneously* without changing the program outcome, i.e., if all I-Break conditions can be satisfied simultaneously. The conjunction of all I-Break conditions is the L-Break condition. Fourth, CARAMEL checks if the computation waste in the loop is not already avoided, i.e., if the loop does not already terminate when the L-Break condition is satisfied. If all the previous steps are successful, CARAMEL reports a performance bug. Fifth, CARAMEL generates a fix for the performance bug. The fix has the basic format `if (cond) break`, where `cond` is the L-Break condition CARAMEL computed in the third step.

Automatic Fix Generation: Automated bug fixing is challenging in general [30], but CARAMEL is able to automatically generate fixes for most bugs because CARAMEL takes advantage of two characteristics of the performance bugs it detects: (1) the bugs have CondBreak fixes, which can be inserted right at the start of the loop, thus avoiding complex interactions with other loop code, and (2) the L-Break condition in a CondBreak fix is a relatively simple boolean expression (Sections II, III).

Evaluation: Our main measure of success for CARAMEL is if real-world developers think the bugs found by CARAMEL are important, as quantified by the number of bugs fixed. We implement *two* CARAMEL tools, one for Java, CARAMEL-J, and one for C/C++, CARAMEL-C. We evaluate CARAMEL-J on 11 popular Java applications: Ant, Groovy, JMeter, Log4J, Lucene, PDFBox, Sling, Solr, Struts, Tika, and Tomcat. CARAMEL-J found *61 new real-world performance bugs* of which *51 bugs have already been fixed by developers*. We evaluate CARAMEL-C on 4 widely used C/C++ desktop and server applications: Chromium, GCC, Mozilla, and MySQL. CARAMEL-C found *89 new real-world performance bugs* of which *65 bugs have already been fixed by developers*. Of the bugs not yet fixed, 7 bugs are confirmed and still under consideration by developers and 16 bugs are still open. 7 bugs were not fixed because they are in deprecated code, old code, test code, or auxiliary projects. *Only 4 bugs* were not fixed because developers considered that the bugs have small performance impact or that the fixes make code more difficult to read. CARAMEL has few false positives, 19 for CARAMEL-J and 4 for CARAMEL-C. Out of 150 bugs, CARAMEL successfully generates fixes for 149 bugs.

II. WHAT PERFORMANCE BUGS HAVE CONDBREAK FIXES?

We discuss below two bug characteristics that help understand what performance bugs have CondBreak fixes. We call *result instruction (RI)* a loop instruction that *may* write to variables live and memory reachable after the loop. RIs are important in understanding performance bugs that have CondBreak fixes. For example, if all RIs in a loop do not (need to) execute under a certain condition, the entire loop, *including* all non-RIs, can be skipped.

(A) Where the Computation Is Wasted: A loop-related performance bug can waste computation either in an entire iteration or in parts of an iteration, in consecutive or arbitrary iterations, in iterations at the start, end, or middle of the loop. Such a bug can have a CondBreak fix (i.e., break out of the loop under a certain condition) if it wastes computation in the entire iteration in one of the following three locations:

[2]We focus on loops because most computation time is spent inside loops and most performance bugs involve loops [19], [23], [45], [56], [61].

(1) for every iteration in the loop, short as *Every*, (2) for every iteration at the end of the loop, short as *Late*, or (3) for every iteration at the start of the loop, short as *Early*. Bugs that waste computation in category Every can be fixed by breaking out of the loop if the L-Break condition is satisfied at the loop entrance, effectively skipping the entire loop. Bugs that waste computation in category Late can be fixed by breaking out of the loop once the computation waste starts. Bugs that waste computation in category Early can be fixed by iterating from the end of the loop and breaking out of the loop once the computation waste starts. Bugs that waste computation only in parts of an iteration or only in specific iterations cannot be fixed by CondBreak fixes and are not the focus of CARAMEL.

(B) How the Computation Is Wasted: In order for the computation of an entire iteration to be wasted under a certain condition, i.e., the L-Break condition, every RI in that iteration has to fall into one of following three cases: (1) the RI is not executed under the L-Break condition, short as *No-Result*, (2) the RI is executed, but, under the L-Break condition, its result does not change the values used by computation after the loop, short as *Useless-Result*, or (3) the RI is executed and its result changes the values used by computation after the loop, but, under the L-Break condition, this result does not affect the perceived outcome of the program, short as *Semantically-Useless-Result*. Identifying Semantically-Useless-Result RIs usually requires developers' expert knowledge, and the L-Break conditions are likely difficult to express in source-code. CARAMEL focuses on No-Result and Useless-Result RIs.

Based on the above discussion, a loop can have a performance bug fixed by a CondBreak fix if *all* RIs in the loop belong to one of the six types shown in Figure 1. Note that the three computation-waste locations in (A) effectively describe which instances of an RI can be skipped in the loop. We describe individual RIs of Types 1–4, i.e., the types CARAMEL focuses on, in Sections II-A–II-D, and we present how multiple RIs appear in the same bug in Section II-F. We briefly discuss Type X and Type Y in Section II-E.

	Every	Late	Early
No-Result	Type 1	Type 2	Type Y
Useless-Result	Type X	Type 3	Type 4

Fig. 1. Types of RIs

The bugs in the following examples are *previously unknown* real-world performance bugs found by CARAMEL. We reported them to developers and the developers fixed all of them.

A. Type 1 RIs

Figure 2 shows a performance bug from Groovy containing a Type 1 RI. Line 3 is the fix and it is *not* part of the original buggy code. The only RI in this loop is `return true` (line 7), which writes the method's return value and also causes the code after the loop to not execute. When `argTypes` is initialized to a non-empty array, the RI *cannot* execute throughout the loop, the entire loop computation is wasted, and the loop can just be skipped. The reason is that `argTypes` is never modified inside the loop. When `argTypes` is initialized to a non-empty array, both `argTypes == null` and `argTypes.length == 0` (line 5) are false throughout the loop, which makes `isZeroArg` false, which makes `match` (line 6) false, which in turn means the RI cannot execute.

```
1 Class[] argTypes = ...
2 for (Iterator i = methods.iterator(); i.hasNext();) {
3 + if (!(argTypes == null) && !(argTypes.length == 0)) break; // FIX
4   MethodNode mn = (MethodNode) i.next();
5   boolean isZeroArg = (argTypes == null || argTypes.length == 0);
6   boolean match =mn.getName().equals(methodName) && isZeroArg;
7   if (match) return true; // RI
8 }
```

Fig. 2. Type 1 RI in a Groovy performance bug

This RI is of Type 1 because, if the I-Break condition is true at the start of the loop, the RI is not executed (category No-Result) in any iteration of the entire loop (category Every). The I-Break condition for the RI is that both `argTypes == null` and `argTypes.length == 0` are false. The L-Break condition is the same as the I-Break condition because there is only one RI. The CondBreak fix is the code added in line 3 (the + at the start of line 3 means the line is added), i.e., the loop breaks when the L-Break condition is true. We discuss fixes equivalent to the CondBreak fix in Section III-E.

