Performance Diagnosis for Inefficient Loops

Abstract

Writing efficient software is difficult. Design and implementation defects can easily cause performance problems, leading to severe performance degradation in production runs. These problems annoy end users and waste a huge amount of energy. Techniques that help diagnose and fix performance problems are desired.

Unfortunately, existing performance diagnosis techniques are still preliminary, failing to provide the desired diagnosis coverage, accuracy, and performance. Existing performance-bug detectors can accurately identify specific type of inefficient computation, but are not ideal for performance diagnosis in term of coverage and performance. Profiling tools cannot identify the code regions that *waste* the most resources, not to mention explain why some resources are wasted, failing both coverage and accuracy. Recent work on statistical performance diagnosis can identify coarse-grained root causes, such as which loop is the root cause of a performance problem. However, it cannot provide fine-grained root cause information, such as why a loop is inefficient and how developers might fix the problem.

In this paper, we first conduct an empirical study to understand what are fine-grained performance problem root causes in real world. We then design a series of static-dynamic hybrid analysis routines that can help identify accurate fine-grained performance root cause information. We further use sampling techniques to lower our diagnosis overhead without hurting diagnosis accuracy or latency. Evaluation using real-world performance problems show that our tool can provide better coverage and accuracy than existing performance diagnosis tools, with small run time 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 testing support [4, 11, 15, 21, 32, 34]. They annoy end users and waste energy during production runs, and have already caused highly publicized failures [14, 22]. Tools that can help developers quickly and accurately diagnose performance problems, find performance-wasting root causes, and fix performance problems are sorely desired.

[1] We also refer performance bugs as performance problems following previous works in this area [11, 25, 33]

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 often an execution showing severe absolute or relative slowness [33]; the root cause contains information like which code region is inefficient and why it is inefficient. An effective diagnosis tool can help developers quickly and correctly figure out a patch to fix 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. It would be good if the diagnosis tool can cover a good portion of common root causes of real-world problems.
- Accuracy. The accuracy requirement applies for two aspects of performance diagnosis: (1) the inefficient code regions need to be accurately located and (2) the reason a specific region is inefficient needs to be accurately explained, so that developers can quickly fix the performance problem.
- Performance. Diagnosis often requires collecting runtime information. The lower the overhead is, the easier for the diagnosis tool to be deployed, especially for production-run usage scenarios.

Although progress has been made, no existing tools can satisfy the above three requirements.

Profiling is the state of practice in performance diagnosis [1, 28]. It is far from providing the desired *accuracy*, as it is designed to tell where computation resources are spent, but not where and why computation resources are wasted. In many cases, the root-cause function of a performance problem may not even get ranked by a profiler [33].

Performance bug detection tools use static or dynamic analysis to identify code regions that match specific inefficiency patterns [7, 24–26, 35–37]. Unfortunately, these tools are not designed for performance diagnosis. Consequently, they are not designed to provide *coverage* for a wide variety of real-world performance problems; they do not target a specific performance symptom, and hence are at disadvantage in terms of diagnosis `accuracy`; dynamic performance-bug detection tools often lead to 10X slowdowns or more [24, 25, 37], not ideal in terms of *performance*.

