Performance Diagnosis for Inefficient Loops

Linhai Song
Fireeye, Inc.
linhai.song@fireeye.com

Shan Lu
University of Chicago
shanlu@uchicago.edu

***Abstract*—Writing efficient software is difficult. Design and implementation defects can cause severe performance degradation. Unfortunately, existing performance diagnosis techniques like profilers are still preliminary. They can locate code regions that consume resources, but not the ones that *waste* resources.**

In this paper, we first design a root-cause and fix-strategy taxonomy for inefficient loops, one of the most common performance problems in the field. We then design a static-dynamic hybrid analysis tool, LDoctor, to provide accurate performance diagnosis for loops. We further use sampling techniques to lower the run-time overhead without degrading the accuracy or latency of LDoctor diagnosis. Evaluation using real-world performance problems shows that LDoctor can provide better coverage and accuracy than existing techniques, with low overhead.

I. Introduction

A. Motivation

Performance bugs[1] are software implementation mistakes that cause unnecessary performance degradation in software. They widely exist in deployed software due to the complexity of modern software [11, 14, 20, 30, 32]. They annoy end users and waste energy during production runs, and have already caused highly publicized failures [13, 21]. Tools that can help developers quickly and accurately diagnose performance problems are sorely desired.

Performance diagnosis is different from performance testing or bug detection: it does *not* aim to expose previously unknown performance anomalies. Instead, it is similar with general failure diagnosis: it is applied *after* an unexpected software behavior (i.e., symptom) is observed, hoping to identify the root cause of this symptom and suggest strategies to eliminate this symptom. In the context of performance problems, the symptom is execution slowness [31]; the root cause is about which code region is inefficient and why. An effective diagnosis tool can help developers quickly and correctly understand and fix already-exposed performance symptoms.

Also like general failure diagnosis, ideal performance diagnosis tools should satisfy three criteria.

- Coverage. Real-world performance problems are caused by a wide variety of reasons. A good diagnosis tool should handle a good portion of them.
- Accuracy. Which code regions are inefficient and why they are inefficient need to be accurately identified.
- Performance. Diagnosis often requires collecting run-time information. The lower the overhead is, the easier for the tool to deploy, especially for production-run usage.

No existing tools can satisfy these three requirements.

Profiling is the state of practice in performance diagnosis. It is far from providing the desired *accuracy*, as it is designed to tell where computation resources are spent, but not where and why resources are wasted. In many cases, the root-cause function may not even get ranked by the profiler [31].

Performance bug detection tools use program analysis to identify code regions that match specific inefficiency patterns [7, 23–26, 33–35]. Unfortunately, these tools are not designed and hence are unsuitable to identify code regions that contribute to a specific slowness symptom. They do not provide *coverage* for a wide variety of real-world problems; they are not guided by performance failure symptoms, and hence are at disadvantage in terms of diagnosis *accuracy*; dynamic detection tools often lead to 10X slowdowns or more [23, 24, 35], not ideal in terms of *performance*.

Recently, progress has been made on statistical debugging for performance diagnosis [31]. This approach compares runs with and without problematic performance, and identifies control-flow constructs, such as a branch *b* or a loop *l*, that are most correlated with the execution slowness. Unfortunately, this approach is *not* effective for loop-related performance problems, which contribute to two thirds of real-world performance problems studied in previous work [11, 31]. It cannot tell whether and how loop *l* is inefficient and hence is limited in its diagnosis *coverage* and *accuracy*.

Figure 1 shows a performance problem in GCC. Recursive function `mult_alg` computes the best algorithm for multiplying `t`, a time-consuming computation. At run time, `mult_alg` is often invoked for many times, and often with the same parameter partly due to its recursive nature. To avoid redundant computation across different instances of `mult_alg`, developers used a hash-table `alg_hash` to remember which parameter `t` has been processed in the past and what is the result. Unfortunately, a mistake in the type declaration of hash-table entry `hash_entry` makes the memoization useless for large `t`. In many cases, a slow path is taken, when the fast path should have been taken. This mistake does not affect software correctness, but causes a large amount of redundant computation[2] and hurts GCC

[1] We also refer performance bugs as performance problems following previous works in this area [11, 24, 31].

[2] This will be considered as a loop-redundancy problem later in this paper, as we consider recursive functions as a special case of loops.

```
struct hash_entry {
-  unsigned int t;
+  HOST_WIDE_UINT t;
};

void mult_alg(... HOST_WIDE_UINT t, ...) {
  hash_index = hash (t);
  if (alg_hash[hash_index].t == t)
  {
    //fast path: reuse previous results
  }else{
    //slow path: expensive recursive computation
    ...
    mult_alg (...);
  }
}
```

Fig. 1: A real-world performance bug in GCC (the '-' and '+' demonstrate the patch; variable and function names are simplified for demonstration purposes)

performance severely, causing hundreds of times slow down for GCC test cases.

Debugging this performance problem is challenging. According to the discussion forum of GCC, developers identified `mult_alg` as the most time-consuming function through profilers early on. However, they did not figure out whether `mult_alg` is inefficient, which part of it is inefficient (`mult_alg` is a large function with about 400 lines of code), and how it is inefficient, until several weeks later.

If a tool can tell developers not only *which* loop or function is responsible for execution slowness, but also *why* and *how* it is inefficient, diagnosis and fixing would be much easier.

B. Contributions

This paper presents a tool LDoctor that can help effectively diagnose inefficient loop problems, the most common type of performance problems [11, 31], with good coverage, accuracy, and performance. After users or developers observe a performance failure symptom, LDoctor can automatically judge whether the symptom is caused by inefficient loops and provide detailed root-cause information that helps understand and fix inefficient loops.

LDoctor tackles this challenging problem in three steps.

First, figuring out a root-cause taxonomy for inefficient loops. Our taxonomy categorizes all inefficient loops into two main categories: *resultless*, when a large amount of computation produces no side effects, and *redundancy*, when a large amount of computation produces already-available results. Each main category is further divided to sub-categories. We strive to make the taxonomy both general enough to cover common inefficient loops, and specific enough to guide the design of LDoctor and eventually help developers understand and fix performance problems (Section II).