B. Type 2 RIs

Figure 3 shows a performance bug from PDFBox containing a Type 2 RI. There are three loops in this code and two RIs, as shown in the figure. RI 1 is an RI for all three loops because it writes `alreadyPresent`, which is live at the end of all three loops. Similarly, RI 2 is an RI for loops 1 and 2. The "..." in the figure replace some complicated control flow and method calls, which we skip for clarity. The "..." contain no RIs. In this section, we focus our discussion on RI 2 because RI 1 is of Type 3, which we will discuss in the next section. Loop 3 does not have a bug, which we will further explain in Section III-D. Loops 1 and 2 are both buggy, as explained next for loop 1. Similar reasoning applies for loop 2.

```
1  boolean alreadyPresent = false;
2  while (itActualEmbeddedProperties.hasNext()) { // Loop 1
3  + if (alreadyPresent) break; // FIX
4    ... // non-RIs
5        while (itNewValues.hasNext()) { // Loop 2
6          ... // non-RIs
7          while (itOldValues.hasNext() && !alreadyPresent) { // Loop 3
8            oldVal = (TextType) itOldValues.next();
9            if(oldVal.getStringValue().equals(newVal.getStringValue())){
10             alreadyPresent = true; // RI 1
11         }}
12         if (!alreadyPresent) {
13           embeddedProp.getContainer().addProperty(newVal);// RI 2
14 }}}}}
```

Fig. 3. Type 2 RI in a PDFBox performance bug

For the first few loop 1 iterations, `alreadyPresent` is false, the condition on line 12 evaluates to true, and RI 2 executes and performs useful computation. However, once `alreadyPresent` is set to true on line 10, the condition on line 12 remains false for the remainder of the loop, and all the remaining computation in the loop can just be skipped. The reason is that the entire loop cannot assign `alreadyPresent` to false. Consequently, once `alreadyPresent` becomes true on line 10, it remains true and disables the execution of RI 2 for the remainder of the loop.

RI 2 is of Type 2 because, once the I-Break condition becomes true, RI 2 is not executed (category No-Result) for

the remaining loop iterations (category Late). The I-Break condition for RI 2 is that `alreadyPresent` equals true. We will explain in the next section that the I-Break condition for RI 1 is also that `alreadyPresent` equals true. The L-Break condition is the conjunction of the two I-Break conditions, i.e., `alreadyPresent` equals true. The CondBreak fix is the code added in line 3, i.e., the loop breaks when the L-Break condition is true.

C. Type 3 RIs

Figure 4(a) shows a performance bug from Tomcat containing two Type 3 RIs, RI 1 and RI 2. Both RIs set variable `elExp` to true. Once either RI is executed, the remaining computation in the loop is unnecessary, at best setting `elExp` to true again.

```
1  boolean elExp = ...
2  while (nodes.hasNext()) {
3  + if (elExp) break; // FIX
4    ELNode node = nodes.next();
5    if (node instanceof ELNode.Root) {
6      if (((ELNode.Root) node).getType() == '$') {
7        elExp = true; // RI 1
8      } else if (checkDeferred && ((ELNode.Root) node).getType()=='#'
9                  && !pageInfo.isDeferredSyntaxAllowedAsLiteral() ) {
10       elExp = true; // RI 2
11 }}}
```

(a) A Type 3 RI in a Tomcat performance bug

```
1 valid &= child.validate(); // RI
```

(b) Type 3 RI in a Sling performance bug

Fig. 4. Type 3 RIs

RI 1 is of Type 3 because, once the I-Break condition becomes true, RI 1's results for the remaining iterations (category Late) do not change the values used by future computation (category Useless-Result). Similar reasoning applies for RI 2. The I-Break condition for RI 1 is that `elExp` equals true. RI 2 has the same I-Break condition. The L-Break condition is the conjunction of the two I-Break conditions, i.e., `elExp` equals true. The CondBreak fix is the code on line 3, i.e., the loop breaks when the L-Break condition is true.

Note that, for RI 1 and RI 2 to be of Type 3, `elExp` can have any type, not necessarily boolean, as long as `elExp` is assigned a constant. In fact, even if `elExp` is not assigned a constant, there is still an alternative way to set `elExp` to a value that does not change after some time, as shown in Figure 4(b). In Figure 4(b), once `valid` is set to false, the semantics of the `&=` operator ensures `valid` remains false. The I-Break condition is that `valid` equals false.

D. Type 4 RIs

Figure 5(a) shows a performance bug from JMeter containing a Type 4 RI (line 6). The variable `length` keeps getting overwritten by the RI. Consequently, all iterations before the last iteration that writes `length` are wasted. The reason is that computation after the loop will only see the last value written to `length`. This last value does not depend on previous iterations, except for the value of `idx`, which can be computed when iterating from the end of the loop.

The RI is of Type 4 because its results for early iterations (category Early) do not affect the values used by future computations (category Useless-Result). The I-Break condition for the RI is that `length` has been written in the loop when

```
1  int length = ...
2  for (int idx = 0; idx < headerSize; idx++) {
3  @ for (int idx = headerSize - 1; idx >= 0; idx--) { // FIX
4    Header hd = mngr.getHeader(idx);
5    if (HTTPConstants.HEADER.equalsIgnoreCase(hd.getName())) {
6      length = Integer.parseInt(hd.getValue()); // RI
7  +   break; // FIX
8  }}
```

(a) Bug and *Alternative* fix

```
1  int length = ...
2  + boolean wasExecuted = false; // FIX
3  for (int idx = 0; idx < headerSize; idx++) {
4  @ for (int idx = headerSize - 1; idx >= 0; idx--) { // FIX
5  + if (wasExecuted) break; // FIX
6    Header hd = mngr.getHeader(idx);
7    if (HTTPConstants.HEADER.equalsIgnoreCase(hd.getName())) {
8  +   if (!wasExecuted) { // FIX
9  +     wasExecuted = true; // FIX
10       length = Integer.parseInt(hd.getValue()); // RI
11 +   } // FIX
12 }}
```

(b) *CondBreak* fix

Fig. 5. Type 4 RI in a JMeter performance bug. The fix in 5(a) is an *Alternative* fix. The *CondBreak* fix is in 5(b).

iterating from the end of the loop. The L-Break condition is the same as the I-Break condition because there is only one RI. For clarity, the CondBreak fix is shown separately, in Figure 5(b). This fix looks complex because we want to make the L-Break condition (i.e., `wasExecuted` equals true) explicit in the code. Differently from the Type 3 RI in Figure 4(a), the RI in this example does not set `length` to a constant. Therefore, we have to create an extra variable `wasExecuted` (lines 2, 5, 8, 9, 11) to track whether `length` has been written. Figure 5(a) shows a simpler, *alternative* fix, that does not use `wasExecuted`. In the alternative fix, the reversed loop breaks the first time when the RI is executed (line7). The simpler, alternative fix comes at a price: it is correct only when the loop has one RI. Otherwise, breaking out of the loop after one RI would incorrectly miss the execution of remaining RIs.

