Algorithmic Profiling for Real-World Complexity Problems

Anonymous Author(s)

ABSTRACT

Complexity problems are a common type of performance problems, caused by algorithmic inefficiency. Algorithmic profiling aims to automatically attribute execution complexity to each executed code construct. It can potentially detect previously unknown complexity problems and diagnose performance failures caused by complexity problems. However, existing algorithmic profiling techniques suffer from several severe limitations, missing opportunities to be deployed in production environment and effectively guide developers to understand and fix complexity problems.

In this paper, we design a tool, ComAir, which can effectively conduct algorithmic profiling under both production-run and in-house settings. Under the guidance of an empirical study on real-world complexity problems, we propose several novel instrumentation methods to significantly lower runtime overhead for the production-run version of ComAir. We also design an effective ranking mechanism to help developers identify root causes for performance failures due to complexity problems. Our experimental results show that ComAir can effectively identify root causes and generate accurate profiling results under the two settings, and it incurs a negligible overhead under the production-run setting with the capability to be really deployed in production environment.

1 INTRODUCTION

As a type of software bug, performance problems[1] are one major source of software's slowness and inefficiency [17, 21, 35, 40, 41]. Performance problems widely exist in production-run software, leading to poor user experience and even economic loss in the field [17, 40, 41]. Several highly-publicized failures have already been caused by performance problems, such as making a website costing millions of dollars useless [20]. Therefore, combating performance problems is urgent.

Many performance bugs are caused by algorithmic inefficiency, such as implementing a linear algorithm in an $O(N^2)$ way, which cannot be optimized away by state-of-the-art compiler optimizations. We refer to these performance problems as *complexity problems* in this paper. Our empirical study on a representative performance-bug benchmark set [17, 40] shows that nearly half of the user-perceived performance problems that *occurred in production environment* are complexity problems. In addition, complexity problems are usually ranked as high-priority bugs in software development. For example, Mozilla developers will immediately try to fix complexity bugs degrading exponentially [30].

Given the above discussion, addressing complexity problems in software production runs is an important aspect of fighting performance bugs. One commonly used method to diagnose complexity problems is through algorithmic profiling [9, 10, 52], which collects profiles from multiple executions of the same program and attributes complexity to different code constructs, such as a loop or a function, in the form of a *cost* function of *input* size. Therefore, algorithmic profiling can potentially be used to detect complexity problems and pinpoint the root causes for performance failures caused by complexity problems.

```
1  //items is an array containing all XML_NODEs
2  //nitems is the index for the input XML_NODE
3  uint xml_parent_tag(XML_NODE * items, uint nitems) {
4    XML_NODE * p, * last = &items[nitems-1];
5    uint level = items[nitems].level - 1;
6    //parent's level is one less than the input node
7    for(p = last; p >= items; p--) {
8      if(p->level == level && p->type == XML_NODE_TAG) {
9        return p - items;
10     }
11   }
12   return 0;
13 }
```

Figure 1: A MySQL performance problem in polynomial complexity. ***During performance failure runs, the execution time scales polynomially with the size of* `items`.**

One MySQL complexity problem is shown in Figure 1. The loop searches parent `XML_NODE` for function parameter `nitems`, which is used as an array index. All `XML_NODE`s are maintained in the array `items`. The loop iterates through the array `items` backwardly and searches for the first `XML_NODE` with `level` one less than the `XML_NODE` indexed by `nitems` (lines 7–11). This piece of code looks innocent. However, there is an outer loop not shown in the figure. The outer loop will keep calling `xml_parent_tag` using the next sibling of the previous `XML_NODE`, which has $O(N^2)$ complexity in terms of the number of children of a parent `XML_NODE`. Developers may think that using an implementation with $O(N^2)$ complexity is fine, since an `XML_NODE` usually does not have too many children. However, during performance failure runs, an `XML_NODE` contains tens of thousands of children, and this leads significant showdown perceived by the end users.

To fix this bug, the developers added an extra field to each `XML_NODE` to save its parent, and this field is initialized when an `XML_NODE` is created. After applying this patch, the code shown in Figure 1 was completely removed. However, it took developers around 5 months to figure this out.[2] Production-run algorithmic profiling can point out the complexity of this piece of code is $O(N^2)$, which is exactly the root cause, and can also provide workload information to confirm the necessity to fix this bug.

Existing techniques are not capable of handling production-run complexity problems efficiently and effectively due to several reasons. First, many algorithmic profilers [9, 10, 52] and performance testing tools [13, 35] are designed for in-house usage and cannot be deployed in production environment due to heavy overhead. Recent techniques on algorithmic profiling [9, 10, 52] can incur more than 30× runtime overhead. Second, in-house profilers often provide limited knowledge of real-world workload, which is necessary to determine if a complexity problem can cause user-perceived performance impact. A previous empirical study shows that workload keeps changing and developers usually do not have a good understanding of real-world workload [17]. Third, existing profilers intended for production usage [15, 19, 22, 38] can measure

[1]We will use performance problems, performance issues, and performance bugs exchangeably, following previous works in this area [40, 41].

