Performance Diagnosis for Inefficient Loops

Abstract

Writing efficient software is difficult. Design and implementation defects can easily cause severe performance degradation. Unfortunately, existing performance diagnosis techniques are still preliminary. Performance-bug detectors can identify specific type of inefficient computation, but are not suitable for accurately diagnosing a wide variety of real-world performance problems. Profilers can locate code regions that consume resources, but not the ones that *waste* resources. Statistical performance diagnosis can identify loops or branches that are most correlated with a performance symptom, but cannot decide whether and why a loop is inefficient, and how developers might fix it.

In this paper, we first design a root-cause and fix-strategy taxonomy for real-world 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.

1. Introduction

1.1 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 and the lack of performance testing support [3, 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.

Like general failure diagnosis, performance diagnosis starts from studying a problem symptom and hopefully ends at identifying the root cause and suggesting a fix strategy. 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 figure out a fix to the performance problem.

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. First, the inefficient code regions need to be accurately located. Second, the reason a specific region is inefficient needs to be accurately explained, so that developers can fix the problem.
- Performance. Diagnosis often requires collecting run-time information. The lower the overhead is, the easier for the diagnosis tool to be deployed, 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 static or dynamic analysis to identify code regions that match specific inefficiency patterns [7, 23–26, 33–35]. Unfortunately, these tools are not designed for and are consequently unsuitable for performance diagnosis. They are not designed to provide *coverage* for a wide variety of real-world problems; their analysis is not guided by performance failure symptom, 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 accurately 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

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

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

void mult_alg(... HOST_WIDE_UINT t, ...) {
  hash_index = t ...;
  if (alg_hash[hash_index].t == t ...)
  {
    //reuse previous results
  }
  //recursive computation
}
```

Figure 1: A real-world performance bug in GCC (the '-' and '+' demonstrate the patch)

l is inefficient and hence is not very helpful in fixing performance problems.

Figure 1 shows a performance bug in GCC. Function `mult_alg` is used to compute the best algorithm for multiplying `t`. It is so time consuming that developers used a hash-table `alg_hash` to remember which `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`, severely hurting performance. In reality, developers easily figured out that `mult_alg` is most correlated with the execution slowness, but spent several weeks to finally figure out `mult_alg`, containing 400 lines of code, is inefficient.

Clearly, more research is needed to improve the state of the art of performance diagnosis — better diagnosis coverage, better diagnosis accuracy, and low run-time overhead for common performance problems, especially loop-related performance problems.

1.2 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.

LDoctor tackles this challenging problem in three steps.

First, figuring out a root-cause taxonomy for inefficient loops. Such a taxonomy is the prerequisite to developing a general and accurate diagnosis tool. Guided by a thorough study of 45 real-world inefficient loop problems, we come up with a hierarchical root-cause taxonomy for inefficient loops that are general enough to cover common inefficient loops, and also specific enough to help developers understand and fix performance problems. More details are in Section 2.

Second, building a tool(kit) LDoctor that can automatically and accurately identify whether and how a suspicious loop is inefficient, together with fix-strategy suggestions. We achieve this following several principles:

- Focused checking. Different from performance-bug detectors that blindly checks 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.
- 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 it to see if it matches any type of inefficiency.
- Static-dynamic hybrid analysis. As we will see, static analysis alone cannot accurately identify inefficiency root causes, especially because some inefficiency problems only happen under specific workload. However, pure dynamic analysis will cause too large run-time overhead. Therefore, we take 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 code, especially inefficient loops.

We evaluated LDoctor on 18 real-world performance problems. Evaluation results show that LDoctor can accurately identify detailed root cause for all benchmarks and provide correct fix-strategy suggestion for 16 benchmarks. All of these are achieved with low run-time overhead.

2. Root-Cause Taxonomy

Previous work has identified a wide variety of performance root-cause categories. However, no existing taxonomy satisfies all the three requirements below and hence cannot be directly used for our diagnosis tool design.

1. Coverage: covering a big portion of real-world inefficient loop problems;
2. Actionability: each root-cause category should be informative enough to help developers decide how to fix a performance problem;
3. Generality: application-specific root causes will not work, as we hope to build diagnosis tools that can automatically identify root causes without developers' help.

This section first presents the root-cause taxonomy designed by us, and then discusses the coverage, actionability, and generality of our taxonomy using a suite of real-world inefficient loops collected by previous work [11, 31].

2.1 Taxonomy Design

We divide all inefficient loops into two root-cause categories: *resultless* and *redundancy*.

2.1.1 Resultless loops

A resultless loop spends a lot of time in computation without producing results useful after the loop. It is further categorized based on which loop iterations are (not) producing useful results.