Second, building a tool(kit) LDoctor that can automatically and accurately identify whether and how a suspicious loop is inefficient, following these principles (Section III):

- Focused checking. Different from performance-bug detectors that blindly check the whole software, LDoctor focuses on loops that are most correlated with performance symptoms. This focus is crucial for LDoctor to achieve both high accuracy and high coverage.

```
1  for (sn=script->start, offset=0;
2                          !sn; sn=sn->next){
3    offset += sn->delta;
4    if (offset == target)
5       return sn;
6  } //script is a linked list with one node for
7  //each byte-code instruction in a JavaScript file
```

Fig. 2: A resultless 0*1? bug in Mozilla

- Taxonomy guided design. To provide good coverage, we follow the root-cause taxonomy discussed above and design analysis routines for every root-cause category. Given a candidate loop, LDoctor applies a series of analysis to see if it matches any type of inefficiency.
- Static-dynamic hybrid analysis. Static analysis alone cannot accurately identify inefficiency root causes, as some inefficiency only happens or matters under specific workload; dynamic analysis alone will cause too large run-time overhead. Therefore, we use a hybrid approach to achieve both performance and accuracy goals.

Third, using sampling to further lower the run-time overhead of LDoctor, without degrading diagnosis capability. Random sampling is a natural fit for performance diagnosis due to the repetitive nature of inefficient loops (Section III-B3).

We evaluated LDoctor on 39 real-world performance problems, coming from two representative benchmark suites [24, 31]. Evaluation results show that LDoctor can accurately identify detailed root cause for all benchmarks and provide correct fix-strategy suggestion for most benchmarks. All of these are achieved with low run-time overhead.

II. Root-Cause Taxonomy

To guide the design of automated diagnosis tools, we need a taxonomy for inefficient loops that satisfies three requirements: (1) *Coverage*, covering a big portion of real-world inefficient loop problems; (2) *Actionability*, each root-cause category being informative enough to help developers fix a performance problem; (3) *Generality*, allowing automated diagnosis to be applied to many applications.

Previous work identified a wide variety of performance root-cause categories. However, existing taxonomies do not satisfy all these requirements and hence cannot be directly used by us. Therefore, we have designed our own taxonomy, as presented below. We will discuss how our taxonomy meets these three requirements qualitatively in this section and quantitatively using real-world performance bugs in Section IV.

A. Resultless loops

Resultless loops spend a lot of time in computation that does not produce results useful after the loop (i.e., no side effects). They can be further categorized to four sub-types based on which part of the loop is (not) producing useful results.

0* loops never produce any results in any iteration. They are rare in mature software systems.

How to fix? They should be deleted from software.

0*1? loops only produce results in the last iteration, if any. They are often related to search: check a sequence of elements

```
+ if(warning_candidate_p(add->expr)) {
    for (tmp = *to; tmp; tmp = tmp->next)
     if (candidate_equal_p (tmp->expr, add->expr)
           && !tmp->writer)
     {
       ...
       tmp->writer = add->writer;
     }
+ }
```

Fig. 3: A resultless [0|1]* bug in GCC

```
char * sss_xph_generate(node_t* aNode)
{
   int count=0;
   for (n = aNode; n ; n = aNode->prev)
     if (n->localName == aNode->localName
          && n->namespaceURI == aNode->namespaceURI)
          count++;
   ...
}//called for every node in a list
```

Fig. 4: A cross-loop redundant bug in Mozilla

one by one until the right one is found. Whether these loops are efficient or not depends on the workload. An example is shown in Figure 2. Large JavaScript files often fill the `script` list with tens of thousands of nodes and cause poor performance.

How to fix? They are often fixed by data-structure changes. For example, the patch for the bug in Figure 2 simply replaced the `script` list with a hash table.

[0|1]* loops may or may not produce results in each iteration. For some workload, the majority of iterations do not produce results and cause performance problems perceived by users. Figure 3 shows such an example. Users complained that compilation became extremely slow when the `-Wsequence-point` checking is enabled. The slowness was caused by the `for` loop in the figure. As the algorithm behind this loop has quadratic complexity in the number of operands in an expression, programs with long expressions suffer severe slow-downs. After further diagnosis, developers observed that this loop rarely had any side-effects, as the `if` condition was rarely satisfied.

How to fix? The patch shown in Figure 3 reflects the typical fix strategy for this type of inefficient loops. The developers should think about what exactly is the condition for the loop to produce results, and use that condition to skip the whole loop whenever possible.

1* loops in this category always generate results in almost all iterations. They are inefficient because their results are useless due to high-level semantic reasons. For example, several Mozilla performance problems are caused by loops that contain intensive GUI operations whose graphical outcome may not be observed by humans and hence can be optimized.

How to fix? Since a deep understanding of software semantics is required to understand the inefficiency of these loops, the fix strategies for these loops likely vary from case to case and are difficult to automate.

B. Redundant loops

Redundant loops spend a lot of time in repeating computation that is already conducted. They can be further categorized to two sub-types based on which part of the loop is the unit of redundancy.

Cross-iteration Redundancy: one iteration repeats what was already done by an earlier iteration of the same loop. Here, we consider a recursive function as a loop, treating one function-call instance as one loop iteration.

How to fix? Intuitively, most redundancy problems can be fixed through memoization or batching — either caching the earlier computation results and skip some following iterations; or combining multiple iterations' work together. For example, the bug shown in Figure 1 is caused by redundant computation across different invocations of recursive function `mult_alg`. The patch essentially enables memoization.

Cross-loop Redundancy: one dynamic instance of a loop spends a big chunk, if not all, of its computation in repeating the work already done by an earlier instance of the same loop.

How to fix? Just like that in cross-iteration redundancy, memoization and batching are the typical fix strategies for this type of loops. Mozilla#477564 shown in Figure 4 is an example. The buggy loop counts how many previous siblings of the input `aNode` have the same name and URI. There is an outer loop, not shown in the figure, that repeatedly updates `aNode` to be its next sibling and calls `sss_xph_generate` with the new `aNode`. This bug is fixed by adding an extra field for each node to save the calculated count, so that a new count value can be calculated by simply adding one to the saved count value of the nearest previous sibling with the same name and URI.

Intuitively, the above categories cover many common inefficient loop problems; each sub-category is concrete to guide the design of a diagnosis tool; none of these (sub-)categories involve application specific knowledges or heuristics.

III. LDoctor design

The design of LDoctor follows three principles. First, *symptom oriented*. LDoctor will be used together with other performance diagnosis tools [31] or profilers, and focus on a small number of loops that are most correlated with a specific already-observed performance symptom, instead of the whole program. Therefore, we will have different design trade-offs in terms of coverage and accuracy, comparing with bug detection tools. Second, *static–dynamic hybrid*. Static analysis alone cannot provide all the needed information to judge whether a loop is inefficient. However, dynamic analysis alone will incur too much overhead. Therefore, we use a hybrid approach throughout our design. Third, *sampling*. Loop-related performance problems have the unique nature of repetitiveness, which make them a natural fit for random sampling. We will design different sampling schemes for different analysis.