Recently, progress has been made on applying statistical debugging to performance diagnosis [33]. Given the symp-

```
//expmed.c
struct alg_hash_entry {
-  unsigned int t;
+  unsigned HOST_WIDE_INT t;
};

void
synth_mult(...unsigned HOST_WIDE_INT 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)

tom of a performance problem (e.g., two similar inputs leading to very different execution speed), statistical debugging, an approach widely used for diagnosing functional failures [12, 18, 19], can accurately identify control-flow constructs that are most correlated with the performance problem by comparing problematic runs and regular runs. Unfortunately, for loop-related performance problems, which contribute to two thirds of real-world performance problems studied in previous work [11, 33], statistical debugging is not very effective. Although it can identify the root-cause loop, it does not provide any information regarding why the loop is inefficient and hence is not very helpful in fixing the performance problem.

Figure 1 shows a real-world performance bug in GCC. Function synth_mult is used to compute the best algorithm for multiplying t. It is so time consuming that developers tried speeding it up through memoization — hash-table alg_hash is used to remember which t has been processed in the past and what is the processing result. Unfortunately, a small mistake in the type declaration of hash-table entry alg_hash_entry causes memoization not to work when t is larger than the maximum value type-int can represent. As a result, under certain workload, synth_mult conducts a lot of redundant computation for same t values recursively, leading to huge performance problems. In practice, developers know the problem is inside synth_mult very early. However, since this function is complicated, it took them several weeks to figure out the root cause. If a tool can tell them not only which loop[2] is the root cause but also why a loop is inefficient (i.e., lot of redundant computation and the need for better memoization in this example), the diagnosis and bug fixing process would be much easier for developers.

Clearly, more research is needed to improve the state of the art of performance diagnosis. Diagnosis techniques that can provide better *coverage* and *accuracy* for common performance problems, especially loop-related performance problems, with good performance is well desired.

1.2 Contributions

This paper presents a tool, LDoctor, that can help effectively diagnose inefficient loop problems with good coverage, accuracy, and performance.

LDoctor targets the most common type of performance problems, inefficient loops, according to previous empirical studies [11, 33]. LDoctor also targets a challenging problem. Although statistical debugging [33] can help identify suspicious loops that are highly correlated with user-perceived performance problems, it provides no information about why the suspicious loop is inefficient, not to mention providing fix suggestions. Although various bug-detection techniques [24–26] can help detect loop-related performance bugs, they all suffer from generality problems, covering only a specific type of loop problems each; the dynamic tools [24, 25] also suffer from performance problem, imposing 10X slowdown or more.

LDoctor tackles this challenging problem in three steps.

First, figuring out a taxonomy for the root causes of common 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. Our empirical study shows that our taxonomy is general enough to cover common inefficient loops, and also specific enough to help developers understand and fix the performance problem. We will present more details in Section 2.

Second, building a tool(kit) LDoctor that can automatically and accurately identify the fine-grained root-cause type of a suspicious loop, together fix-strategy suggestions. We achieve this following several principles:

- Focused checking. Different from performance-bug detection tools that blindly checks the whole software, LDoctor focuses on performance symptoms and only check inefficient loop candidates identified by existing tools, a statistical debugging tool in our current prototype [33]. This focused study is crucial for LDoctor to achieve high accuracy.
- Root-cause taxonomy guided design. To provide the needed coverage, we follow the root-cause taxonomy discussed above and design analysis routines for every root-cause sub-category. Given a candidate inefficient loop, we will apply a series of analysis to it to see if it matches any type of inefficiency.
- Static-dynamic hybrid analysis to balance performance and accuracy. 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

[2] Recursive functions are handled similarly as loops in this paper.

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

2. Root-Cause Taxonomy

The first task we are facing is to figure out a root-cause taxonomy for real-world inefficient loops. Our taxonomy design is guided by checking a benchmark suite of real-world inefficient loops, collected by previous work [11, 33]. We also try to satisfy three requirements in our 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: too application-specific root causes will not work, as we need to build diagnosis tools to automatically identify these root causes without developers' help. In this section, we will first present our taxonomy, together with real-world examples, followed by our empirical study about how a suite of real-world inefficient loop problems fall into our root-cause categories.

We should note that we are definitely not the first one who tries to categorize the root-cause patterns of performance problems. We are also not the first to identify many of the root-cause categories below. Instead, guided by previous work and our empirical study, we try to figure out our own taxonomy that satisfy the above three requirements and can work well with the diagnosis tool that we will build later.

2.1 Taxonomy

Our taxonomy contains two major root cause categories, *resultless* and *redundancy*, and several sub-categories, as below.

Resultless Resultless loops spend a lot of time in computation that does not produce any results that will be used after the loop. Depending on when such resultless computation is conducted, we further consider four sub-categories.

0*: This type of loops never produce any results in any iteration. This type of loops should be 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: they check a sequence of elements one by one until the right one is found. Not all loops of this type are inefficient. If they are, they are often fixed by data-structure changes. Figure 2 shows an

```
//jsscript.c
jssrcnote *
js_GetSrcNote(JSScript *script, jsbytecode *pc)
{
  ...
  for (sn = SCRIPT_NOTES(script);
       !SN_IS_TERMINATOR(sn); sn = SN_NEXT(sn)) {
    offset += SN_DELTA(sn);
    if (offset == target && SN_IS_GETTABLE(sn))
       return sn;
  }
  return NULL;
}
```

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

example from Mozilla JavaScript engine. The loop searches through an array containing source code information for the input `pc`. In practice, this loop needs to execute more than 10000 iterations for many long JavaScript encountered by users, which leads to huge performance problems — web browsers freeze. To fix this problem, developers simply replace array with hash table when processing long JavaScript files.

```
// Bad input contains a long expression:
// QStringList function_list=QStringList()
// <<"abasep" <<"abs" <<"absint" ...

// c-common.c
static bool
candidate_equal_p (const_tree x, const_tree y)
{
  return (x == y) || (x && y &&
                      operand_equal_p (x, y, 0));
}

// Comments from developers:
// "No point tracking CALL_EXPRs that aren't
// ECF_CONST (because then operand_equal_p
// fails anyway) nor STRING_CSTs (which can't be
// written into)"
static void
merge_tlist(...)
{
  ...
  for (tmp2 = *to; tmp2; tmp2 = tmp2->next)
    if (candidate_equal_p (tmp2->expr, add->expr))
    {
      found = 1;
      if (!tmp2->writer)
        tmp2->writer = add->writer;
    }
  ...
}
```

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

[0|1]*: Every iteration in this type of loops may or may not produce results. Under certain workload, the majority of the loop iterations do not produce any results and lead to performance problems perceived by end users. The inefficiency problem caused by this type of loops can often be solved by adding extra conditions to skip the loop under certain contexts. Figure 3 shows such an example from GCC. Only when the if branch executed, one iteration will generate results. The bug happens when GCC checks violation of sequence point rule. The user notice severe performance degradation when enabling the checking. The buggy loop is super-linear inefficient in terms of the number of operands in one expression. The buggy input contains an expression, which is long and special. The loop takes a lot of iterations during bad run, but none of the iterations will generate results. The patch is to add extra checking before processing each operand. When it also has the special feature like the bad input, the loop will be skipped.

[1]*: Loops in this category always generate results in almost all iterations. They are inefficient, because the results generated could be 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. Although every iteration of these loops produce side effects, the performance problem forced developers to change the program and batch/skip some GUI operations.