E. Type X and Type Y RIs

A Type X instruction would be similar to RI 1 and RI 2 in Figure 4(a) if the value of `elExp` would be a constant true before the loop started. In practice, CARAMEL never found such RIs. A Type Y RI cannot write to the same memory locations in different loop iterations. Checking that all dynamic instances of the same static instruction can only write to disjoint memory locations requires complex static analysis, and CARAMEL does not perform such checks.

F. Bugs with Multiple RIs

A buggy loop can contain multiple RIs of the same or different types. The only constraint is that a Type 4 RI cannot co-exist with Type 2 or Type 3 RIs because the former requires the bug fix to skip iterations at the start of the loop and the latter requires the bug fix to skip iterations at the end of the loop. In practice, we did not encounter Type 1 RIs co-existing with other types of RIs. Some RIs can be of multiple types, as shown next for RI 3 and RI 9. The L-Break condition is the conjunction of all RIs' I-Break conditions.

Figure 6 shows an example bug with multiple RIs from PDFBox. Unlike the bugs in Figure 3 and Figure 4(a), this bug has nine RIs that have different I-Break conditions. RI 1 and

RI 2 are Type 2 RIs because once `annotNotFound` is set to false (line 9), `annotNotFound` *cannot become true again* and the condition on line 6 evaluates to false in the remaining loop iterations. Similar reasoning applies for RI 4–8, `sigFieldNotFound` (line 14), and the condition on line 11. RI 3 and RI 9 are *simultaneously* Type 2 (similarly to RI 1–2 and RI 4–8, respectively) and Type 3. The I-Break conditions for RI 1–3 and RI 4–9 are `annotNotFound` equals false and `sigFieldNotFound` equals false, respectively. The L-Break condition is the conjunction of all nine I-Break conditions, i.e., both `annotNotFound` and `sigFieldNotFound` equal false.

```
1  boolean annotNotFound = ...
2  boolean sigFieldNotFound = ...
3  for ( COSObject cosObject : cosObjects ) {
4  + if (!annotNotFound && !sigFieldNotFound) break; // FIX
5    ... // some non−RIs
6    if (annotNotFound && COSName.ANNOT.equals(type)) {
7      ... // RI 1 and some non-RIs
8      signatureField.getWidget().setRectangle(rect); // RI 2
9      annotNotFound = false; // RI 3
10   }
11   if (sigFieldNotFound && COSName.SIG.equals(ft)&&apDict!=null){
12     ... // RI 4, RI 5, RI 6, RI 7 and some non-RIs
13     acroFormDict.setItem(COSName.DR, dr); // RI 8
14     sigFieldNotFound=false; // RI 9
15 }}
```

Fig. 6. Multiple RIs in a PDFBox performance bug

III. Detecting and Fixing Performance Bugs That Have CondBreak Fixes

We next present the high-level CARAMEL algorithm (Section III-A) and the algorithm steps (Sections III-B–III-E).

A. High-Level Algorithm

Figure 7 shows the high-level algorithm for CARAMEL. CARAMEL is a static technique that works on intermediate code representation (IR). CARAMEL receives as input the loop to analyze and various information to help the static analysis, e.g., the control flow graph for the method containing the loop, pointer aliasing information, and a call graph.

```
1  void detectPerformanceBug(Loop l, Method m, AliasInfo alias) {
2    Set⟨Instruction⟩ allRIs = getRIs(l, m, alias);
3    Set⟨Condition⟩ allCond = new Set⟨Condition⟩();
4    for (Instruction r : allRIs) {
5      Condition one = typeOne(r, l, m, alias);
6      Condition two = typeTwo(r, l, m, alias);
7      Condition three = typeThree(r, l, m, alias);
8      Condition four = typeFour(r, l, m, alias, allRIs.size());
9      if(one.false()&&two.false()&&three.false()&&four.false()) return;
10     allCond.putIfNotFalse(one, two, three, four);
11   }
12   if(satisfiedTogether(allCond) && notAlreadyAvoided(allCond, l)) {
13     String fix = generateFix(allCond, l, m);
14     reportBugAndFix(fix, allRIs, allCond);
15 }}
```

Fig. 7. CARAMEL high-level algorithm

CARAMEL works in five steps. First, CARAMEL computes the loop RIs using routine static analysis (line 2). Second, for each RI `r`, CARAMEL checks if `r` belongs to one of the four types presented in Section II and computes `r`'s I-Break condition accordingly (lines 4–10). If `r` does not belong to any of the four types, all the conditions computed on lines 5–8 are false and therefore the loop does not have a bug (line 9). If `r` is of one of the four types, CARAMEL saves for further use `r`'s I-Break condition (line 10). Third, CARAMEL checks if all RIs can be skipped *simultaneously* without changing the program outcome, i.e., if the I-Break conditions for individual RIs can be satisfied simultaneously (line 12). The conjunction of the I-Break conditions is the L-Break condition. Fourth, CARAMEL checks if the computation waste in the loop is not already avoided, i.e., if the loop does not already terminate when the L-Break condition is satisfied (line 12). Fifth, using the L-Break condition, CARAMEL generates a fix (line 13) and reports the bug (line 14). The bug report contains the fix, and, for each RI, the RI type and I-Break condition.

The above algorithm enables CARAMEL to detect and fix performance bugs that involve multiple RIs, either of the same or different types, similar to the bugs in Figures 3, 4(a), and 6. This is because, after step two, CARAMEL works only with a collection of conditions, and CARAMEL is not concerned with how these conditions were obtained in step two.

Preliminary: Boolean Expressions: To compute the I-Break conditions and the L-Break condition, CARAMEL reasons about boolean expressions. CARAMEL represents and reasons about a boolean expression as one or multiple *Atoms* connected by boolean operators (`NOT`, `AND`, `OR`). An Atom refers to either a boolean variable or a boolean expression containing non-boolean operators. Atoms do not contain other Atoms. For example, an Atom could be a method call returning a boolean value or a comparison between two integers. To keep complexity low and scale, CARAMEL does not reason about operations inside Atoms. An Atom can be either true or false but *not* both simultaneously.