We envision LDoctor to be used in the following way. Existing profilers and statistical debugging tool first identify suspicious loops that are most correlated with certain performance symptoms [31]. LDoctor will then apply a series of analysis to judge whether a given loop belongs to any root-cause type discussed in Section II and recommend corresponding fix strategy. This series of analysis could be

conducted during either in-house diagnosis, where failure-triggering inputs are available, or production runs, with the run-time analysis component of LDoctor running at user sites.

A. Resultless Checker

Our resultless checker includes two parts. First, we use static analysis to figure out which are the side-effect instructions in a loop and hence decide whether a loop might belong to 0*, 0*1?, [0|1]*, or 1*. Second, to determine whether 0*1? and [0|1]* loops are indeed inefficient under specific workload, we use dynamic analysis to figure out what portion of loop iterations are resultless at run time.

1) Static analysis: LDoctor considers two types of instructions as side-effect instructions: (1) writes to heap or global variables; (2) writes to stack variables defined outside the loop and may be used after the loop (checked through liveness analysis). LDoctor also analyzes all functions called by a loop directly or indirectly — a function F that updates variables defined outside F makes the corresponding call statement in F's caller a side-effect instruction.

LDoctor then categorizes loops into four types. Given a natural loop l, when l contains at least one side-effect instruction along every path that starts from the loop header and ends at the loop header, it is a 1* loop; if there exists at least one side-effect instruction inside l, but not on every path, it is a [0|1]* loop; if there is no side-effect instructions inside l, l is either a 0* or a 0*1? loop, which will be differentiated by checking all the loop exit-blocks. For example, the loop in Figure 2 is 0*1?, because its exit block on Line 5 contains side effects.

2) Dynamic monitoring: Further checking is conducted to determine whether 0*1? and [0|1]* loops are inefficient.

For a 0*1? loop, since it only generates results in the last iteration, we only need to know the total number of loop iterations of each loop instance to figure out the loop *resultful rate*, which we define as the ratio between the number of iterations with side effects and the total number of iterations. The implementation is straightforward — we initialize a local counter to be 0 in the pre-header of the loop; we increase the counter by 1 in the loop header to count the number of iterations; we dump that counter to log when the loop exits.

For [0|1]*, we need to count not only the total number of iterations, but also the exact number of iterations that execute side-effect instructions at run time. To do that, our instrumentation uses a local boolean variable `HasResult` to represent whether one iteration have side effect or not. `HasResult` is set to `False` in the loop header, and set to `True` after each side-effect instruction. It will be used to help count the number of side-effect iterations.

B. Redundancy Checker

We will compare the computation of different iterations from one loop instance and that of different loop instances from one static loop, and judge whether there is redundancy. Specifically, several questions need to be answered.

What to compare? Given two loop iterations (or loop instances) c_1 and c_2, since they originate from the same source code C, a naive approach is to record and compare the value read by every memory read instruction in c_1 and c_2. We could do better by checking fewer instructions, as some of these values are determined by values read earlier in c_1 and c_2. Informally speaking, we only need to compare *input* values of c_1 and c_2 to decide whether they are redundant with each other. We will present formal definition of *input* and the detailed algorithms in Section III-B1.

How to compare? Naively, we can judge two iterations or two loop instances to be redundant with each other only when they read exactly the same data and conduct exactly the same computation. However, in practice, redundant loops may be doing largely, but not completely, the same computation across iterations or loop instances. We will discuss how we handle this issue in Section III-B2.

How to lower the overhead of record-and-compare? We will use both static optimization (Section III-B4) and dynamic sampling (Section III-B3) to reduce the amount of data that is recorded and compared, reducing time and spatial overhead.

1) Identifying and recording inputs: Informally, we use static analysis to identify a set of memory-read instructions that the computation of code C depends on. We refer to these instructions as *input instructions* for C. The values returned from them at run-time, referred to as *inputs*, will be tracked and compared to identify redundant computation among different instances of C.

Specifically, LDoctor first identifies side-effect instructions in C, similar with that in Section III-A1. It then conducts backward static slicing from these instructions, considering control and data dependency. For every memory read r (`read(v)`) that static slicing encounters, LDoctor checks whether r satisfies either one of the following two conditions. If r does, it is marked as an *input* instruction; otherwise, LDoctor continues growing the slice beyond r. The two conditions are: (1) the value of v is defined outside C; (2) v is a heap or global variable. The rationale for the first condition is that slicing outside C is unnecessary for redundancy judgement among instances of C. The rationale for the second condition is that tracking data-dependency through heap or global variables is complicated in multi-threaded C/C++ programs. The analysis for cross-iteration and cross-loop redundancy analysis is similar — simply replacing C in the above algorithm with one iteration or the whole loop body.

Our analysis considers function calls inside C — slicing is conducted for return values of callees and side-effect instructions inside callees. We omit encountered constant values through slicing, as they do not affect redundancy judgement.

2) Identifying redundant loops: After identifying *inputs* using static analysis, LDoctor instruments the program to record *input* values at run time. The run-time trace also includes information that differentiate values recorded from different instructions, loop iterations, loop instances, etc. Once the trace under problematic workload is collected either during off-line debugging or production runs, LDoctor will process the trace and decide whether the loops under study contain cross-iteration or cross-loop redundancy.

```
1  int found = -1;
2  while ( found < 0 ) {
3  //Check if string source[] contains target[]
4    char first  = target[0];
5    int max = sourceLen - targetLen;
6
7    for (int i = 0; i <= max; i++) {
8      // Look for first character.
9      if (source[i] != first) {
10       while (++i <= max && source[i] != first);
11     }
12
13     // Found first character
14     // now look at the rest
15     if (i <= max) {
16       int j = i + 1;
17       int end = j + targetLen - 1;
18       for (int k = 1; j < end && source[j] ==
19               target[k]; j++, k++);
20
21       if (j == end) {
22         /* Found whole string target. */
23         found = i;
24         break;
25       }
26     }
27   }
28
29   //append another character; try again
30   source[sourceLen++] = getchar();
31 }
```

Fig. 5: A cross-loop redundant bug in Apache (The `for` loop on line 7 searches a string `source` for a target substring `target`. Since the outer loop on line 2 appends one character to `source` in every iteration (line 30), the `for` loop is always working on a similar `source` from its previous execution, with a lot of redundancy.)

a) High-level algorithms: We need to answer two questions: how to judge whether two iterations or loop-instances are doing redundant work, and how to judge whether a loop contains sufficient redundant computation to be inefficient. Our answer stick to one principle: there should be sufficient amount of redundant computation to make a loop likely culprit for a user-perceived performance problem and to make itself worthwhile to get optimized by developers.