[2]We count the time from when developers confirmed this is performance bug to when the patch was submitted.

only how much time is spent in each code construct during one single run, but fail to synthesize information from multiple runs or provide any indication about how the execution time scales.

The goal of our research is twofold. First, we want to better understand complexity problems, including their root causes, how user-perceived performance impact in production environment is generated, their categories, and how the bugs in each category are fixed. Toward this end, we conduct an empirical study on a representative performance-bug benchmark suite [17, 40], containing 65 user-perceived performance bugs from five real-world applications. We discovered that 1) around three-fourths of the studied complexity problems are caused by repeated executions of a loop or a recursive function; 2) for most complexity problems, the users provide indications about how to change input sizes to reproduce the scaling problems during reporting; and 3) complexity problems usually take a longer time to diagnose and fix, and more effective tool supports are needed. To the best of our knowledge, our work is the first study focusing on complexity problems.

Second, guided by the findings from the empirical study, we develop ComAir, the first automated tool to effectively and efficiently conduct algorithmic profiling under production-run setting. Given a program under profiling, ComAir first instruments the program using light-weighted instrumentation methods. The key idea is to use software-based sampling to conduct algorithmic profiling. Given a code construct, like a loop or a function, with multiple dynamic instances in a program run, we sample some of the dynamic instances to record their input and cost information and leverage the mark-and-recapture method [43] to infer input and cost information for all instances. ComAir selectively instruments different code constructs and creates multiple program versions distributed to end users. ComAir then collects runtime profiles as different program versions are running on different user sites under different workloads. Next, ComAir synthesizes the profiles and generates a cost function of input size for each code construct to describe its complexity. Finally, ComAir reports a ranked list of code constructs in terms of their likelihood of containing performance bugs due to complexity problems.

ComAir provides several benefits over existing profilers: 1) ComAir supports production-run algorithmic profiling by taking into account real-world workloads and can thus discover most of the real complexity problems; 2) ComAir incurs a negligible runtime overhead while achieving accurate profiling results; 3) ComAir simplifies the task of detecting performance bugs related to complexity problems by providing a ranked list of code constructs in terms of their likelihood of containing complexity problems; and 4) ComAir provides information about real-world workloads processed by each code construct and guides developers toward making better priority decisions for performance bug fixing and optimization.

We envision that ComAir can be used in at least two scenarios. First, ComAir can be deployed on user side to conduct production-run algorithmic profiling. The results (i.e., a ranked list of code constructs with their cost functions) are returned to developers, which can help to localize performance bugs and understand which code constructs have the most performance impact on the system. Second, ComAir can be configured as an in-house testing tool used by developers and testers. After launched with existing testing inputs, ComAir automatically analyzes executed functions and reports their complexity information, which can help developers predict how each function scales.

To evaluate ComAir, we use 38 real-world complexity problems from two sources. These complexity problems cover different types of complexity and different types of root-cause code constructs. Experimental results show that ComAir can successfully identify the root cause for each evaluated bug without reporting any false positive under both in-house and production-run settings, and incur less than 5% runtime overhead under the production-run setting, while can still generate accurate profiling results.

In summary, we make the following contributions:

- We conduct the first empirical study on real-world complexity problems. The study provides several important findings and implications that can help developers better understand complexity problems and design tools to handle them.
- We develop ComAir, the first performance profiling tool that is intended to be used in production environment to effectively detect performance issues caused by different types of complexity problems and attribute the problems to specific code constructs with a negligible overhead. ComAir can also be configured as an in-house testing tool.
- We conduct thorough experiments and demonstrated the effectiveness, accuracy, and efficiency of ComAir.

2 BACKGROUND

In this section, we present preliminaries of algorithmic profiling and discuss existing techniques and their limitations.

2.1 Preliminaries

To conduct algorithmic profiling for a given code construct, we first need to record input and cost information in multiple program runs. We then plot the recorded information with input size as x-axis and cost as y-axis. In the end, we infer complexity for the code construct as a cost function of input size. To conduct algorithmic profiling effectively, we need to define suitable metrics for input and cost.

Input Metric. Inputs for a code construct can be viewed at different stack layers. For example, from the OS layer, a code construct can take inputs through memory read, so that accessed memory cells can be used to measure inputs. From the application layer, a code construct can take inputs from other components of the same program through internal data structures.

Read Memory Size (RMS) [9, 10] and Data Structure Size (DSS) [52] are two commonly used input metrics. RMS is measured as the the number of distinct memory cells whose first access is read. Both reads executed by the code construct directly and reads executed by the callees from the code construct are considered.