```
//nsSessionStore.js
generate: function sss_xph_generate(aNode)
{
  ...
  for (let n = aNode; (n = n.previousSibling); )
    if (n.localName == aNode.localName
        && n.namespaceURI == aNode.namespaceURI
        && (!nName || n.name == nName))
      count++;
  //count will be used to generate a ID
  //for the aNode
  ...
}
```

Figure 4: A cross-loop redundant bug in Mozilla

Redundant Redundant loops spend a lot of time in repeating computation that is already conducted. Depending on the unit of the redundant computation, we further consider two sub-categories.

```
// nsPlacesTransactionsService.js
for (var i = 0; i < _childTransactions.length; ++i)
{
-  var txn = _childTransactions[i];
-  txn.wrappedJSObject.id = _id;
-  txn.doTransaction();
+  childTransactions[i].wrappedJSObject.id = _id;
}

+ let aggregateTxn =
+             new placesAggregateTransactions(...)
+ aggregateTxn.doTransaction();
```

Figure 5: A cross-iteration redundant bug in Mozilla

Cross-iteration Redundancy: Loop iteration is the redundancy unit here: one iteration repeats repeating what was already done by an earlier iteration of the same loop. Here, we consider recursive function calls 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, GCC#27733 shown in Figure 1 is fixed by better memoization. Mozilla#490742 in Figure 5 represents a slightly different type of cross-iteration redundancy. The inefficient loop in this case saves one URL into the "Favorite Links" database in each iteration. One database transaction in each iteration turns out to be to time consuming, with too much redundant work across iterations. At the end, developers decide to batch all database updates into one big transaction, which speeds up some workload like bookmarking 50 tabs from popping up timeout windows to not blocking.

Cross-loop Redundancy: A whole loop is the redundancy unit here: 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 is to count how many previous siblings of the input `aNode` have the same name and URI. There is an outer loop for the buggy one, and the outer loop will update `aNode` by using its next siblings. The bug is fixed by saving the calculated count for each node. A new count value is calculated by adding one to the saved count value of the nearest previous sibling with the same name and URI.

2.2 Empirical study

Having presented our taxonomy above, we will see how it works for a set of real-world inefficient loop problems collected by previous work [11, 33]. Specifically, we want to check the coverage and the actionability of our taxonomy presented above: (1) How common are real-world performance problems caused by the patterns discussed above? (2) How are real-world problems fixed by developers? Can we predict their fix strategies based on the root-cause pattern?

2.2.1 Methodology

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 1: Applications and bugs used in the study

Previous work [11, 33] studied the on-line bug databases of five representative open-source software projects, as shown in Table 1. 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[3]. More details can be found in the original papers that collect these bugs.

2.2.2 Observations

Are the root-causes in our taxonomy common? The answer is *yes*. Resultless loops are about as common as redundant loops (24 vs. 21). Intuitively, 0* loops, where no iteration produces any results, 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.

How are real-world performance bugs fixed? Overall, the fix strategies are well aligned with root-cause patterns. For example, all inefficient loops with resultless [0|1]* are fixed by skipping the loop under certain contexts, and almost all inefficient loops with redundant root causes are fixed either by memoization or batching the computation. In fact, as we will discuss later, simple analysis can tell whether memoization or batching should be used to fix a problem. The only problem is that there are no silver bullets for 1* loops.

Implications As we can see, resultless loops and redundant loops are both common reasons that cause user-perceived performance problems in practice. The root-cause categories discussed above also match with developers' fix strategy well — with a little bit amount of extra analysis, one can pretty much predict the fix strategy based on the root-cause pattern. Consequently, tools that cover these root causes will have high chances to satisfy the coverage and accuracy requirements of performance diagnosis.

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, 33] uses developers tagging and online developer/user 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 [33] 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.

We believe that the bugs in our study provide a representative sample of the well-documented and fixed performance bugs that are user-perceived and loop-related in the studied applications. Since we did not set up the root-cause taxonomy to fit particular bugs in this bug benchmark suite, we believe our taxonomy and diagnosis framework presented below will go beyond these sampled performance bugs.