For the first question, LDoctor takes a strict definition for cross-iteration redundancy and a looser definition for cross-loop redundancy checking. Specifically, two iterations that conduct exactly the same computation are considered redundant; two loop instances that conduct largely the same computation are considered redundant. The rationale is that a whole loop instance contains a lot of computation, much more than one iteration in general. Even if only part of its computation is redundant, it could still be the root-cause of a user-perceived performance problem and worth developers' attention. In practice, we rarely see different loop instances doing exactly the same computation. In cross-loop redundancy examples shown in Figure 4 and Figure 5, each loop instance is doing similar, but not exactly the same, work from its previous instance. For the second question, we believe there should be a threshold. A loop will be considered inefficient if its *redundancy rate* goes beyond the threshold.

b) Detailed algorithm implementation: The implementation of checking cross-iteration redundancy is straightforward. We calculate a loop's *cross-iteration redundancy rate* based on the number of (distinct) iterations in a loop: $\text{Rate}_{\text{C.I.}} = 1 - \frac{\text{\# of distinct iterations}}{\text{\# of iterations}}$. This rate ranges between 0 and 1, the smaller it is the less redundancy the loop contains.

Checking cross-loop redundancy goes through several steps. First, for k dynamic instances of a static loop L that appear at run time, denoted as l_1, l_2, ..., l_k, we check whether redundancy exists between l_1 and l_2, l_2 and l_3, and so on. Second, we compute a *cross-loop redundancy rate* for L: $\text{Rate}_{\text{C.L.}} = \frac{\text{\# of redundant pairs}}{\text{\# of pairs}}$ (In this example, # of pairs is $k-1$). This rate ranges between 0 and 1, the smaller it is the less redundancy L contains. We only check redundancy between consecutive loop instances, because checking that between every pairs of loop instances is time consuming.

The key of this implementation is to judge whether two dynamic loop instances l_1 and l_2 are redundant or not. The challenge is that l_1 and l_2 may have executed different numbers of iterations; in different iterations, different sets of input instructions may have executed. Therefore, we cannot simply chain values from different input instructions and iterations together and compare two data sequences. Instead, we decide to check the redundancy for each input instruction across l_1 and l_2 first, and then use the average *redundancy rate* of all input instructions as the *redundancy rate* between l_1 and l_2.

We calculate the redundancy for one input instruction I by normalizing the edit-distance between the two sequences of values returned by I in the two loop instances, denoted by *SeqA* and *SeqB*. The exact formula is the following:

$$Redundancy(I) = \frac{EditDistance(SeqA,SeqB) - |Length(SeqA) - Length(SeqB)|}{max(Length(SeqA),Length(SeqB))}$$

3) Dynamic optimization (sampling): Recording values returned by every input instruction would lead to huge runtime overhead. LDoctor uses random sampling to reduce this overhead, which requires almost no changes to our redundancy identification algorithm discussed in Section III-B2.

Note that, although LDoctor uses sampling, its diagnosis is still conducted in just **one** run, with almost **no** sacrifice to diagnosis latency or quality. This is **different** from traditional sampling techniques for correctness diagnosis [17, 18], where many more failure runs and hence longer latency are needed once sampling is enabled. The reason is that performance bugs have a unique repetitive nature: a loop can cause a severe performance problem only when it contains many redundant iterations/instances. Therefore, we can still recognize redundant behavior in just **one** failure run, as long as the sampling is not insanely sparse (Section V).

For cross-iteration redundancy analysis, we randomly decide at the beginning of every iteration whether to track the values returned by input instructions in this iteration. The implementation is similar with previous sampling work [17, 18]. Specifically, we create a clone of the original loop iteration code, and insert value-recording instructions along the cloned copy. We then insert a code snippet that randomly decides to execute the cloned copy or the original copy at the beginning of a loop iteration.

For cross-loop redundancy analysis, We randomly decide at the beginning of every loop instance whether to track values for this instance. Since we will need to compare two consecutive loop instances for redundancy, once we decide to sample one loop instance, we will make sure to sample the immediate next loop instance too. The implementation is straightforward by cloning the whole loop and making sampling decisions in the loop pre-headers.

4) Static optimization: We conduct a series of static analysis to reduce the number of instructions we need to monitor.

First, we avoid tracking multiple reads that we can statically prove to return the same value. Since we implement LDoctor in LLVM, we leverage the SSA construction and the *mem2reg* pass in LLVM to avoid unnecessary tracking of stack variables. For example, the read of `max` on line 10 of Figure 5 is an *input* instruction of the `while` loop on the same line. LLVM identifies it as a loop invariant and lifts the read of `max` out of the loop. Consequently, LDoctor only records its value once during each loop, instead of each iteration. LLVM is conservative in lifting heap/global variables out of a loop. LDoctor conducts a best-effort loop invariant analysis for heap/global variables. For example, LDoctor identifies that the values of `aNode->localName` and `aNode->namespaceURI` do not change throughout one loop instance in Figure 4, and hence only traces them once outside the loop.

Second, we leverage the scalar evolution (SE) analysis in LLVM to remove the monitoring to some loop-induction related variables. SE analysis can tell which variables are loop-induction variables (e.g., `i` on line 10 of Figure 5) and what are their strides. In cross-iteration redundancy checking, if a loop iteration's input set contains a loop-induction variable, we know different iterations' inputs would be different and hence conclude that there is no redundancy without run-time analysis. In cross-loop redundancy checking, when the address of a read is a loop induction variable, such as `source[i]` on line 10 of Figure 5, we only record the address range at run time, instead of every value returned by the memory read. This optimization could lead to false positives — different loop instances may work on variables read from similar memory locations but with different values. We did not encounter such false positives in our experiments.

C. Fix Strategy Recommendation

LDoctor suggests a fix strategy for each inefficient loop and provides related information for carrying out that strategy. This process is straightforward following our taxonomy discussion in Section II, as there is often only one or two natural fix strategies for each sub-category of root causes.

Once a loop is identified as *resultless*, LDoctor will suggest the loop to be deleted in case of 0^*, conditionally skipped in case of [0|1]*, or a data-structure change in case of 0*1?. LDoctor also reports side-effect instructions and what percentage of loop iterations executes each such instruction, helping developers refactor the loop or change data structures.