```
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
```

Figure 2: A resultless 0*1? bug in Mozilla

```
+ 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;
     }
+ }
```

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

0*: This type of loops never produce any results in any iteration. They are rare in mature software systems. They should simply be deleted from the program.

0*1?: This type of loops only produce results in the last iteration, if any. They are often related to search: check a sequence of elements one by one until the right one is found. Clearly, whether these loops are efficient or not depends on the workload. When indeed inefficient, they are often fixed by data-structure changes. 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. The patch simply replaced the `script` list with a hash table.

[0|1]*: Each iteration in this type of loops may or may not produce results. For some workload, the majority of iterations do not produce results and cause performance problems perceived by users. These problems are often solved by adding a simple check to conditionally skip the expensive loop. 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. At the end, the patch greatly improved performance through a simple checking that allows the loop to be skipped in most cases.

1*: Loops in this category always generate results in almost all iterations. They are inefficient because their results are useless due to some high-level semantic reasons. Understanding and fixing this type of inefficiency problems often require deep understanding of the program and are difficult to automate. 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.

```
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
```

Figure 4: A cross-loop redundant bug in Mozilla

2.1.2 Redundant loops

Redundant loops spend a lot of time in repeating computation that is already conducted. They are further categorized below based on their redundancy unit.

Cross-iteration Redundancy: Loop iteration is the redundancy unit here: 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.

This type of inefficiency is often fixed by memoization or batching, depending on whether the redundancy involves I/O operations. For example, the GCC bug shown in Figure 1 is caused by redundant computation across different invocations of recursive function `mult_alg`. The patch essentially allows memoization to work.

Cross-loop Redundancy: A whole loop is the redundancy unit: 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. Developers often fix this type of inefficiency problems through memoization: caching the earlier computation results and skip following redundant loops. Mozilla#477564 shown in Figure 4 is an example for this type of bugs. 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.

2.2 Taxonomy Assessment

We assess the coverage, actionability, and generality of our taxonomy using a set of real-world inefficient loop problems collected by previous work [11, 31].

2.2.1 Methodology

Previous work [11, 31] studied the on-line bug databases of five representative open-source software projects, as shown in Table 2. 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

	Apache	Chrome	GCC	Mozilla	MySQL	$M_{emoization}$	$B_{atching}$	C	S	O_{ther}	Total
	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 1: The distribution among different applications and different fix strategies for bugs with each root-cause category. (S: conditionally skip the loop; C: change the data structure)

Application Suite Description (language)	# Bugs
Apache Suite HTTPD: Web Server (C) TomCat: Web Application Server (Java) Ant: Build management utility (Java)	11
Chromium Suite Google Chrome browser (C/C++)	4
GCC Suite GCC & G++ Compiler (C/C++)	8
Mozilla Suite Firefox: Web Browser (C++, JavaScript) Thunderbird: Email Client (C++, JavaScript)	12
MySQL Suite Server: Database Server (C/C++) Connector: DB Client Libraries (C/C++/Java/.Net)	10
Total	45

Table 2: Applications and bugs used in the study

the target of the study here[2]. More details can be found in previous papers that collected these bugs.

2.2.2 Assessment

Coverage As shown in Table 1, 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. In fact, no bugs in this benchmark suite belong to this category. All other root-cause sub-categories are well represented.

Actionability As demonstrated in Table 1, 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 the problem. 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. In fact, as we will discuss later, simple analysis can tell whether memoization or batching should be used to fix a redundant loop. The only problem is that there are no silver bullets for fixing 1* loops.

[2] The definition of "loop-related" in this paper is a little bit broader than earlier paper [31], which only considers 43 problems as loop-related.

Generality The root-cause categories in our taxonomy are designed to be generic. Table 1 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.

2.2.3 Caveats

Just as previous empirical study work, our empirical study above needs to be interpreted with our methodology in mind. The performance bugs examined above cover a variety of applications, workload, development environments and programming languages. However, there are still uncovered cases, like distributed systems, and scientific computation.

Previous work [11, 31] uses developers tagging and online discussion to judge whether a bug report is about performance problems and whether the performance problem under discussion is noticed and reported by users or not. We follow the methodology used in previous work [31] to judge whether the root-cause of a performance problem is related to loops or not. We do not intentionally ignore any aspect of loop-related performance problems. Some loop-related performance problems may never be noticed by end users or never be fixed by developers, and hence skip our study. However, there are no conceivable ways to study them.