[3] The definition of "loop-related" in this paper is a little bit broader than earlier paper [33], which only considers 43 problems as loop-related.

3. LDoctor design

LDoctor is composed of a series of analysis routines, each designed to identify whether a given loop belongs to one specific type of root causes, as we will present below.

As briefly mentioned in Section 1, the design of these analysis routines follows the following principles.

- Providing diagnosis information, not detecting bugs. LDoctor will be used together with other performance diagnosis tools [33] 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. As we will see, static analysis alone will not be able to provide all the needed information for performance diagnosis; dynamic analysis alone will incur too much unnecessary overhead. Therefore, we will use static-dynamic hybrid approach throughout our design.
- Using sampling to decrease run-time overhead. Loop-related performance problems have the unique nature of repetitiveness, which make them a natural fit for random sampling. Therefore, we will design different sampling schemes for different analysis routines.

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, for 0*1? and [0|1]* loops, we use dynamic analysis to figure out what portion of loop iterations are resultless at run time, which will help decide whether the loop is indeed inefficient.

3.1.1 Static Analysis

We consider side-effect instructions as those instructions that write to variables defined outside the loop. The analysis to identify side-effect instructions is straightforward. We consider all functions that are 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 library functions

	Apache	Chrome	GCC	Mozilla	MySQL	Total	Fix Strategy
Total # of loop-related bugs	11	4	8	12	10	45	
			# of *Resultless* bugs				
0*	0	0	0	0	0	0	
0*1?	0	0	0	2	3	5	C(4)|S(1)
[0|1]*	0	1	1	1	1	4	S(4)
1*	1	2	3	6	3	15	B(4)|S(4)|O(7)
			# of *Redundant* bugs				
Cross-**iteration** redundancy	7	1	2	1	1	12	B(4)|M(7)|O(1)
Cross-**loop** redundancy	3	0	2	2	2	9	B(4)|M(5)

Table 2: Number of bugs in each root-cause category. B, M, S, C, and O represent different fix strategies: B(atching), M(emoization), S(kipping the loop), C(hange the data structure), and O(thers). The numbers in the parentheses denote the number of problems that are fixed using specific fix strategies.

or function calls through function pointers as functions that have side effects, unless the library functions are specially marked by us in a white list.

After identifying side-effect instructions, it is straightforward to categorize loops into the four types discussed above. Loop 0* contains no side-effect instructions. Loop 1* contains at least one side-effect instruction along every path that starts from the loop header and ends at the loop header. The remaining cases are either 0*1? or [0|1]*. Differentiating these two cases is also straight-forward. In short, when the basic block that contains side-effect instructions is part of the natural loop, the case belongs to [0|1]*; instead, if the side-effect basic block is post-dominated by one of the loop-exit blocks and is dominated by the loop header, yet is not part of the natural loop, the case belongs to 0*1?.

Finally, 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 is indeed the root cause of a user-perceived performance problem and worth fixing.

For a 0*1? loop, since it only generates results in the last iteration, we only need to know the total number of iterations (or the average total number of iterations when the loop has multiple instances) to figure out the *resultless rate* of the loop. 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`.

We could speed up the above counting using sampling. However, since the run-time overhead of the above counting is low, as shown in Section 4, our current prototype of LDoctor does not use sampling for this part of run-time analysis.

3.1.3 Limitations

The technique designed in this section has the following limitations. First, when callee may have side effect, we will consider it will have side effect in the caller side, and do not consider the real execution inside callee. This could bring false negatives, because we could miss resultless cases, where side effect instructions inside callee do not execute. Experiments results in Section 4 show that this is not a big issue, since we do not miss any resultless bugs.

Second, our dynamic instrumentation does not consider concurrent execution of the monitored loop, because most of buggy loops we study only execute in one single thread. When the monitored loop is executed in multi-thread, like loop marked with omp pragma, we need to synchronize updates to global variables.

3.2 Redundancy Checker

3.2.1 Design overview

To check whether there is redundant computation across different iterations of one loop instance or across different instances of one static loop, we need to address several challenges.

How to judge redundancy between two iterations/loop-instances? Given two iterations (or loop-instances) i_1 and i_2,