When a loop is identified as *redundant*, LDoctor conducts extra analysis to decide whether batching or memoization should be used. For cross-iteration redundancy, LDoctor suggests batching if the redundancy is related to I/O operations and suggests memoization otherwise. Specifically, when the only side effect of a loop is from I/O operations and the same statement(s) is executed in every iteration, LDoctor suggests batching the specific I/O operations. Otherwise, LDoctor suggests memoization, and reports input instructions that lead to redundant computation and redundancy rate.

For cross-loop redundancy, whether to use memoization or batching often depends on which strategy is cheaper. LDoctor uses a simple heuristic. If the side effect of each loop instance is to update a constant number of memory locations, like the buggy loop in Figure 4 and Figure 5, we recommend memoization. If the side effect is to update a sequence of memory locations, with the number of locations increasing with the workload, memoization is unlikely to save much and hence batching is suggested. LDoctor will also report the memory reads that return the same values again and again, which can help decide what to memoize.

IV. Assessment of Root-Cause Taxonomy

A. Methodology

Previous work [11, 31] studied the on-line bug databases of five representative open-source software projects: (1) Apache suite, including HTTPD web server written in C/C++, TomCat server written in Java, and Ant build management tool in Java; (2) Chromium Google Chrome browser in C/C++; (3) GCC compiler suite in C/C++; (4) Mozilla suite, including Firefox web browser and Thunderbird Email client in C/C++ and JavaScript; (5) MySQL database server in C/C++. Through a mix of random sampling and manual inspection, they found 65 performance problems that are perceived and reported by users. Among these 65 problems, 45 problems are related to inefficient loops and hence are the target of the study here.

B. Assessment

Coverage: As shown in Table I, our taxonomy does cover all inefficient loops under study. Resultless loops are about as common as redundant loops (24 vs. 21). Not surprisingly, 0* loops are rare in mature software. All other root-cause sub-categories are well represented.

Actionability: As shown in Table I, the root-cause categories in our taxonomy are well correlated with fix strategies. This indicates that our taxonomy is actionable — once the root cause is identified, developers roughly know how to fix. For example, almost all 0*1? resultless loops are fixed by data-structure changes; all [0|1]* resultless loops are fixed by conditionally skipping the loop; almost all redundant loops are fixed either by memoization or batching. The only problem is that there are no silver bullets for fixing 1* loops.

Generality: The root-cause categories in our taxonomy are designed to be generic. Table I also shows that these categories each appears in multiple application in our study. The only exception is 0^*-resultless, which never appears.

In summary, the study above informally demonstrates that our taxonomy is suitable to guide our design of LDoctor.

	Apache	Chrome	GCC	Mozilla	MySQL	$M_{emoization}$	$B_{atching}$	C	S	O_{ther}	Total
# of Bugs	11	4	8	12	10	12	12	4	9	8	45
Cross-**iteration** Redundancy	7	1	2	1	1	7	4	0	0	1	12
Cross-**loop** Redundancy	3	0	2	2	2	5	4	0	0	0	9
0*1? Resultless	0	0	0	2	3	0	0	4	1	0	5
[0|1]* Resultless	0	1	1	1	1	0	0	0	4	0	4
0* Resultless	0	0	0	0	0	0	0	0	0	0	0
1* Resultless	1	2	3	6	3	0	4	0	4	7	15

TABLE I: Distribution among different applications and fix strategies for bugs with each root-cause category. (C: change data structure; S: conditionally skip loop; darker background means more correlation between root causes and fixes)

V. Evaluation of LDoctor

A. Methodology

Implementation and Platform We implement LDoctor in LLVM-3.4.2 [16], and conduct our experiments on an i5-3330S machine, with Linux 3.19 kernel.

Benchmarks LDoctor is a diagnosis tool that helps understand and fix already-manifested performance problems, not a detection tool that help predict not-yet-manifested problems. Consequently, our benchmarks are performance problems that have already happened in real world, and we will reproduce them to evaluate LDoctor. For a thorough evaluation, we use benchmarks from two different sources.

First, *general* benchmark suite. We evaluate LDoctor on all bugs, 18 in total, that we can reproduce among the 45 bugs discussed in Section IV. These 18 bugs cover a wide variety of inefficiency root causes, as shown in the upper half of Table II. We use the problem-triggering inputs reported by real-world users in LDoctor run-time analysis. Among these 18, seven are extracted from Java/JavaScript programs. For these benchmarks, we did our best to keep all bug-related data structures and caller/callee functions intact, and re-implement them in C++ following the original data and control flow.

Note that, we expect LDoctor to also work for most, if not all, of the remaining 27 bugs listed in Table I. However, these 27 bugs are too difficult to reproduce, as they depend on special hardware/software environment (e.g., Windows OS and .Net libraries) and cannot be easily extracted or reimplemented. For example, some bugs are related to GUI widgets, which are too complicated to re-implement. Some bugs' inefficient loops traverse big graphs, whose bug-triggering inputs (i.e., the graphs) are difficult to figure out without reproducing the original bugs.

Second, we evaluate LDoctor on Toddler benchmark suite [1, 24]. Toddler project provides the bug-triggering inputs and detailed explanation for 21 inefficient-loop bugs that have been confirmed and fixed by developers, and we evaluate LDoctor on *all* of them. Due to the focus of Toddler, all of these bugs are caused by inefficient *nested* loops and only cover two types of inefficiency root causes, as shown in the bottom half of Table II. We extract these bugs and re-implemented them in C/C++; we re-implement basic Java data structures following a recent version of `openjdk`. Each extracted benchmark contains at least five loops, except for two cases with four loops each.

The general benchmark suite and Toddler benchmark suite both provide a large set of repeatable inefficient loop problems that we can access. At the same time, they were initially set up for different reasons and methodologies, and hence well complement each other. All the benchmarks led to severe performance problems. After developers fixed these problems, the performance of each benchmark improves 4X – 500X under the user-reported workload in our experiments.

Evaluation settings Our evaluation uses existing statistical performance diagnosis tool [31] to process a performance problem and identify one or a few suspicious loops for LDoctor to analyze. For all but four benchmarks, statistical debugging identifies the real root-cause loop as the most suspicious loop. For the remaining four benchmarks, which all come from Table I, the real root-cause loops are ranked number 2, 2, 4, and 10.