For space limitations, we do not go into the details of how CARAMEL works with boolean expressions, and we give only a high-level overview for two techniques used by CARAMEL. These two techniques can be substituted by more sophisticated techniques, such as symbolic execution. However, for CARAMEL's purposes, these two techniques offer good results at considerably reduced complexity. Technique **T-PathExec** computes the execution condition of a loop instruction as the disjunction of all path constraints that correspond to the acyclic execution paths leading from the loop header to the instruction. A path constraint is the conjunction of all branch conditions, represented by Atoms and negated when necessary, along a path. CARAMEL uses T-PathExec in steps two and four of the CARAMEL algorithm. Technique **T-Instantiation** computes, for a boolean expression `E` (in DNF form) and some Atoms `set`, the values of the `set` Atoms for which `E` is guaranteed false or true. Conversely, T-Instantiation determines if `E` *may* be true or false, irrespective of Atoms in `set`. T-Instantiation tries all possible combinations of values for the `set` Atoms and uses logic rules such as "False `AND` Unknown equals False" to determine the value of `E`. For example, for `E = $Atom1 AND $Atom2` and `set = {$Atom1}`, T-Instantiation determines that, when `$Atom1` equals true, `E` is Unknown, and, when `$Atom1` equals false, `E` is false. In the usage context of CARAMEL, `set` has few Atoms (e.g., when identifying Type 1 RIs, `set` contains the loop-invariant Atoms in `E`, which is rarely more than 3), and therefore T-Instantiation rarely tries more than 8 combinations. CARAMEL uses T-Instantiation in steps two, three, and four of the CARAMEL algorithm.

B. Detecting the Four RI Types

In the second step of the CARAMEL algorithm, CARAMEL determines if a given RI (all RIs are known from step one) belongs to one of the four types and, if so, the RI's I-Break condition. We describe here each type and the algorithm CARAMEL uses to detect it. Identifying all RIs that belong to each type would require complicated and non-scalable analysis. At the same time, not all RIs that belong to each type are common and have I-Break conditions that are easy to express in source-code. CARAMEL focuses on RIs whose type can be identified using scalable analysis and whose I-Break conditions are easy to express in source-code. CARAMEL can miss some RIs that belong to these four types, as explained below.

Type 1: An RI `r` is of Type 1 if there exists a condition `C` such that `r` cannot execute throughout the loop if `C` is true when the loop starts. The I-Break condition for `r` is `C`. To judge whether `r` belongs to Type 1, CARAMEL searches for `r`'s I-Break condition. **Theoretically**, the I-Break condition could be composed of any variables and expressions that appear or do not appear in the entire program. However, inferring or searching for such generic I-Break conditions is difficult. **In practice**, CARAMEL considers only I-Break conditions that (1) can be computed by analyzing the potential execution paths that may reach `r` from the loop header and (2) are composed of Atoms that can be proved to be loop-invariant based on control and data-flow analysis. Constraint (1) makes detecting candidate I-Break conditions feasible and scalable, while constraint (2) makes it easy to prove that a candidate I-Break condition cannot change its value throughout the loop execution. Additionally, constraint (1) ensures the I-Break condition is easy to express in source-code, because the candidate I-Break conditions contain only variables and expressions already present in the loop.

The CARAMEL algorithm uses T-PathExec to compute the execution condition `rExecCond` for `r`, gets the loop-invariant Atoms in `rExecCond`, uses T-Instantiation to get these Atoms' values for which `rExecCond` is guaranteed to be false irrespective of the values of other Atoms in `rExecCond`, and constructs `r`'s I-Break condition based on these Atom values.

Type 2: An RI `r` is of Type 2 if there exists a condition `C` such that `r` cannot execute once `C` becomes true during the loop execution. The I-Break condition for `r` is `C`. Detecting Type 2 and Type 1 RIs are similar and have similar challenges. CARAMEL applies similar constraints when searching for `r`'s I-Break condition, with only one difference for constraint (2). For Type 2 checking, CARAMEL only considers Atoms that are assigned one constant in the loop. This constraint makes it easy to prove that a candidate I-Break condition cannot change its value after all its component Atoms are updated in the loop.

The CARAMEL algorithm for detecting Type 2 RIs is similar to the algorithm for detecting Type 1 RIs, with two modifications. First, instead of identifying loop-invariant Atoms in `rExecCond`, the Type 2 algorithm detects Atoms in `rExecCond` that are assigned only one boolean constant value in the loop. Second, when CARAMEL computes the Atoms' values for which `rExecCond` is guaranteed to be false, CARAMEL takes into account that the Atoms identified above can take only the corresponding constant values.

Type 3: An RI `r` is of Type 3 if, after a certain loop iteration, `r` can only write to the same output locations it wrote in previous iterations and the values written are identical to the existing values in these locations; we call these existing values `S`. The I-Break condition is that the output locations contain `S`. To judge whether `r` belongs to Type 3, CARAMEL examines `r`'s output locations and output values. **Theoretically**, `r` could be any instruction, including a call to a method with complex control flow that writes to many memory locations. Reasoning about such a general `r` is difficult. **In practice**, CARAMEL focuses only on RIs that (1) have a single output location, (2) either write a constant (similar to Figure 4(a)) or perform the `&=` or `|=` operations (similar to Figure 4(b)), which effectively correspond to `S` being constants false or true, respectively, and (3) have an output location that is not written to in the loop with other values except `S`. Constraint (1) makes it easy to detect `r` does not change its output locations, while constraints (2) and (3) make it easy to prove that, after a certain loop iteration, `r` can only write `S`. Additionally, constraint (2) ensures the I-Break condition is easy to express in source-code, because `S` can be identified statically.

The CARAMEL algorithm uses straightforward static analysis to implement the above checks.

Type 4: An RI `r` is of Type 4 if `r` outputs values independent of computation in early iterations, except for the loop index computation, and if `r` cannot change its output locations. The I-Break condition is that the values in the output locations have been updated the first time when iterating from the end of the loop. To judge whether `r` belongs to Type 4, CARAMEL examines `r`'s output locations and output values. **Theoretically**, `r` could be any instruction, including a method call, and checking the above conditions for such a general `r` is difficult. **In practice**, CARAMEL focuses only on RIs that (1) have a single output location, (2) appear in loops that have no cross-iteration data dependency, except for the loop index computation, and (3) appear in loops that have only one RI. Constraint (1) makes it easy to detect that `r` does not change its output locations and constraint (2) makes it easy to prove that the output values are independent of computation in earlier iterations. Additionally, constraint (3) ensures the fix is similar to the alternative fix in Figure 5(a), instead of the CondBreak fix in Figure 5(b).

The CARAMEL algorithm checks if the loop has one RI and if no instruction in the loop body writes to memory or variables live between iterations, except for the loop index.