3. LDoctor design

LDoctor consists of a series of analysis that judges whether a given loop belongs to any root-cause type discussed in Section 2.1. Its design follows the following principles.

- Failure diagnosis, not bug detection. LDoctor will be used together with other performance diagnosis tools [31] and focus on a small number of loops that are most correlated with a specific performance symptom, instead of being applied to the whole program. Therefore, we will have different design trade-offs in terms of coverage and accuracy, comparing with bug detection tools.
- Static-dynamic hybrid analysis. Static analysis alone will not be able to 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.

- 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.

Usage scenario Existing profilers and statistical debugging tool can identify suspicious loops that are most correlated with certain performance symptoms [31]. LDoctor will analyze these suspicious loops, identify which loop(s) is inefficient, why it is inefficient, and provide fix suggestions. This analysis could be conducted during in-house diagnosis, where failure-triggering inputs are available. It could also be conducted during production runs, with the run-time analysis component of LDoctor running at user sites.

3.1 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 belongs to 0*, 0*1?, [0|1]*, or 1*. Second, to accurately determine 0*1? and [0|1]* loops, we use dynamic analysis to figure out what portion of loop iterations are resultless at run time.

3.1.1 Static Analysis

LDoctor considers two types of instructions as side-effect instructions: (1) write to heap or global variables; (2) write to a stack variable, defined outside the loop, and the update value may be used after the loop. The second condition is 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. We consider all calls to library functions or through function pointers as having side effects, unless the library functions are specially marked by us in a white list. This analysis is complete, but not sound. We may identify side-effect instructions that are actually not.

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. In the last case, LDoctor further checks all the loop exit-blocks. If there exists an exit-block of l that contains a side-effect instruction and is post-dominated by another exit-block of l, this is a 0*1? loop. For example, line 5 in Figure 2 is such an exit block. Otherwise, l is a 0* loop.

Note that since the 1* pattern contains the least amount of information about computation *inefficiency*, LDoctor will not report a loop's root-cause type as 1*, if more informative root-cause type is identified for this loop (e.g., cross-iteration or cross-loop redundancy).

3.1.2 Dynamic Monitoring

Except for 0*, none of the other three type of loops are inefficient for sure. We need dynamic analysis to figure out what portion of loop iterations are resultless at run time, which will help decide whether the loop worths fixing.

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 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. For performance concerns, before instrumenting side-effect blocks, we check whether there are post-domination relation between each pair of side-effect blocks. If both block A and block B are side-effect blocks and block A post-dominates block B, we only instrument block A to update `HasResult`.

3.2 Redundancy Checker

3.2.1 Design overview

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 the value read by every memory read instruction in c_1 and c_2. We will know whether c_1 and c_2 are redundant with each other by comparing these values. Of course, we could do better by checking fewer instructions, as some of these values are determined by values read earlier in c_1 and/or 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 our formal definition of *input* values, and how we identify and record them in Section 3.2.2.

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 3.2.3.

How to lower the overhead of record-and-compare? We will use both static optimization (Section 3.2.5) and dynamic sampling (Section 3.2.4) to reduce the amount of data that is recorded and compared, reducing time and spatial overhead.

How to suggest the most suitable fix strategy? As discussed in Section 2.2.2 and Table 1, memoization and batching are both common fix strategies for redundant loops. To pick the right strategy, extra analysis is conducted by LDoctor and presented in Section 3.2.6.

3.2.2 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 3.1.1. It then conducts backward static slicing from these instructions, considering both control and data dependency. For every memory read instruction *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, because constant values will not affect the redundancy judgement.

Note that another possible approach is to simply record and compare what are written by the side-effect instructions in *C*. We did not choose this approach because we believe it is not sufficient to judge redundancy. Consider a loop that goes through a character array to check if there exists any special characters in the array and returns a boolean variable denoting its checking result. Running this loop on different arrays is not redundant, but could easily be judged as redundant when only comparing the loop side-effect (e.g., always returning `FALSE`).

Example Figure 5 shows a simplified code-snippet from Apache, containing a cross-loop redundancy bug. The `for` loop starting on line 7 searches a string `source` for a target sub-string `target`. Since the outer loop, which starts on line 2, appends one character to `source` in every iteration (line 30), every execution of the `for` loop is working on a similar `source` string from its previous execution, with a lot of 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 }
```