To evaluate the coverage, accuracy, and performance of LDoctor, we mainly conduct three sets of evaluation. First, we apply LDoctor to the real root-cause loop to see if LDoctor can correctly identify the root-cause category and suggest a fix strategy. Second, we apply statistical performance debugging [31] to all our benchmarks and apply LDoctor to the top 5 ranked loops[3] to see how accurate LDoctor is. Third, we evaluate the run-time performance of applying LDoctor to the real root-cause loop.

We use 0.001 as the default *resultful rate* threshold for identifying 0*1? and [0|1]* resultless loops; we use 0.5 as the default *redundancy rate* threshold for identifying redundant loops. We use $1/100$ sampling rate for cross-loop redundancy analysis and $1/1000$ sampling rate for cross-iteration redundancy analysis (there tend to be more loop iterations than loop instances). All our diagnosis results require only **one** run under the problem-triggering input. All our performance results are obtained by taking average of **ten** runs, with the variation among different runs always less than 5%.

B. Coverage Results

Overall, LDoctor provides good diagnosis coverage, as shown by the check-marks in Table II (the "Did LDoctor Identify ..?" columns). LDoctor identifies the correct root cause for **all** 39 benchmarks, and suggests fix strategies that match what developers took in practice for 33 out of 39 cases.

[3]Some extracted benchmarks have fewer than 5 loops. We simply apply LDoctor to all loops in these cases.

Benchmark Information					Did LDoctor Identify ...?		Number of False Positives					
BugID	**KLOC**	**P. L.**	**RootCause**	**Fix**	**Root Cause**	**Fix Strategy**	**0*1?**	**[0|1]***	**C-I$_b$**	**C-I$_m$**	**C-L**	**Total**
Mozilla347306	88	C	0*1?	C	✓	✓	-	-	-	-	-	-
Mozilla416628	105	C	0*1?	C	✓	✓	-	-	-	-	-	-
Mozilla490742	-	JS	C-I	B	✓	✓	-	-	-	-	-	-
Mozilla35294	-	C++	C-L	B	✓	✓	-	-	-	-	-	-
Mozilla477564	-	JS	C-L	M	✓	✓	-	-	-	-	-	-
MySQL27287	995	C++	0*1?/C-L	C	✓	✗	-	0_1	-	-	-	0_1
MySQL15811	1127	C++	C-L	M	✓	✓	-	-	-	-	-	-
Apache32546	-	Java	C-I	B	✓	✓	-	-	-	-	-	-
Apache37184	-	Java	C-I	M	✓	✓	-	-	-	-	-	-
Apache29742	-	Java	C-L	B	✓	✓	-	-	-	-	-	-
Apache34464	-	Java	C-L	M	✓	✓	-	-	-	-	-	-
Apache47223	-	Java	C-L	B	✓	✓	-	-	-	-	-	-
GCC46401	5521	C	[0|1]*	S	✓	✓	-	0_1	-	-	-	0_1
GCC1687	2099	C	C-I	M	✓	✓	-	-	-	-	-	-
GCC27733	3217	C	C-I	M	✓	✓	-	-	-	-	-	-
GCC8805	2538	C	C-L	B	✓	✓	-	0_1	-	-	-	0_1
GCC21430	3844	C	C-L	M	✓	✓	0_1	0_3	-	0_1	0_1	0_6
GCC12322	2341	C	1*	S	✓	✗	0_1	0_1	-	0_1	0_1	0_4
Apache53622	-	Java	C-L, 0*1?	C	✓	✗	-	-	-	-	-	-
Apache53637	-	Java	C-L	B	✓	✓	-	-	-	-	-	-
Apache53803	-	Java	0*1?	C	✓	✓	-	-	-	-	-	-
Apache53821	-	Java	0*1?	C	✓	✓	-	0_1	-	-	-	0_1
Apache53822	-	Java	0*1?	C	✓	✓	0_1	-	-	-	-	0_1
Collections406	-	Java	C-L, 0*1?	B, C	✓	✓	-	-	-	-	-	-
Collections407	-	Java	0*1?	S	✓	✗	-	-	-	-	-	-
Collections408	-	Java	0*1?	S	✓	✗	-	0_2	-	-	-	0_2
Collections409	-	Java	C-L	B	✓	✓	-	-	-	-	-	-
Collections410	-	Java	C-L	B	✓	✓	-	-	-	-	-	-
Collections412	-	Java	C-L, 0*1?	B, C	✓	✓	-	-	-	-	-	-
Collections413	-	Java	0*1?	C	✓	✓	-	-	-	-	-	-
Collections425	-	Java	0*1?	S	✓	✗	-	-	-	-	-	-
Collections426	-	Java	0*1?	C	✓	✓	-	-	-	-	-	-
Collections427	-	Java	0*1?	C	✓	✓	-	0_1	-	-	-	0_1
Collections429-0	-	Java	0*1?	C	✓	✓	-	-	-	-	-	-
Collections429-1	-	Java	0*1?	C	✓	✓	-	-	-	-	-	-
Collections429-2	-	Java	0*1?	C	✓	✓	0_1	-	-	-	-	0_1
Collections434	-	Java	0*1?	C	✓	✓	0_1	-	-	-	-	0_1
Groovy5739-0	-	Java	0*1?	C	✓	✓	-	0_1	-	-	-	0_1
Groovy5739-1	-	Java	0*1?	C	✓	✓	-	0_1	-	-	-	0_1

TABLE II: LDoctor evaluation results. **(In the benchmark-information columns, '-' denotes benchmarks extracted from real-world applications, C-I denotes cross-iteration redundancy, and C-L denotes cross-loop redundancy. M, B, C, S represent fix strategies, as discussed in Table I. Apache53622, Collections406, and Collections412 contain two inefficient loops, with two root-causes listed; MySQL27287 contains one root-cause loop that conducts two types of inefficient computation. In false-positive columns, '-' denotes no false positive, and x_y denotes real (x) and benign false positives (y) reported by LDoctor in top-5 suspicious loops of each benchmark.)**

The six cases where LDoctor and developers suggest/take different fix strategies fall into three categories. First, the fix strategy taken by developers is a subset of what suggested by LDoctor. For MySQL#27287 and Apache#53622, the root-cause loops contain both cross-loop redundancy and 0*1? inefficiency. Consequently, LDoctor suggests two fix strategies. In practice, the developers acknowledge both types of inefficiencies, but the patches only changed the data structures, which eliminated both types of inefficiencies in case of MySQL#27287 and left the cross-loop redundancy unsolved in Apache#53622.