C. Checking Whether RIs can be Skipped Simultaneously

In the third step of the CARAMEL algorithm, CARAMEL checks if a scenario exists for which all RIs can be *simultaneously* skipped without changing the program outcome, i.e., all RIs' I-Break conditions can be satisfied simultaneously. The L-Break condition enabling this scenario is the conjunction of all I-Break conditions.

Figure 8 gives a simplified example from Lucene of why CARAMEL performs this check. The two RIs in this loop are of Type 1 and have I-Break conditions `roundNum < 0` equals true and `roundNum < 0` equals false, respectively. However, this loop does not contain a performance bug because the executions of the two RIs cannot be skipped *simultaneously*, i.e., `roundNum < 0` cannot simultaneously be true and false.

```
1 int roundNum = ...
2 StringBuilder sb = ...;
3 for (final String name : colForValByRound.keySet()) {
4   if (roundNum < 0) {
5     sb.append(Format.formatPaddLeft("-", template)); // RI 1
6   } else {
7     sb.append(Format.format(ai[n], template)); // RI 2
8 }}
```

Fig. 8. Simplified code from Lucene. The RIs are of Type 1, but these RIs do not create a performance bug.

To perform this check, CARAMEL applies T-Instantiation on the conjunction of selected I-Break conditions, effectively checking there exists at least one combination of the involved Atoms' values that makes the conjunction true. CARAMEL optimizes this check by applying it on selected, instead of all I-Break conditions, because the definitions of some RI types already guarantee that their corresponding I-Break conditions will never conflict with each other. For example, the I-Break condition for a Type 3 RI cannot conflict with the I-Break condition for a Type 1 RI. The reason is that the Atoms in the I-Break condition of a Type 1 RI must be loop-invariant and therefore cannot appear in a Type 3 RI's I-Break condition.

D. Checking the Computation Waste is Not Already Avoided

In the fourth step of the CARAMEL algorithm, CARAMEL checks the execution is not already exiting the loop when the L-Break condition is true. Loop 3 in Figure 3 (lines 7–11) is an example of why CARAMEL performs this check. As mentioned in Section II-B, loop 3 does not have a performance bug and we now explain why. The only RI in loop 3 is RI 1 (line 10), which is of Type 3 and therefore it may seem loop 3 performs useless computation once `alreadyPresent` is set to true. However, once `alreadyPresent` becomes true, the loop exits (`!alreadyPresent`, line 7), and therefore the loop does not have a performance bug. CARAMEL performs this check using T-PathExec, which is used to detect the execution condition for loop exits, and T-Instantiation, which is used to detect if the paths to loop exits may be taken in the original code when the L-Break condition is true.

E. Automatic Fix Generation

In the fifth step of the CARAMEL algorithm, CARAMEL generates source-code fixes. Automatic bug fixing is a difficult problem in general, but it is feasible for CARAMEL because CARAMEL focuses on bugs that have CondBreak fixes.

CARAMEL generates fixes in two steps. First, CARAMEL generates a source-code level L-Break condition composed of source-code level variables declared and initialized outside of the loop. Because the variable and method names are available at the IR level, this process is straightforward in general. The only challenge is that some Atoms in the L-Break condition may involve variables that are not suitable for the final source-code fix. Specifically, some variables are introduced by compiler in the IR representation and do not exist in the source-code, while some other variables are declared or initialized inside the loop by developers and therefore cannot be used in the fix at the start of the loop. The solution is straightforward: CARAMEL repetitively replaces these unsuitable variables with their assigned expression. This step guarantees not to change the value of the L-Break condition because of the way the I-Break conditions are defined in Section III-B.

Second, CARAMEL formats the fix according to the types of the RIs, and computes the line number where the fix is to be inserted using line number information from the intermediate source code representation. When the loop contains RIs of Type 1, 2, or 3, the fix simply inserts `if (L-Break condition) break` after the loop header, as shown in the Section II examples. In the special case when the loop contains *only* Type 1 RIs, the fix is `if (L-Break condition == false) theLoop`, effectively executing the loop only when the L-Break condition is false. This alternative fix is equivalent with the CondBreak fix, but is preferred by developers. When the loop contains a Type 4 RI, CARAMEL reverses the loop if the loop has an integer index variable that is incremented by one in the loop header, similar to that in Figure 5(a); otherwise, CARAMEL reports fix generation failure. General loop reversal is difficult to do automatically, but treating the above case was enough to fix the bugs we encountered in practice. CARAMEL can handle more cases for loop reversal in the future.

F. False Positives, False Negatives, and Incorrect Fixes

CARAMEL can have false positives, false negatives, and can generate incorrect fixes, though in practice these issues were not significant (Section V). These issues are typically created by unsoundness or incompleteness in the underlying static analysis framework. We discuss sources of false positive in Section V-B. CARAMEL can have false negatives because, due to unsoundness in the static analysis, CARAMEL can label non-RI instructions as RI, and the spurious RIs can make unnecessary loop computation look useful. In practice this was not a major problem, as CARAMEL found 150 new bugs in 15 widely used Java and C/C++ applications. Theoretically, CARAMEL can generate an incorrect fix for a real bug because, due to incompleteness in the static analysis (Section V-B), CARAMEL may not detect some RIs. If this happens, the L-Break condition does not contain all the I-Break conditions and the fix causes the execution to exit the loop too early. In practice CARAMEL *did not* generate any incorrect fix.

IV. TWO IMPLEMENTATIONS

We implement *two* CARAMEL tools, for both Java and C/C++ programs, which we call CARAMEL-J and CARAMEL-C, respectively. We implement CARAMEL-J and CARAMEL-C using WALA [1] and LLVM [29] static analysis frameworks, respectively. Implementing the high-level algorithms in Section III-B takes into account the fact that the IRs provided by WALA and LLVM are in SSA form. CARAMEL-J uses the pointer aliasing information provided by WALA.

CARAMEL-C conservatively assumes that every write to heap is an RI. The analysis in both CARAMEL-J and CARAMEL-C is inter-procedural. The implementation closely follows the presentation in the previous section. The only exception is that CARAMEL-C currently detects only bugs that have one RI, and therefore CARAMEL-C does not perform step three in the CARAMEL algorithm. We do not discuss further implementation details due to space limitations.

V. EVALUATION

We evaluate CARAMEL on *real-world applications* from Java and C/C++ using our two CARAMEL implementations, CARAMEL-J and CARAMEL-C, respectively. We use 11 popular Java applications (Ant, Groovy, JMeter, Log4J, Lucene, PDFBox, Sling, Solr, Struts, Tika, and Tomcat) and 4 widely used C/C++ desktop and server applications (Chromium, GCC, Mozilla, and MySQL). Figure 9 gives a short description of these applications. We analyze the latest code versions of these applications, except for Lucene, for which we use a slightly older version, because CARAMEL does not support Java 7. Of all the Lucene bugs found by CARAMEL, only two bugs are in code that no longer exists in the latest version. In total, CARAMEL generates 173 bug reports. This section first presents the 150 new bugs found by CARAMEL. It then discusses the 23 false positives reported by CARAMEL, the fix generation results, and CARAMEL's running time. We conduct the experiments on two Intel i7, 4-core, 8 GB machines, running at 2.5 GHz and 3.4 GHz for the Java and C/C++ experiments, respectively.