```
0    //java.lang.String
1    static int
2    indexOf(char[] source, int sourceCount,
3            char[] target, int targetCount )
4    {
5      ...
6      char first  = target[targetOffset];
7      int max = sourceCount - targetCount;
8
9      for (int i = 0; i <= max; i++) {
10       // Look for first character.
11       if (source[i] != first) {
12         while (++i <= max && source[i] != first);
13       }
14
15       // Found first character
16       //now look at the rest of v2
17       if (i <= max) {
18         int j = i + 1;
19         int end = j + targetCount - 1;
20         for (int k = 1; j < end && source[j] ==
21                 target[k]; j++, k++);
22
23         if (j == end) {
24           /* Found whole string. */
25           return i ;
26         }
27       }
28     }
29     return -1;
30   }
31
32   //TelnetTask.java
33   public void waitForString(...)
34   {
35 +   int windowIndex = -s.length();
36 -   while (sb.toString().indexOf(s) == -1) {
37 +   while (windowIndex++ < 0 ||
38 +    sb.substring(windowIndex).indexOf(s) == -1){
39       sb.append((char) is.read());
40     }
41   }
```

Figure 6: A cross-loop redundant bug in Apache

since they have the same source code, a naive, yet expensive, solution is to record and compare the return value of every memory read conducted by i_1 and i_2. Two better alternative solutions are to record and compare only the values written or read by the side-effect instructions, such as line 9 in Figure 4, or the source instructions[4], such as source[i] at line 12 of Figure 6.

Among the the above two potential solutions, our design chooses the second one. The reason is that, if there is indeed redundancy, repetitive patterns at the side-effect instructions are caused by repetitive patterns in the source instructions. For the purpose of performance diagnosis, it is more informative to track the cause, rather than the effect.

How to handle partial redundancy? In practice, redundant loops may be doing largely the same, instead of exactly the same, computation across iterations or loop instances. It is also possible that only some, instead of all, iterations in a loop are doing redundant computation. We will discuss how we handle this issue in Section 3.2.3.

[4] We define source instructions in a code region r as a set of memory-read instructions that side-effect instructions in r depend on and do not depend on any other instructions inside r.

How to lower the overhead of record-and-compare? Even if we only record and compare the values returned by source instructions, instead of all instructions in a loop, the run-time overhead and the log size would still be large. We will use two ways to lower this time and spatial overhead. First, static analysis can already tell some source instructions will always return the same value across iterations/loop-instances, and hence need not be traced at run time. Second, we can leverage the repetitive nature of performance problems and use random sampling to lower the overhead without degrading the diagnosis capability. We will discuss details of these two optimization in Section 3.2.4 and 3.2.5.

How to provide the most suitable fix-strategy suggestion? Finally, as discussed in Section 2.2.2 and Table 2, memoization and batching are both common fix strategies for redundant loops. To pick the right fix strategies to fix a redundant loop, we will conduct some extra analysis. We will discuss this in Section 3.2.6.

3.2.2 Identifying source instructions

Informally, we use static analysis to identify a set of memory-read instructions that the loop computation depends on. We refer to these instructions as *source* instructions. The values returned from them at run-time will be tracked and compared to identify redundant computation.

Specifically, we first identify side-effect instructions in the loop, as discussed in Section 3.1.1; we then conduct static slicing from these side-effect instructions, considering both control and data dependency, to identify source instructions.

Our slicing ends when it reaches either a local-variable read conducted outside the loop or a heap/global-variable read anywhere in the program. For the latter case, our slicing stops because tracking data-dependency through heap/global variables is very complicated in multi-threaded C/C++ programs. For the former case, not including local-variable reads inside the loop can help us reduce the amount of data that needs to be recorded. When there are function calls inside the loop, we conduct slicing for return values of callees and side-effect instructions inside callees. We omit encountered constant values through slicing, because constant values will not influence whether a loop or an iteration is redundant.

The analysis for cross-iteration and cross-loop redundancy analysis is pretty much the same. The only difference is that, if a memory read instruction i in a loop depends on the value returned by instruction j in an earlier iteration, we stop tracing the dependence at i and consider i as a source instruction for cross-iteration redundancy analysis, while we continue the slicing for cross-loop redundancy analysis. For example, the instruction defining the value of `i` is the only side-effect instruction for the loop at line 12 of Figure 6. The source instructions calculated by cross-loop dependence analysis include memory read `source[i]` inside the loop, and three values defined outside the loop, which represent the initial value of `i`, the value of `max` and `first` respectively. In contrast, the source instructions calculated by cross-iteration dependence analysis include value `i` defined in previous iteration, memory read `source[i]`, and two values defined outside the loop (`max` and `first`).

3.2.3 Identifying redundant loops

After identifying source instructions, we instrument these source instructions, so that the values returned by these memory read instructions can be recorded at run time. Specifically, we will assign a unique ID for each source instruction, and a pair of $< InstID, Value >$ will be recorded at run time with the execution of a source instruction. Our trace also includes some delimiters and meta information that allows trace analysis to differentiate values recorded from different loop iterations, different loop instances, and so on.

After collecting values returned by source instructions from every iteration of one or multiple loop instances, we need to process the trace and decide whether the loops under study contain cross-iteration redundancy 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 — should the two iterations conduct exactly the same computation? 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 root-cause for a user-perceived performance problem and to make itself worthwhile to get optimized by the 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 iterating 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 is cross-iteration redundant.

For example, Figure 7 shows a loop from Apache-Ant that contains cross-iteration redundancy. As we can see, under the problem triggering input, only several distinct listeners are contained inside the vector, and most of iterations of the loop inside function `fireMessageLoggedEvent` are doing exactly the same computation.

For cross-loop redundancy, we need to answer similar questions, especially how to judge whether two loop instances are doing redundant work — should they contain exactly the same number of iterations and doing exactly the same computation in each iteration?

Our answers here are different from our answers above for cross-iteration redundancy analysis. We do not require