Figure 5: A cross-loop redundant bug in Apache

Take the simple loop at line 10 of Figure 5 as an example. Its only side-effect is to update `i`. For cross-loop redundancy analysis, we will consider the whole loop and identify these as *input*s: every `source[i]` read inside the loop; the initial value of `i` defined outside the loop; `max` defined outside the loop; and `first` defined outside the loop. For cross-iteration redundancy analysis, the *input*s for each iteration is slightly different: value `i` defined in previous iteration, the `source[i]` read in that iteration, and two values defined outside the loop, `max` and `first`.

Take the big loop at line 7 of Figure 5 as an example. The *input*s for the whole loop include: three values defined outside the loop (`max`, `first`, `targetLen`) and every `source[.]` and `target[.]` read inside the loop. This set is already smaller than the set of all data read by the loop. We will further shrink this set, removing invariant and others, in Section 3.2.5.

3.2.3 Identifying redundant loops

After identifying *inputs* using static analysis, LDoctor then instruments the program to record *inputs* values at run time. The run-time trace will also include delimiters and meta information that allows trace analysis to differentiate values recorded from different instructions, loop iterations, loop in-

stances, and so on. 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. We will first present our high-level algorithms, followed by the exact implementation in our prototype.

High-level algorithms For cross-iteration redundancy, we need to answer two questions. First, how to judge whether two iterations are doing redundant work? Second, is a (dynamic) loop problematic when it contains only few iterations that are redundant with each other?

Our answer to these two questions 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. Consequently, for the first question, LDoctor takes a strict definition — only iterations that are doing exactly the same computation are considered redundant. Since one iteration may not contain too much computation, a weaker definition here may lead to many false positives. For the second question, we believe there should be a threshold. In our current prototype, when the number of distinct iterations is less than half of the total iterations, we consider the loop to be cross-iteration redundant.

For cross-loop redundancy, we need to answer similar questions: how to judge whether two loop instances are doing redundant work?

Following the same principle discussed above, we have slightly different answers here. We do not require two loop instances to conduct exactly the same computation to be 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 cases where different loop instances are 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.

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 *cross-loop 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. The exact formula is the following:

$$Redundancy(I) = \frac{dist(SeqA,SeqB) - (len(SeqA) - len(SeqB))}{len(SeqA)}$$

Here, *SeqA* and *SeqB* represent the two value sequences corresponding to I from two loop instances, with *SeqA* being the longer sequence. *dist* means edit distance, and *len* means the length of a value sequence. Since the edit distance is at least the length-difference between the two sequences and at most the length of the longer sequence, we use the subtraction and division shown in the formula above to normalize the redundancy value.

3.2.4 Dynamic optimization (sampling)

Recording values returned by every input instruction would lead to huge run-time overhead. LDoctor uses random sampling to reduce this overhead, which requires almost no changes to our redundancy identification algorithm discussed in Section 3.2.3.

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 4).