Second, the fix strategy taken by developers is related to what suggested by LDoctor. The root-cause loops in Collections bugs #407, #408, and #425 all conduct frequent linear searches in arrays. LDoctor suggests data-structure changes for these three cases. Developers' patches still keep the original data structures, but they did use hash-sets, which contain the same content as the arrays, to help conditionally skip the loops.

Third, LDoctor cannot suggest fix strategy for 1* loops. For GCC#12322, LDoctor correctly tells that the loop under study does not contain any form of inefficiency and produce results in every iteration, and hence fails to suggest any fix strategy. In practice, GCC developers decide to skip the loop, which will cause some programs compiled by GCC to be less performance-optimal than before.

C. Accuracy Results

As shown in Table II, LDoctor is accurate, having 0 real false positive and 22 benign false positives in total for all the top five loops of the 39 benchmarks.

	LDoctor w/ optimization			w/o optimization	
BugID	**Resultless**	**C-L R.**	**C-I R.**	**C-L R.**	**C-I R.**
Mozilla347306	< 0.01%	18.29%	18.84%	355.51**X**	661.49**X**
Mozilla416628	< 0.01%	4.87%	2.41%	72.52**X**	112.15**X**
MySQL27287	5.47%	0.20%	-	263.74**X**	880.95**X**
MySQL15811	-	0.35%	-	413.59**X**	1087.16**X**
GCC46401	< 0.01%	< 0.01%	< 0.01	32.89 **X**	59.07**X**
GCC1687	-	/	1.84%	/	223.30**X**
GCC27733	< 0.01%	/	< 0.01%	/	18.44**X**
GCC8805	-	0.50%	0.19%	2.89**X**	3.36**X**
GCC21430	-	4.86%	2.37%	164.02**X**	254.9**X**
GCC12322	-	4.77%	< 0.01%	25.56**X**	24.87**X**

TABLE III: Run-time overhead of LDoctor (only non-extracted benchmarks are shown; -: dynamic analysis is not needed; /: not applicable).

Here, benign false positives mean that LDoctor analysis result is true — some loops are indeed cross-iteration/loop redundant or indeed producing results in only a tiny portion of loop iterations. However, those problems are *not* fixed by developers in their performance patches.

There are several reasons for these benign performance problems. The main reason is that they are not the main contributor to the performance problem perceived by the users. This happens to 20 out of the 22 benign cases. In fact, this is not a problem for LDoctor in real usage scenarios, because statistical debugging or profiling can often tell that these loops are not top contributors to the performance problems. Two cases happen when fixing the identified redundant/resultless problems are very difficult and hence developers decide not to fix them.

The accuracy of LDoctor benefits from its run-time analysis. For example, our run-time analysis has correctly pruned out 24 false positives in 0*1? inefficiency detection for our benchmarks. Each of these 24 loops is a top-5 suspicious loop in one of our benchmarks; it only generates side effects in its last iteration, and hence is identified as 0*1? by static analysis. Without run-time information, LDoctor would judge all of them as inefficient (0*1? resultless). Fortunately, LDoctor run-time counts the number of iterations of each loop instance and correctly identifies them as false positives. Similarly, LDoctor run-time analysis helps prune out 19 false positives for [0|1]* loop identification.

LDoctor can also help improve the accuracy of statistical debugging in identifying which loop is the root-cause loop. For example, the real root-cause loop of Apache#34464 and GCC#46401 both rank number two by the statistical performance diagnosis tool. Fortunately, LDoctor can tell that the number one loops in both cases do not contain any form of inefficiency, resultless or redundancy.

D. Performance

As shown in Table III, the performance of LDoctor is good. The overhead is consistently under or around 5% except for one benchmark, Mozilla#347306. We believe LDoctor is promising for potential production run usage. We can easily further lower the overhead through sparser sampling.

As we can also see from the table, our performance optimization discussed in Section III-B3 and III-B4 has helped.

The performance benefit of sampling is huge. Without sampling, redundancy analysis lead to over 100X slowdown for six benchmarks. The benefit of static optimization is also non-trivial. For example, for MySQL#15811 and MySQL#27287, static analysis alone can judge that they do not contain cross-iteration redundancy (i.e., no run-time overhead): the computation of each iteration depends on loop induction variables, which are naturally different in different iterations. As another example, the buggy loops of MySQL#27287 and MySQL#15811 access arrays. After changing to tracking the initial and ending memory-access addresses of the array, instead of the content of the whole array accesses, the overhead is reduced from 12.18% to 0.20% for MySQL#27287, and from 20.53% to 0.35% for MySQL#15811 respectively.

E. Parameter Setting and Sensitivity

Sampling rates We have tried different sampling rates for redundancy analysis. Intuitively, sparser sampling leads to lower overhead but worse diagnosis results. Due to space constraints, we briefly summarize the results below.

When we lower the sampling rate from 1/100 to 1/1000 in cross-loop redundancy analysis, among all the benchmarks in Table III, Mozilla#416628 incurs the largest overhead (merely 4.51%). When we lower the sampling rate from 1/1000 to 1/10000 in cross-iteration redundancy analysis, among all the benchmarks in Table III, Mozilla#347306 has the largest overhead (merely 7.38%). In both cases, the diagnosis results remain the same for all but GCC#12322, where too few samples are available to judge redundancy.

Resultful and redundancy rate Our default setting should work in most cases. In fact, the diagnosis results are largely **in**sensitive to the threshold setting. For example, the results would remain the same when changing the redundancy rate threshold from 0.5 to any value between about 0.1 and 0.66. We will have 1 more false negative and 1 fewer benign false positive, when the rate is 0.7. The trend is similar for resultless loop checking. Developers can adjust these thresholds. They can even get rid of thresholds, and only use the raw values of resultful/redundancy rates to understand the absolute and relative (in)efficiency nature of suspicious loops. Based on our experiments, the difference between efficient and inefficient loops is obvious based on these rates.

VI. Threats to Validity

LDoctor does not cover all loop inefficiency problems. For example, it does not handle performance problems caused by cache-line false sharing in multi-core machines or lock contention issues. It also cannot provide useful fixing suggestions for 1* inefficient loops, as discussed in Section V-B.

The static analysis in LDoctor could occasionally lead to inaccurate diagnosis. Specifically, LDoctor static analysis could conservatively identify some non-side-effect instructions, such as a write to a heap variable that is not used after the loop, as having side effects (Section III-A1). The SE-based

optimization discussed in Section III-B4 could cause false positives in cross-loop redundancy diagnosis. These cases are all rare in practice and not encountered in our experiments.