```
//Project.java
public synchronized void
addBuildListener(BuildListener listener) {
+  if (!newListeners.contains(listener)) {
     newListeners.addElement(listener);
+  }
}

private void
fireMessageLoggedEvent(...) {
   ...
   while (iter.hasNext()) {
     BuildListener listener
          = (BuildListener) iter.next();
     listener.messageLogged(event);
   }
}
```

Figure 7: A cross-iteration redundant bug in Apache

two loop instances to conduct exactly the same computation to be identified as redundant. The rationale is that a whole loop instance contains a lot of computation, much more than one iteration in general. Consequently, 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 fact, in practice, we almost have never seen cases where different loop instances are doing exactly the same computation. For example, Figure 4 demonstrates a cross-loop redundancy problem in Mozilla. Here, the latter instances contain more iterations than previous instances. Figure 6 shows an example in Apache, The inner loop, which starts from line 9, searches from the beginning of a string `sb` for a target sub-string `s`. Since the outer loop, which starts on line 36, appends one character to `sb` in every iteration, every inner loop instance is doing computation that is similar, but not exactly the same, from its previous instance.

Detailed algorithm implementation The implementation of checking cross-iteration redundancy is straightforward. We will record a sequence of $< InstID, Value >$ pair for every monitored iteration, with each *InstID* representing a unique source instruction. We consider two iterations to be redundant, if their sequences are exactly the same. To make sure a loop contains sufficient redundant computation, we calculate a loop's *cross-iteration redundancy rate* — dividing the total number of iterations in the loop by the number of distinct iterations. The smaller the rate is, with 1 being the minimum possible value, the less cross-iteration redundancy the loop contains.

The implementation of 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 — dividing the number of redundant pairs by $k-1$. The smaller the rate is, with 0 being the minimum possible value, the less cross-loop redundancy L contains. Here we only check redundancy between consecutive loop instances, because checking the redundancy between every pairs of loop instances would be very 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 number of iterations; in different iteration, a different set of source instructions may have executed. Therefore, we cannot simply merge values from different source instructions and iterations together and compare two big data sequence. Instead, we decide to check the redundancy for each source instruction across l_1 and l_2 first, and then use the average *redundancy rate* of all source instructions as the *cross-loop redundancy rate* between l_1 and l_2.

We calculate the redundancy for one source 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:

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

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 performance optimization: sampling

Recording values returned by every source instructions would lead to huge run-time time. To lower the overhead, we use random sampling to reduce the number of instructions that we track at run time. Due to the repetitive nature of performance bugs, we will still be able to recognize redundant computation as long as the sampling rate is not too sparse (we will evaluate this in Section 4). Our sampling scheme requires almost no changes to our redundancy identification algorithm discussed in Section 3.2.3.

Cross-iteration redundancy analysis Our high-level sampling strategy is straightforward: we will make a random decision at the beginning of every iteration about whether to track the values returned by source instructions in this iteration or not.

The implementation is similar with previous sampling work [18, 19]. 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 At high-level, we make a random decision at the beginning of every loop instance about whether to track values for this instance or not. 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 immediately next loop instance too.

The implementation is similar with that for cross-iteration redundancy analysis. The only difference includes: (1) clone is made for the whole loop and the code snippet that controls sampling is added to the pre-header of the loops; (2) sampling is conducted when `CurrentID` equals either `NextSampleID` or `NextSampleID+1`, with `NextSampleID` increased by a random value in the latter case.

Handling recursive functions We also conduct sampling to our redundancy analysis for recursive functions. For a recursive function, we first create an instrumented clone copy of the whole function body. We then add a sampling-control code snippet at the entry point of the function, where `CurrentID` and `NextSampleID` are maintained to decide whether to execute the original function body or the instrumented value-recording clone copy. We also create clones for all the callee functions of the recursive function under study, so that the sampling decision can be correctly conducted throughout the call chain.

3.2.5 Static analysis for performance optimization

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

First, we identify and avoid monitoring memory reads whose return values can be statically proved to not change throughout one loop instance (i.e., for cross-iteration redundancy analysis) or multiple loop instances (i.e., for cross-loop redundancy analysis). Since we implement LDoctor at LLVM byte-code level, a major part of this analysis is already done by LLVM, which lifts loop-invariant memory accesses out of loops. The only extra analysis we did is to prune memory reads whose reading address is loop invariant, and there are no writes inside the loop which are possibly conducted on the same address. For example, the read address of `aNode.localName` and `aNode.namespaceURI` in Figure 2 is loop-invariant, and there are no write conducted on the same address. We do not need to record these two memory read.

Second, we identify and avoid monitoring some memory reads whose return values can be statically proved to be different throughout one loop instance in cross-iteration redundancy analysis. Specifically, for read on loop induction variable, such as i for loop at line 12 of Figure 6, we also know for sure that their return values are different in different iterations. If source instructions include read on loop induction variables, we know that there could not be cross-iteration redundancy. We use scalar evolution analysis provided by LLVM to identify induction variables and avoid tracking their values.

Third, sometimes we only record the memory-address range of a sequence of memory read, instead of the value returned by every read, in cross-loop redundancy analysis. The loop at line 12 in Figure 6 shows an example. The content of array `source` is never modified throughout the outer-loop, which starts at line 12 in the Figure 6. Therefore, to check whether different inner loop instances read similar sequence of array data, we only need to record the starting and ending array index touched by each inner loop, significantly reduce the monitoring overhead. To accomplish this optimization, we again leverage the scalar evolution analysis provided by LLVM. The scalar evolution analysis tells us whether the address of a memory read instruction is a loop induction variable. For example, the address for memory read `source[i]` is added by one in each loop iteration, so it is a loop induction variable. From the scalar evolution analysis, we know that the starting address is `source` plus the initial value of `i`, and the ending address is source plus the ending value of `i`.

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 to fix redundancy that involves I/O related operations, based on our empirical study. Therefore, we treat I/O related redundancy separately. Specifically, 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 to use. LDoctor currently 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 the buggy loop in Figure 6, we recommend memoization. Instead, if the side effect is updating an sequence of memory locations, with the number of locations increasing with the workload, memoization is unlikely to help save much computation.