L	#	App	Description	LoC	Classes(J) Files(C)
Java	1	Ant	build tool	140,674	1,298
	2	Groovy	dynamic language	161,487	9,582
	3	JMeter	load testing tool	114,645	1,189
	4	Log4J	logging framework	51,936	1,420
	5	Lucene	text search engine	441,649	5,814
	6	PDFBox	PDF framework	108,796	1,081
	7	Sling	web app. framework	202,171	2,268
	8	Solr	search server	176,937	2,304
	9	Struts	web app. framework	175,026	2,752
	10	Tika	content extraction	50,503	717
	11	Tomcat	web server	295,223	2,473
C/C++	12	Chromium	web browser	13,371,208	10,951
	13	GCC	compiler	1,445,425	781
	14	Mozilla	web browser	5,893,397	5,725
	15	MySQL	database server	1,774,926	1,684

Fig. 9. The applications used in experiments

A. New Bugs Found by CARAMEL

CARAMEL is very effective at detecting performance bugs. CARAMEL finds a *total of 150 new bugs*, 61 bugs in Java applications and 89 bugs C/C++ applications. Of these, *116 bugs*, 51 and 65 in Java and C/C++, respectively, *have already been fixed by developers*. Of the bugs not yet fixed, 7 bugs are confirmed and still under consideration by developers and 16 bugs are still open. 7 bugs were not fixed because they are in deprecated code, old code, test code, or auxiliary projects. *Only 3 bugs* were not fixed because developers considered that the bugs do not have a significant performance impact. *Only 1 bug* was not fixed because developers considered that the fix hurts code readability.

We manually inspected all Java and C/C++ bugs reported by CARAMEL and we find they are similar. The only exception is that CARAMEL-C can currently detect only bugs with one RI (Section IV), and therefore all the C/C++ bugs in this evaluation have one RI. The bug examples shown so far in the paper are from Java code. Figure 10 shows an example bug from GCC, which contains a Type 3 RI. This bug was confirmed and fixed by developers.

```
1 bool irred_invalidated = ...
2 FOR_EACH_EDGE (ae, ei, e->src->succs) {
3 + if (irred_invalidated) break; // FIX
4   if (ae != e && ae->dest != EXIT_BLOCK_PTR
5         && !bitmap_bit_p (seen, ae->dest->index)
6         && ae->flags & EDGE_IRREDUCIBLE_LOOP) {
7     irred_invalidated = true; // RI
8 }}
```

Fig. 10. A GCC performance bug found by CARAMEL

Figure 11 shows the detailed results for the new bugs found by CARAMEL. The numbers in the table refer to the numbers of distinct buggy loops, with each loop containing one or multiple RIs. 16 out of the 61 Java bugs in Figure 11 contain more than one RI. As explained in Section II-F, most of these 16 bugs contain RIs of the same type, with only a few bugs containing RIs of Type 2 and Type 3, as shown in the table (the column headers show the type of the RIs in the bug). CARAMEL-C can currently detect only bugs with one RI (Section IV), and therefore no C/C++ bug in Figure 11 contains multiple RIs.

Application	Type 1 RIs	Type 2+3 RIs	Type 3 RIs	Type 4 RIs	**SUM**
Ant	0	0	1	0	**1**
Groovy	2	0	7	0	**9**
JMeter	0	0	3	1	**4**
Log4J	0	0	5	1	**6**
Lucene	6	0	7	1	**14**
PDFBox	1	5	3	1	**10**
Sling	0	0	6	0	**6**
Solr	0	0	2	0	**2**
Struts	2	0	2	0	**4**
Tika	0	0	1	0	**1**
Tomcat	0	0	3	1	**4**
Chromium	0	0	13	9	**22**
GCC	1	0	21	0	**22**
Mozilla	0	0	20	7	**27**
MySQL	3	0	13	2	**18**
SUM:	**15**	**5**	**107**	**23**	**150**

Fig. 11. New bugs found by CARAMEL

CARAMEL found bugs in all 15 applications in Figure 11, including in GCC, which is highly tuned for performance and has been developed for more than two and a half decades. Indeed, *all the bugs that we reported to GCC have already been fixed by developers*.

CARAMEL found all four RI types in bugs. Looking at the type breakdown in Figure 11, we see Type 3 RIs are more frequent than RIs of other types. We manually inspect all the bugs reported by CARAMEL and we find the bugs containing Type 3 RIs typically appear in code performing a linear search for objects that have certain properties, such as the bugs in

Figure 4. This is a common operation in real-world code, and therefore it presents more opportunities for developers to introduce such bugs.

B. False Positives

CARAMEL reports few false positives, as shown in Figure 12. We manually inspect all false positives and find three causes. *Complex Analysis* false positives occur when CARAMEL incorrectly judges, in step three of its algorithm, that some L-Break conditions are satisfiable. Such false positives can be reduced by complex analysis, as described in this section, but the number of such false positives does not justify the added complexity. *Concurrent* false positives are caused by expressions that appear to be loop-invariant, but that in reality can be modified by a concurrent thread. Such false positives can be reduced using static analysis [41] or heuristics [57]. *Infrastructure* false positives occur because WALA may give incomplete results, as described later in this section.

Application	Complex Aly.	Concurrent	Infrastructure
Ant	0	1	0
Groovy	0	0	0
JMeter	0	0	0
Log4J	0	2	0
Lucene	2	3	0
PDFBox	0	0	1
Sling	0	0	1
Solr	0	0	1
Struts	1	0	1
Tika	2	0	0
Tomcat	1	0	3
Chromium	0	0	0
GCC	1	0	0
Mozilla	2	0	0
MySQL	1	0	0
SUM:	**10**	**6**	**7**

Fig. 12. False positives and their cause

Figure 13 shows a Complex Analysis false positive from Tomcat. Here, CARAMEL detects that RI 1 and RI 2 are of Type 1 with I-Break conditions `allRolesMode == AllRolesMode.AUTH_ONLY_MODE` equals false (line 5) and `allRolesMode == AllRolesMode.STRICT_AUTH_ONLY_MODE` equals false (line 9), respectively. CARAMEL incorrectly judges, in step three of its algorithm, that it is possible to satisfy these two I-Break conditions simultaneously, and CARAMEL reports a bug. However, this conclusion is wrong because `AllRolesMode` is an enumeration with three values and the loop is executed only when `allRolesMode` is not equal to the third value. Therefore, when the loop executes, `allRolesMode` can only have one of the two remaining values. CARAMEL could avoid this false positive by employing a complex analysis that takes into account the values enumeration variables can take, *and* that determines the condition under which the entire loop is executed.