Our default settings of resultless and redundancy rate thresholds work well in our evaluation. However, they could potentially lead to false positives and false negatives, as discussed in Section V-E.

The results presented in Section IV and Section V should be interpreted together with corresponding methodologies and not be overly generalized. They reflect our best effort in evaluating our root-cause taxonomy and LDoctor using a non-biased set of real-world inefficient loop problems that have been perceived by users and fixed by developers. The benchmark suite covers a variety of applications, workload, development environments and programming languages. However, there are definitely uncovered cases, like problems in distributed systems and scientific computing systems, and others. Furthermore, although we did not intentionally ignore any aspect of loop-related performance problems, some loop-related problems may never be noticed by end users or never be fixed by developers, and hence may skip our evaluation. However, there are no conceivable ways to study them, particularly considering that LDoctor is designed to diagnose already-manifested performance problems not to predict not-yet-observed problems.

VII. Related Works

Profilers are the most commonly used tools that help developers understand performance [4, 6, 12, 15, 22, 29, 37]. They aim to tell where computation resources are spent, not where and why computation resources are wasted. The root-cause code region of a performance problem often is not inside the top-ranked function in the profiling result [31]. Even if it is, developers still need to spend a lot of effort to understand whether and what kind of computation inefficiency exists.

Recently proposed tools go beyond profiling. They can identify slow call-stack patterns inside event handlers [9], help understand performance causality relationship among system components [36], identify inputs or configuration entries that are most responsible for performance problems [2], and estimate the performance impact of any potential optimization at any point of a multi-threaded program [5]. They all target different problems from LDoctor.

LDoctor is most related to the recent statistical performance debugging work [31], both trying to identify source-code level root causes for user-perceived performance problems. This previous work identifies which loop is most correlated with a performance symptom through statistical analysis, but cannot answer whether or what type of inefficiency this loop contains. LDoctor complements it by accurately pointing out whether the suspicious loop is inefficient, which type of inefficiency a loop contains if any, and what is the best fix strategy.

Many dynamic and static analysis tools have been built to detect different types of performance problems, such as run-time bloat [7, 33, 34], low-utility data structures [35], cacheable data [23], false sharing in multi-thread programs [19], inefficient nested loops [24], loops with unnecessary iterations [25, 26], input-dependent loops [32].

As discussed in Section I, these tools are all useful, but are not suitable for performance diagnosis. Bug detectors are not guided by any specific performance symptoms. Consequently, they take different coverage-accuracy trade-offs from LDoctor. LDoctor tries to cover a wide variety of root-cause categories and is more aggressive in identifying root-cause categories, because it is only applied to a few loops known to be highly correlated with a specific performance symptom. Performance-bug detection tools are more conservative and try hard to lower false positive rates, because they need to process the whole program, instead of just a few loops. This requirement causes bug detectors to each focus on a specific root-cause category. Bug detectors also do not aim to provide fix suggestions. For the few that do provide [25], they only focus on very specific fix patterns, such as adding a break into the loop. In addition, dynamic performance bug detectors often lead to 10X slowdowns or more, and never tried sampling.

Automated tools have been developed to detect inefficient loop bugs [24–26]. As discussed above, these tools are good detection tools but not suitable for diagnosis. For example, Toddler [24] only targets inefficient nested loops, and hence can only cover about half of the bugs in our *general* benchmark suite. Being a dynamic tool, it also incurs 10X or more slowdowns, which is much more expensive than LDoctor. Caramel [25] statically detects inefficient loops that can be fixed by adding conditional-breaks. Very few bugs in Table I are like this. CLARITY [26] statically detects redundant traversal bugs, which arise if a program fragment repeatedly iterates over a data structure, such as an array or list, that has not been modified between successive traversals of the data structure. Like Toddler, it targets nested loops. We expect it to detect a sub-set of the cross-loop redundancy loops and a sub-set of 0*1? resultless loops in our benchmark set.

There are many test inputs generation techniques that help performance testing [3, 27, 28]. Some techniques aim to improve the test selection or prioritization during performance testing [8, 10]. All these techniques combat performance bugs from different aspects from performance diagnosis.

VIII. Conclusion

Performance diagnosis is time consuming and also critical for complicated modern software. LDoctor tries to automatically pin-point the root cause of the most common type of real-world performance problems, inefficient loops, and suggest fix strategies to developers. It achieves the coverage, accuracy, and performance goal by leveraging (1) a comprehensive root-cause taxonomy; (2) a hybrid program analysis approach; and (3) customized random sampling that is a natural fit for performance diagnosis. Our evaluation shows that LDoctor can accurately identify detailed root causes of real-world inefficient loops and suggest fix strategies. Future work can improve LDoctor by providing more detailed fix suggestions and more information to diagnose and fix 1* loops.

REFERENCES

[1] Performance bugs detected and reported in toddler project. http://www1.chapman.edu/ anistor/toddler/.

[2] M. Attariyan, M. Chow, and J. Flinn. X-ray: automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[3] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE, 2009.

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

[5] C. Curtsinger and E. D. Berger. Coz: finding code that counts with causal profiling. In *SOSP*, 2015.

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

[7] 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.

[8] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *ICSE*, 2012.

[9] 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.

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

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

[12] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *OOPSLA*, 2011.

[13] Jyoti Bansal. Why is my state's aca healthcare exchange site slow? http://blog.appdynamics.com/news/why-is-my-states-aca-healthcare-exchange-site-slow/.

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

[15] C. H. Kim, J. Rhee, H. Zhang, N. Arora, G. Jiang, X. Zhang, and D. Xu. Introperf: Transparent context-sensitive multi-layer performance inference using system stack traces. *SIGMETRICS*, 2014.

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

[17] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[18] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[19] T. Liu and E. D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. In *OOPSLA*, 2011.

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

[21] G. E. Morris. Lessons from the colorado benefits management system disaster. www.ad-mkt-review.com/publichtml/air/ai200411.html, 2004.

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

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

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

[25] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*, 2015.

[26] O. Olivo, I. Dillig, and C. Lin. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, 2015.

[27] M. Pradel, M. Huggler, and T. R. Gross. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA, 2014.

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

[29] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: mobile app performance monitoring in the wild. In *OSDI*, 2012.

[30] C. U. Smith and L. G. Williams. Software performance antipatterns. In *Proceedings of the 2nd international workshop on Software and performance*, 2000.

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

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

[33] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, 2010.

[34] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *PLDI*, 2009.

[35] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI*, 2010.

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

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