4. Evaluation

4.1 Methodology

Implementation and Platform We implement LDoctor in LLVM-3.4.2 [17], and conduct our experiments on a i7-960 machine, with Linux 3.11 kernel.

BugID	KLOC	P. L.	Root Cause	Fix
Mozilla347306	88	C	0*1?	C
Mozilla416628	105	C	0*1?	C
Mozilla490742	N/A	JS	C-I	B
Mozilla35294	N/A	C++	C-L	B
Mozilla477564	N/A	JS	C-L	M
MySQL27287	995	C++	0*1?,C-L	C
MySQL15811	1127	C++	C-L	M
Apache32546	N/A	Java	C-I	B
Apache37184	N/A	Java	C-I	M
Apache29742	N/A	Java	C-L	B
Apache34464	N/A	Java	C-L	M
Apache47223	N/A	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. (N/A: we skip the size of benchmarks that are extracted from real-world applications. Root cause "C-I" is short for cross-iteration redundancy. Root cause "C-L" is short for cross-loop redundancy. C, B, M, and S represent different fix strategies. C is short for Change the data structure. B is short for Batch. M is short for Memoization. S is short for Skip the loop.)

Benchmarks We use 18 out of the 45 bugs listed in Table 2 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. These 18 bugs include all the benchmarks used in the recent statistical performance debugging work [33]. The remaining bugs listed in Table 2 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:

- Coverage. Given our benchmark suite that covers a wide variety of real-world root causes, can LDoctor identify all those root causes?
- Accuracy. When analyzing non-buggy loops, will LDoctor generate any false positives?
- Performance. What is the run-time overhead of LDoctor?

Evaluation settings The imagined usage scenario of LDoctor is that one will apply LDoctor to identify detailed root causes and provide fix-strategy suggestion for a small number of suspicious loops that are most correlated with the specific performance problem.

In our evaluation, we leverage previous statistical performance debugging work [33] to preprocess a performance problem and identify one or a small number of suspicious loops for LDoctor to analyze. For 14 out of the 18 benchmarks, statistical performance debugging identifies the real root-cause loop as the top ranked suspicious loop. For the remaining 4 benchmarks, the real root-cause loops are ranked as number 2, 2, 4, and 10. Overall, we believe future tools can accurately identify the most one or a couple of suspicious loops.

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 the correct fix-strategy suggestion (Section 4.2). Second, we apply statistical performance debugging [33] to all our benchmarks and apply LDoctor to the top 5 ranked loops[5] 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 *resultless rate* threshold for identifying 0*1? loops, 0.01 as the *resultless rate* threshold for identifying [0|1]* loops, 0.5 as the *cross-loop redundancy rate*, and 2 as the *cross-iteration redundancy rate* (i.e., the number of distinct iterations is less than half of the total iterations).

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.

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 MySQL27287, 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 GCC12322, 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 performance impact of the original loop to the GCC compilation process. Providing this type of fix strategy suggestion goes beyond the capability of LDoctor.

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

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

Table 4: Coverage Results.

4.3 Accuracy Results

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

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 12 out of the 14 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

BugID	0*1?	[0|1]*	C-I$_b$	C-I$_m$	C-L	Total
Mozilla347306	-	-	-	-	-	-
Mozilla416628	-	-	-	-	-	-
Mozilla490742$_a$	-	-	-	-	-	-
Mozilla35294$_a$	-	-	-	-	-	-
Mozilla477564$_a$	-	-	-	-	-	-
MySQL27287	-	0_1	-	-	-	0_1
MySQL15811	-	-	-	-	-	-
Apache32546	-	-	-	-	-	-
Apache37184$_a$	-	-	-	-	-	-
Apache29742$_a$	-	-	-	-	-	-
Apache34464	-	-	-	-	-	-
Apache47223	-	-	-	-	-	-
GCC46401	-	0_1	-	-	-	0_1
GCC1687	-	-	-	-	-	-
GCC27733$_a$	-	-	-	-	-	-
GCC8805	-	0_2	-	-	-	0_2
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, when applying to top 5 loops reported by statistical performance diagnosis for each benchmark. '-' represents zero false positive. Other cells report real false positives and benign false positives, which is in the subscript.

BugID	0*1?	[0|1]*
Mozilla347306	-	-
Mozilla416628	-	-
Mozilla490742$_a$	-	-
Mozilla35294$_a$	-	-
Mozilla477564$_a$	-	1_0
MySQL27287	-	0_1
MySQL15811	-	-
Apache32546	-	-
Apache37184$_a$	-	-
Apache29742$_a$	-	-
Apache34464	-	1_0
Apache47223	-	-
GCC46401	-	2_1
GCC1687	1_0	-
GCC27733$_a$	2_0	-
GCC8805	1_0	0_2
GCC21430	0_1	0_3
GCC12322	1_1	1_1

Table 6: False positive results when only applying static resultless loop analysis for the top 5 loops reported by statistical performance diagnosis. The meaning of '-'s and numbers in the table is the same as that in Table 5.

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. Table 6 shows the false positive numbers of resultless loop analysis with only static analysis. As we can see, there are quite many loops among the top 5 suspicious loops that only generate side-effect in the last iteration or so. However, many of these loops are actually not inefficient because the portion of resultless iterations is small.

The good accuracy of LDoctor can actually help improving the accuracy of identifying which loop is the root cause loop. For example, the real root-cause loop of Apache34464 and GCC 46401 both rank number two by the statistical performance diagnosis tool. LDoctor can tell that the number one loops in both cases do not contain any form of inefficiency, resultless or redundancy. This result can potentially used to improve the accuracy of identifying root cause loops.

4.4 Performance

As shown in Table 7, the performance of LDoctor is very good. The overhead is consistently under or around 5% except for one benchmark, Mozilla347306. We believe LDoctor is promising for potential production run usage. Of course, if we apply LDoctor to multiple loops at the same time, the overhead will be higher. However, the current results are obtained by running the program only **once** under the problem-triggering workload. The sampling nature of LDoctor will allow us to keep the overhead low at the ex-

	LDoctor			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	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	/	142.29**X**
GCC27733$_a$	~0	/	4.73%	/	17.41**X**
GCC8805	-	~0	~0	2.22**X**	3.52**X**
GCC21430	-	5.46%	0.69%	107.20**X**	159.89**X**
GCC12322	-	1.75%	~0	21.07**X**	38.44**X**

Table 7: Run-time overhead of applying LDoctor to the buggy loop. Only performance results from non-extracted benchmarks are shown here. Not: not applying the performance optimization in LDoctor. We present run-time slowdown for cases where the performance optimization is not applied. -: static analysis can figure out the results and hence no dynamic analysis is conducted. /: not applicable.

change of running the program for a couple of more times, if needed.

As we can also see from the table, our performance optimization discussed in Section 3.2.4 and 3.2.5 has contributed a lot to the good performance of LDoctor.

Without sampling, while still applying our static optimization, our redundancy analysis would lead to over 100X slowdown for five benchmarks.

The buggy loops of MySQL#27287 and MySQL#15811 sequentially access the content of an array. 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).

The side-effect of the buggy loop for MySQL15811 is to calculate the length of a string. The variable representing the length is an induction variable. The side-effect of the buggy loop for MySQL27287 is to calculate the index of a searched target, and the variable representing the index is also an induction variable. We can reply on static analysis to figure out that these two loops are not cross-iteration redundant.