Cross-iteration redundancy analysis Our high-level sampling strategy is straightforward: 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, including functions called by the loop directly or indirectly, and insert value-recording instructions along the cloned copy. We then insert a code

snippet that conducts random decision to the beginning of a loop iteration. Two variables `CurrentID`, which is initialized as 0, and `NextSampleID`, which is initialized by a random integer, are maintained in this code snippet. `CurrentID` is increased by 1 for each iteration. When it matches `NextSampleID`, the control flow jumps to the value-recording clone of the loop iteration and the `NextSampleID` is increased by a random value. Different sampling sparsity setting will determine the range from which the random value is generated.

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.

3.2.5 Static optimization

We conduct a series of static analysis to reduce the number of instructions we need to monitor.

First, we avoid tracking multiple read instructions 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, for fear of changing the location of potential exceptions, etc. 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. This analysis is sound but not complete. LDoctor may miss some loop invariants. For in-house debugging, since we know the problem-triggering inputs do not lead to exceptions, this optimization is safe. For production-run usage, users can turn off this heap/global variable related optimization.

Second, we leverage the scalar evolution analysis in LLVM to remove the monitoring to some loop-induction related variables. The scalar evolution analysis can tell which variables are loop-induction variables (e.g., `i` on line 10 of Figure 5) and what are their strides. We then use this information in two ways. One is for 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 cross-iteration redundancy without any run-time analysis. The other is for cross-loop redundancy checking. When the address of a memory 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 (i.e., the start and the end), 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.

3.2.6 Fix strategy recommendation

As discussed in Section 2.2, extra analysis is needed to decide whether batching or memoization should be suggested to fix a loop that conducts redundant computation.

For cross-iteration redundancy, batching is often used for I/O operations. Therefore, we treat I/O related redundancy separately. When the only side effect of a loop is I/O operations and the same statement(s) is executed in every loop iteration, we report this as I/O related redundancy problem and suggest batching as a potential fix strategy.

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. Instead, 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.

4. Evaluation

4.1 Methodology

Implementation and Platform We implement LDoctor in LLVM-3.4.2 [16], and conduct our experiments on a i7-960 machine, with Linux 3.11 kernel.

Benchmarks We use 18 out of the 45 bugs listed in Table 1 as our evaluation benchmarks. Among these 18, seven are extracted from Java or JavaScript programs and re-implemented in C++, as LDoctor currently only handles C/C++ programs; one is extracted from a very old version of Mozilla. The remaining bugs listed in Table 1 are much more difficult to use as benchmarks, either because they depend on special hardware/software environment or because they involve too complex data structures to extract. Overall, these 18 bugs cover a wide variety of performance root causes, as shown in Table 3.

Metrics Our experiments are designed to evaluate LDoctor from three main aspects: (1) *Coverage*. Given our benchmark suite that covers a wide variety of real-world root causes, can LDoctor identify all those root causes? (2) *Accuracy*. When analyzing non-buggy loops, will LDoctor generate any false positives? (3) *Performance*. What is the runtime overhead of LDoctor?

BugID	KLOC	P. L.	RootCause	Fix
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
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
GCC1687	2099	C	C-I	M
GCC27733	3217	C	C-I	M
GCC8805	2538	C	C-L	B
GCC21430	3844	C	C-L	M
GCC12322	2341	C	1*	S

Table 3: Benchmark information. (-: benchmarks extracted from real-world applications. "C-I": cross-iteration redundancy. "C-L": cross-loop redundancy. C, B, M, S: fix strategies as discussed in Table 1.)

BugID	Reported Root Cause	Fix Suggestion
Mozilla347306	✓	✓
Mozilla416628	✓	✓
Mozilla490742	✓	✓
Mozilla35294	✓	✓
Mozilla477564	✓	✓
MySQL27287	✓	✗
MySQL15811	✓	✓
Apache32546	✓	✓
Apache37184	✓	✓
Apache29742	✓	✓
Apache34464	✓	✓
Apache47223	✓	✓
GCC46401	✓	✓
GCC1687	✓	✓
GCC27733	✓	✓
GCC8805	✓	✓
GCC21430	✓	✓
GCC12322	✓	✗

Table 4: Coverage Results.