Figure 14 shows a Concurrent false positive from Lucene. Here, CARAMEL detects Atoms `channel.isClosed()` and `thread == null` (lines 1, 2) as loop-invariant, and therefore concludes that the RI on line 2 is of Type 1. However, this

```
1  AllRolesMode allRolesMode = ...;
2  for (int i = 0; i < constraints.length; i++) {
3    SecurityConstraint constraint = constraints[i];
4    if (constraint.getAllRoles()) {
5      if (allRolesMode == AllRolesMode.AUTH_ONLY_MODE) {
6        log.debug("Granting access for ..."); // RI 1
7      }
8      String[] roles = request.getContext().findSecurityRoles();
9      if (roles.length == 0 && allRolesMode == AllRolesMode.
             STRICT_AUTH_ONLY_MODE) {
10       log.debug("Granting access for ..."); // RI 2
11 }}}
```

Fig. 13. Complex Analysis false positive from Tomcat

conclusion is wrong because `channel` and `thread` are both shared variables that can be modified by another thread, in parallel with this loop's execution. This is a typical custom synchronization that can be detected by existing tools [57].

```
1 while (!channel.isClosed()) {
2   if (thread == null) return; // RI
3   try {sleep(RETRY_INTERVAL);} catch (Exception e) {/*ignored*/}
4 }
```

Fig. 14. Concurrent false positive from Lucene

The Infrastructure false positives appear because, to scale to large programs, we instruct WALA to not analyze code inside some libraries, e.g., `java.awt` and `javax.swing`, as recommended in WALA's performance guidelines [1]. This may cause WALA to give incomplete results, which may cause CARAMEL to miss some RIs in step one of its algorithm.

C. Automatic Fix Generation

CARAMEL successfully generates fixes for 149 out of 150 bugs. We manually inspected all these fixes and confirmed all are correct. For one bug in Tomcat, CARAMEL-J could not generate a fix due to a limitation in WALA. Specifically, WALA does not always provide line number information for assignment instructions. Therefore, for this Tomcat bug, CARAMEL could not generate a fix like the fix in Figure 5(a), because CARAMEL did not know where to insert the `break`. For the other bugs involving Type 4 RIs, WALA did not suffer from this problem and CARAMEL could generate fixes. Note that loop headers are *not* assignment instructions. Therefore, generating fixes immediately before or after loop headers, which is how CARAMEL generates fixes for loops containing other RI types (Section III-E), is not affected by this limitation.

We compare the fixes generated by CARAMEL with the fixes adopted by developers and find they are similar, with one exception. For bugs containing only one Type 3 RI, CondBreak fixes are different from manual fixes, because developers prefer to insert a `break` immediately after the RI. CARAMEL could have easily followed developers' style and generated the same fixes, if WALA was able to provide the line number of the RI. However, as describe above, WALA cannot guarantee to provide line number for assignment instructions, and CARAMEL chooses to generate the basic CondBreak fixes, inserted right after the loop header.

D. Overhead

Figure 15 shows CARAMEL's running time in minutes. Columns *Sequential* and *Parallel* give the time for the sequential and parallel version CARAMEL using three threads,

respectively. CARAMEL's parallel version divides the loops in *N* groups, starts *N* threads, and lets each thread analyze the loops in one group. CARAMEL-J's parallel execution takes up to two hours, for all but three applications. Most of this time is spent in WALA's inter-procedural pointer analysis. We consider this running time acceptable, because developers do not need to write test code, like for a dynamic bug detection technique, or devise complex usage scenarios, like for a profiler. Furthermore, after the initial run, subsequent runs of CARAMEL on the same code can focus only on code that has changed, in the spirit of regression testing [59]. The speedup of the parallel version over the sequential version is over 2.5X for all but four applications, which shows CARAMEL makes effective use of modern multi-core machines. CARAMEL-C is much faster than CARAMEL-J because CARAMEL-C does not use interprocedural pointer-alias analysis, but instead conservatively assumes that every write to heap is an RI. We did not consider necessary to parallelize CARAMEL-C because the running time is small, ranging from several minutes for GCC and MySQL up to one and a half hours for Chromium.

Application	Sequential	Parallel	Speedup (X)
Ant	183	72	2.54
Groovy	345	128	2.70
JMeter	118	52	2.27
Log4J	108	45	2.40
Lucene	1068	417	2.56
PDFBox	106	38	2.79
Sling	355	190	1.87
Solr	1062	627	1.69
Struts	226	77	2.94
Tika	113	42	2.69
Tomcat	258	89	2.90
Chromium	85	n/a	n/a
GCC	3	n/a	n/a
Mozilla	52	n/a	n/a
MySQL	10	n/a	n/a

Fig. 15. CARAMEL running time (minutes)

VI. DISCUSSION

Importance of Bugs That Have CondBreak Fixes: The importance of a bug is ultimately decided by developers: if the developers think the bug is important enough to fix, it means that detecting and fixing that bug is important. We evaluated CARAMEL on 15 real-world applications, and 116 of the bugs found by CARAMEL are already fixed by the developers.

Generality of Bugs That Have CondBreak Fixes: These are definitely not all the bugs that have non-intrusive fixes. However, these bugs are general: 15 real-world applications, written both in Java and in C/C++, contain such bugs. We hope CARAMEL's promising results will motivate future research to detect other performance bugs that have non-intrusive fixes.

Estimating the Offered Speedup: CARAMEL is a static technique and cannot easily estimate the speedup offered by the bug fix. Developers may appreciate such additional information. However, as our results show, developers typically fix the bugs reported by CARAMEL even without knowing the exact speedup. Future work can try to estimate the speedup, perhaps using techniques inspired by [9], [19].

VII. RELATED WORK

Improving Performance and Detecting Performance Problems: Several techniques identify slow code [13], [21], [36], [40], [56], [60], runtime bloat [5], [15], [43], [58], or increasing execution time [11], [16], [61]. Siegmund et al. [50] and Guo et al. [20] predict how configuration options influence performance, Trubiani et al. [53] consider uncertainty in performance modeling, Malik et al. [33] detect deviations in load tests, and Lu and Song [51] investigate design points in performance statistical debugging. Other techniques generate performance tests [8], [10], [18], [48], [62], detect performance regression [47], latent performance bugs [26], concurrency performance problems [32], [46], [52], and idle time [4]. Unlike all these techniques, CARAMEL makes the novel design decision to focus on performance bugs that have simple and non-intrusive fixes. Specifically, CARAMEL detects performance bugs that have CondBreak fixes. Such bugs are not covered by previous work.