We also tried sampling with different sampling rates. For cross-loop redundancy, we also conduct experiments under sampling rate 1 out of 1000. Both run-time overhead and collected sample will be reduced. Mozilla347306 is still the benchmark with largest overhead, but the overhead is reduced to 0.49%. For two benchmarks, GCC8805 and GCC12322, we cannot sample more than 10 loop instances and hence cannot draw strong conclusion about their root-cause type. For all other benchmarks, we can still have more than 10 loop instances and get exactly the same root-cause analysis results presented above.

We also conduct cross-iteration redundant experiment under different sampling rates. When the sampling rate is 1 out of 100, there are two benchmarks, whose run-time overhead is larger than 30%. The overhead for Mozilla347306 is 59.57%, and the overhead of GCC21430 is 30.65%. When changing sampling rate to 1 out of 10000, Mozilla347306 has the largest overhead, which is 4.47%. And mozilla347306 is the only one whose overhead is larger than 2%. Except for GCC12322, all other benchmarks will have more than 100 iterations sampled, under the sample rate is 1 out of 10000. Consequently, the same root-cause analysis results will be reported for these benchmarks as the ones presented above.

5. Related Works

5.1 Performance Diagnosis

System developers always spend a lot of time in understanding software performance problems. Many tools and techniques have been built before to help developers.

Profilers are the most commonly used tools that help developers understand system performance. Even after decades of research, people are still making significant progresses in this area recently. Some recent work helps associate complexity with bottleneck functions [5, 39]; some improves the profiling of calling context trees [6], some tries to provide more precise profiling results [23], and some targets new types of applications, like asynchronous programs [31] and interactive programs [13]. 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 [33]. 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. IntroPerf [16] can help automatically infer the latency of function calls based on operating system tracers. StackMine [9] can identify slow call-stack patterns inside event handlers. The work by Yu et al. [38] analyzes system traces to understand performance impact propagation and performance causality relationship among different system components. X-ray [2] helps identify inputs or configuration entries that are most responsible for performance problems. Although these techniques are all very useful, they target different problems from LDoctor. LDoctor targets the inefficient loop problem, a common type of computation inefficiency problem that contributes to about two thirds of real-world user-perceived performance problems studied by previous work [33], and tries to pinpoint the exact root cause and provide fix-strategy suggestions.

LDoctor is most related to the recent statistical performance debugging work [33], both trying to identify source-code level root causes for user-perceived performance problems. This previous work can only identify which loop is most correlated with a user identified performance symptom through statistical analysis, but cannot provide any information regarding what type of inefficiency this loop contains. LDoctor complements it by providing detailed root cause information and fix strategy suggestions through program analysis.

5.2 Performance Bug Detection

Bug detection aims to find previously unknown defects inside programs that can lead to unnecessary performance degradation.

Many dynamic tools have been built to detect run-time bloat [7, 35, 36], low-utility data structures [37], cacheable data [24], performance slowdown caused by false sharing in multi-thread programs [20], inefficient nested loops [25], input-dependent loops [34], and others. Static tools have also been proposed. Some static tools identify loops that are executing unnecessary iterations and hence could have been sped up by some extra `break` statements [26, 27]. Some static tools [11] identify performance defects that violate specific efficiency rules, often related to API usage.

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 apply its analysis to 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 [26], 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. Wise [3] learns how to restrict conditional branches to get worst-case complexity from small inputs, and uses the learned policy to generate worst-case complexity inputs with larger size. EventBreak [30] generates performance testing inputs for web applications. SpeedGun [29] generates performance regression tests for changed concurrent classes.

Some techniques aim to improve the test selection or prioritization during performance testing. Forepost [8] is a test selection framework toward exposing more performance bottlenecks. It run applications under a small set of randomly selected test inputs to learn rules about which types of inputs are more likely to trigger intensive computation. Then it use learned rules to pick remaining test inputs in order to expose more performance bottlenecks. Huang et al. [10] study 100 performance regression issues. Based on the studying results, they design a performance risk analysis, which can help developers prioritize performance testing.

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] http://sourceware.org/binutils/docs/gprof/.

[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 '09, pages 463–473, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. . URL `http://dx.doi.org/10.1109/ICSE.2009.5070545`.

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

[5] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, 2012.

[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 *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 156–166, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL `http://dl.acm.org/citation.cfm?id=2337223.2337242`.

[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 *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 60–71, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. . URL `http://doi.acm.org/10.1145/2568225.2568232`.

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

[12] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, 2002.

[13] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, 2011.

[14] 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/.

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

[16] 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.

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

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

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

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

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

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

[23] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of java profilers. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, 2010.

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

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

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

[27] O. Olivo, I. Dillig, and C. Lin. Automated detection of performance bugs via static analysis. In *PLDI*, 2015.

[28] OProfile. OProfile – A System Profiler for Linux. http://oprofile.sourceforge.net.

[29] 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, pages 13–25, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2645-2. . URL `http://doi.acm.org/10.1145/2610384.2610393`.

[30] M. Pradel, P. Schuh, G. Necula, and K. Sen. Eventbreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '14, pages 33–47, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. . URL `http://doi.acm.org/10.1145/2660193.2660233`.

[31] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

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

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

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

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

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

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

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

[39] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, 2012.