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 14 out of the 18 benchmarks, statistical debugging identifies the real root-cause loop as the most suspicious loop. For the remaining benchmarks, 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 provide correct fix-strategy suggestion. 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.

For all benchmarks we use, real-world users have provided at least one problem-triggering input in their on-line bug bugs. We use these inputs in our run-time analysis.

As discussed in Section 3, our analysis contains several configurable thresholds. In our evaluation, we use 0.001 as the *resultful rate* threshold for identifying 0*1? and [0|1]* resultless loops; we use 0.5 as the *redundancy rate* threshold for identifying redundant loops.

All the analysis and performance results presented below regarding cross-loop analysis is obtained using $1/100$ sampling rate; all the results regarding cross-iteration analysis is obtained using $1/1000$ sampling rate. We use sparser sampling rate in the latter case, because there tend to be more loop iterations than loop instances. All our diagnosis results require only **one** run under the problem-triggering input.

More discussions about all the parameters/thresholds presented above, including how to set them and how sensitive they are, are discussed in Section 4.5.

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

4.2 Coverage Results

Overall, LDoctor provides good diagnosis coverage, as shown in Table 4. LDoctor identifies the correct root cause for **all** 18 benchmarks, and suggests fix strategies that exactly match what developers took in practice for 16 out of 18 cases. There are only two cases where LDoctor fails to suggest the fix strategy that developers used. For MySQL#27287, the root-cause loop is both cross-loop redundant and 0*1? inefficient. LDoctor suggests both changing data structures and memoization as fix strategies. In practice, the developers find a new data structure that can eliminate both root causes. 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. However, GCC developers feel that it is worthwhile considering the slow-down caused by the original loop.

4.3 Accuracy Results

As shown in Table 5, LDoctor is accurate, having 0 real false positive and 14 benign false positives for all the top 5 loops.

BugID	0*1?	[0|1]*	C-I$_b$	C-I$_m$	C-L	Total
Mozilla347306	-	-	-	-	-	-
Mozilla416628	-	-	-	-	-	-
Mozilla490742	-	-	-	-	-	-
Mozilla35294	-	-	-	-	-	-
Mozilla477564	-	-	-	-	-	-
MySQL27287	-	0_1	-	-	-	0_1
MySQL15811	-	-	-	-	-	-
Apache32546	-	-	-	-	-	-
Apache37184	-	-	-	-	-	-
Apache29742	-	-	-	-	-	-
Apache34464	-	-	-	-	-	-
Apache47223	-	-	-	-	-	-
GCC46401	-	0_1	-	-	-	0_1
GCC1687	-	-	-	-	-	-
GCC27733	-	-	-	-	-	-
GCC8805	-	0_1	-	-	-	0_1
GCC21430	0_1	0_3	-	0_1	0_1	0_6
GCC12322	0_1	0_1	-	0_1	0_1	0_4

Table 5: False positives of LDoctor for top 5 loops reported by statistical debugging for each benchmark. '-': no false positive. x_y: real (x) and benign false positives (y).

Here, benign false positives mean that the LDoctor analysis result is true — some loops are indeed cross-iteration/loop redundant or indeed producing results in only a small portion of all the 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 11 out of the 13 benign cases. In fact, this is not really a problem for LDoctor in real usage scenarios, because statistical debugging can accurately tell that these loops are not top contributors to the performance problems. The remaining 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, there are 7 benchmarks in total that each contains a loop that generates side effect only in its last iteration. Without run-time information, LDoctor would judge all of them as inefficient (0*1? resultless). Fortunately, LDoctor run-time counts the total number of iterations and correctly identifies 3 of them as truly inefficient and severe enough to cause the corresponding performance problem.

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.