Automatic Bug Fixing: Several recent techniques have been propose to automatically fix bugs [30]. GenProg [17], [55] uses genetic programming, LASE [28], [35], SysEdit [34], and FixWizard [44] use edits similar to previous edits. Other techniques [3], [12], [14], [27], [31], [42], [49], [54] repair bugs using approaches such as SMT, semantic analysis, software contracts, developer input, etc. Unlike these techniques, CARAMEL automatically fixes performance bugs. Furthermore, taking advantage of the unique properties of the bugs it detects, CARAMEL successfully fixes 149 out of 150 bugs.

VIII. CONCLUSIONS

Performance bugs affect even well tested software written by expert programmers. In practice, fixing a performance bug can have both benefits and drawbacks, and developers fix a performance bug only when the benefits outweigh the drawbacks. Unfortunately, the benefits and drawbacks can be difficult to assess accurately. This paper presented CARAMEL, a novel technique that detects and fixes performance bugs that have non-intrusive fixes likely to be adopted by developers. Specifically, CARAMEL detects performance bugs that have CondBreak fixes: when a condition becomes true during loop execution, just break out of the loop. We evaluated CARAMEL on *real-world applications*, including 11 popular Java applications (Ant, Groovy, JMeter, Log4J, Lucene, PDFBox, Sling, Solr, Struts, Tika, and Tomcat) and 4 widely used C/C++ applications (Chromium, GCC, Mozilla, and MySQL). CARAMEL found *61 new performance bugs* in the Java applications and *89 new performance bugs* in the C/C++ applications. Of these bugs, *developers have already fixed 51 performance bugs* in the Java applications and *65 performance bugs* in the C/C++ applications. CARAMEL makes a promising first step in detecting performance bugs that have non-intrusive fixes.

ACKNOWLEDGMENTS

This material is based upon work partially supported by NSF under Grant Nos. CCF-1439091, CCF-1217582, CCF-1054616, CNS-0958199, CCF-0916893, CCF-1442157, and CCF-1439957, and the Alfred P. Sloan Foundation.

REFERENCES

[1] WALA: T.J. Watson libraries for analysis. http://wala.sourceforge.net.

[2] Adobe Systems Inc. Known issues and bugs. http://helpx.adobe.com/-acrobat/kb/known-issues-bugs-acrobat-reader.html.

[3] M. Alkhalaf, A. Aydin, and T. Bultan. Semantic differential repair for input validation and sanitization. In *ISSTA*, 2014.

[4] E. A. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *OOPSLA*, 2010.

[5] S. Bhattacharya, M. G. Nanda, K. Gopinath, and M. Gupta. Reuse, recycle to de-bloat software. In *ECOOP*, 2011.

[6] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley, 2010.

[7] Bugzilla@Mozilla. Bugzilla keyword descriptions. https://bugzilla.mozilla.org/describekeywords.cgi.

[8] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *ICSE*, 2009.

[9] R. P. L. Buse and W. Weimer. The road not taken: Estimating path execution frequency statically. In *ICSE*, 2009.

[10] F. Chen, J. Grundy, J. Schneider, Y. Yang, and Q. He. Automated analysis of performance and energy consumption for cloud applications. In *ICPE*, 2014.

[11] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *PLDI*, 2012.

[12] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *ASE*, 2009.

[13] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining hot calling contexts in small space. In *PLDI*, 2011.

[14] F. Demarco, J. Xuan, D. L. Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *CSTVA*, 2014.

[15] 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.

[16] S. Goldsmith, A. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *FSE*, 2007.

[17] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*, 2012.

[18] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *ICSE'12*.

[19] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[20] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *ASE*, 2013.

[21] 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.

[22] P. Huang, X. Ma, D. Shen, and Y. Zhou. Performance regression testing target prioritization via performance risk analysis. In *ICSE*, 2014.

[23] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, 2012.

[24] R. Johnson. More details on today's outage, 2010. https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919.

[25] P. Kallender. Trend Micro will pay for PC repair costs, 2005. http://www.pcworld.com/article/120612/article.html.

[26] C. E. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding latent performance bugs in systems implementations. In *FSE*, 2010.

[27] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE*, 2013.

[28] M. Kim and N. Meng. Recommending program transformations—Automating repetitive software changes. In *Recommendation Systems in Software Engineering*, pages 421–453. 2014.

[29] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[30] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[31] P. Liu and C. Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *ICSE*, 2012.

[32] T. Liu and E. D. Berger. SHERIFF: Precise detection and automatic mitigation of false sharing. In *OOPSLA*, 2011.

[33] H. Malik, H. Hemmati, and A. E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *ICSE*, 2013.

[34] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: Generating program transformations from an example. In *PLDI*, 2011.

[35] N. Meng, M. Kim, and K. S. McKinley. LASE: Locating and applying systematic edits by learning from examples. In *ICSE*, 2013.

[36] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, H. Cai, and G. Yin. An online service-oriented performance profiling tool for cloud computing systems. *Frontiers of Computer Science*, 7(3):431–445, 2013.

[37] Microsoft Corp. Connect. https://connect.microsoft.com/.

[38] D. Mituzas. Embarrassment, 2009. http://dom.as/2009/06/26/embarrassment/.

[39] I. Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, 2009.

[40] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In *PLDI*, 2010.

[41] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, 2006.

[42] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *ICSE*, 2013.

[43] K. Nguyen and G. H. Xu. Cachetor: Detecting cacheable data to remove bloat. In *ESEC/FSE*, 2013.

[44] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *ICSE*, 2010.

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

[46] J. Oh, C. J. Hughes, G. Venkataramani, and M. Prvulovic. LIME: A framework for debugging load imbalance in multi-threaded execution. In *ICSE*, 2011.

[47] M. Pradel, M. Huggler, and T. R. Gross. Performance regression testing of concurrent classes. In *ISSTA*, 2014.

[48] M. Pradel, P. Schuh, G. C. Necula, and K. Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *OOPSLA*, 2014.

[49] H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, and L. J. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *ICSE*, 2012.

[50] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. S. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *ICSE*, 2012.

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

[52] A. Tarvo and S. P. Reiss. Using computer simulation to predict the performance of multithreaded programs. In *ICPE*, 2012.

[53] C. Trubiani, I. Meedeniya, V. Cortellessa, A. Aleti, and L. Grunske. Model-based performance analysis of software architectures under uncertainty. In *QoSA*, 2013.

[54] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *ISSTA*, 2010.

[55] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, 2009.

[56] X. Xiao, S. Han, D. Zhang, and T. Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ISSTA*, 2013.

[57] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI*, 2010.

[58] G. H. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions on Software Engineering Methodology*, 23(3):23, 2014.

[59] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

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

[61] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *PLDI*, 2012.

[62] P. Zhang, S. G. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *ASE*, 2011.