	LDoctor w/ optimization			w/o optimization	
BugID	**Resultless**	**C-L R.**	**C-I R.**	**C-L R.**	**C-I R.**
Mozilla347306	1.07%	22.40%	10.17%	304.37**X**	468.74**X**
Mozilla416628	0.80%	4.10%	2.99%	567.51**X**	85.6**X**
MySQL27287	<0.01%	1.66%	-	109.55**X**	352.07**X**
MySQL15811	-	0.03%	-	227.04**X**	424.44**X**
GCC46401	3.12%	3.80%	5.95%	21.07**X**	38.44**X**
GCC1687	-	/	<0.01%	/	142.29**X**
GCC27733	<0.01%	/	4.73%	/	17.41**X**
GCC8805	-	<0.01%	<0.01%	2.22**X**	3.52**X**
GCC21430	-	5.46%	0.69%	107.20**X**	159.89**X**
GCC12322	-	1.75%	<0.01%	21.07**X**	38.44**X**

Table 6: Run-time overhead of applying LDoctor to the buggy loop (only non-extracted benchmarks are shown). -: dynamic analysis is not needed; /: not applicable.

4.4 Performance

As shown in Table 6, 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 will discuss later, the current diagnosis results are obtained by running the program only **once** under the problem-triggering workload. If we use sparser sampling, more failure runs will be needed to obtain good diagnosis results.

As we can also see from the table, our performance optimization discussed in Section 3.2.4 and 3.2.5 has helped.

The performance benefit of sampling is huge. Without sampling, even with all the static optimization, redundancy analysis lead to over 100X slowdown for five 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: the computation of each iteration depends on loop induction variables, which are naturally different in different iterations. Consequently, run-time overhead is completely eliminated for these two benchmarks. 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 11.77% to 1.66% for MySQL#27287, and from 20.46% to 0.03% for MySQL#15811 respectively (sampling is conducted consistently here).

4.5 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, Mozilla#347306 still incurs the largest overhead (0.49%). The diagnosis results remain

the same for all but GCC#12322, where too few samples are available to judge redundancy.

When we lower the sampling rate from 1/1000 to 1/10000 in cross-iteration redundancy analysis, Mozilla#347306 has the largest overhead (4.47%). The diagnosis results remain the same for all but GCC#12322.

Resultful and redundancy rate Since the severity of resultless and redundant loops depends on workload, it is natural that LDoctor uses two thresholds in its diagnosis. In fact, the diagnosis results are largely insensitive 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.

Our default setting should work for many problems. 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 usually obvious, based on these rates.

5. Related Works

5.1 Performance Diagnosis

Profilers are the most commonly used tools that help developers understand performance [4, 6, 12, 15, 22, 29, 37]. Different from our work, profiling aims 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, tools that go beyond profiling and provide more automated analysis have been proposed. StackMine [9] identifies slow call-stack patterns inside event handlers. The work by Yu et al. [36] analyzes system traces to understand performance impact propagation and causality relationship among system components. X-ray [1] helps identify inputs or configuration entries that are most responsible for performance problems. Coz [5] can estimate the performance impact of any potential optimization at any point of a multi-threaded program. Although these techniques are all very useful, they 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 providing detailed root cause information and fix strategy suggestions.

5.2 Performance Bug Detection

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 1, these tools are all useful in improving software performance, but are not suitable for performance diagnosis. With different design goals from LDoctor, these bug detection tools 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 small number of loops that are known to be highly correlated with the specific performance symptom under study. Performance-bug detection tools has to be more conservative and tries hard to lower false positive rates, because it needs to process the whole program, instead of just a few loops. This requirement causes bug detection tools to each focus on a specific root-cause category. Performance bug detection tools 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 do not try to achieve any performance goals. None of them has tried applying sampling to their bug detection and often leads to 10X slowdowns or more.

5.3 Other Techniques to Fight Performance Bugs

There are many test inputs generation techniques that help performance testing [2, 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.

6. 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 provide fix-strategy suggestions to developers. It achieves the coverage, accuracy, and performance goal of performance diagnosis by leveraging (1) a comprehensive root-cause taxonomy; (2) a hybrid static-dynamic 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 loop problems and provide fix-strategy suggestions. Future work can further improve LDoctor by providing more detailed fix suggestions and providing more information to help diagnose and fix 1* loops.

References

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

[2] 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.

[3] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[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. Automated detection of performance bugs via static analysis. 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.