Take MySQL#27287 (Figure 1) as an example, during an execution of the buggy loop, the input size computed by RMS is roughly two times the number of actual accessed elements inside array `items`. This is because different `XML_NODE`s' `level` and `type` fields are read in different loop iterations. Although variable `p` and `level` are also read in each iteration, RMS only considers distinct memory cells and only increments its value for the first read on these two variables in the first iteration. For the outer loop, which is not shown in Figure 1, its RMS is roughly equal to the size of `items`, because the outer loop invokes `xml_parent_tag()` for every `XML_NODE` in the `items` and RMS only counts distinct memory cells.

Data Structure Size (DSS) is measured as the number of distinct accessed elements of a data structure. In Figure 1, if we focus on array `items`, in one execution of the inner loop, the input size computed by DSS is equal to the number of accessed array elements. As for the outer loop, the input size computed by DSS is equal to the size of `items`, because it invokes `xml_parent_tag()` for every element in `items`.

Cost Metric. When a code construct is executed, it could consume many different types of resources, such as CPU cycles, network bandwidth, storage, and so on. In this paper, we focus on CPU cycles (or computation cost), since computation cost is often correlated with user-perceived software slowdown [40]. We leave the measurement for other types of resources for future work.

Computation cost can be measured as the number of executed basic blocks (BBs) [9, 10]. The underlying assumption is that each executed BB will roughly consume the same CPU cycles. Executed instructions can also be used as cost metric. The number of executed instructions is highly correlated with executed BBs. For a buggy loop, the number of loop iterations (LIs) can represent the computation cost of a dynamic loop instance. Given a recursive function, the number of recursive invocations (RIs) of the function can serve as execution cost. Similarly to BBs, if we use LIs and RIs, we assume that each loop iteration or recursive invocation roughly consumes the same computation cost.

2.2 Limitations in Existing Techniques

There are at least three limitations in existing algorithmic profiling techniques. First, existing techniques can only attribute complexity to every executed code construct, while cannot provide more information about which code construct is the root cause for a performance slowdown recorded by multiple execution profiles. For example, after applying Aprof [9, 10] to the performance bug in Figure 1, there are 16 functions identified as having super-linear complexity. If we simply rank reported functions using their costs, `main` function will always be ranked as No.1. This is because callees' cost is aggregated to their callers, while code constructs involving computation in high complexity are buried inside call chains. Therefore, *more advanced techniques are needed to tell developers which function is the root cause from profiling results of existing techniques.*

Second, existing techniques incur significant runtime overheads. They simply record all direct information to calculate the metrics, failing to consider programs' structures and missing opportunities to optimize performance. For example, as an existing technique, Aprof can incur 30X slowdown in previous evaluation [9, 10].

Third, one option to mitigate runtime overhead is leveraging sampling, which is a widely-used technique to lower runtime overhead and deploy existing techniques to production environment [18, 24, 25, 40]. However, *existing algorithmic profiling techniques are not cooperative well with sampling approach.* Take MySQL#27287 in Figure 1 as an example, the n^2 complexity can only be observed on the side of the outer loop. One single execution of the outer loop contributes almost all of the execution time during performance failure runs. If we try to sample the outer loop, we may not get any sample and thus miss the opportunity to identify the n^2 complexity, or we get a sample but have to record all information for the sampled execution of the outer loop, which can still lead to a non-tolerable slowdown.

To address the above limitations, we design a novel ranking mechanism to identify functions that contain complexity problems causing the recorded slowdown. We infer complexity from the side of inner loops for cases like MySQL#27287, and successfully apply sampling to algorithmic profiling to lower runtime overhead. We will discuss more detailed design later in Section 4.

3 UNDERSTANDING REAL-WORLD COMPLEXITY PROBLEMS

In this section, we present an empirical study on real-world complexity problems. Our empirical study is conducted in two steps. First, we quantitatively compare complexity problems with non-complexity problems from a public performance-bug benchmark set [17, 40, 41]. We want to understand the popularity of complexity problems and whether they are different from non-complexity problems during reporting and diagnosis. Second, we build a taxonomy for complexity problems. For bugs in each category, we study what code constructs involve complexity problems, how the slowdown is caused and perceived by end users, and the fixing strategies.

3.1 Quantitative Comparison

The benchmark suite contains 65 user-perceived performance bugs collected from Apache, Chrome, GCC, Mozilla, and MySQL. These applications cover various types of functionalities and are implemented in different programming languages, including C/C++, Java, C#, and JavaScript. All the five applications are large and mature, with millions of lines of code and long development histories. All bugs in the benchmark suite are perceived and reported by real-world users and have large performance impact.

After carefully analyzing the bug reports and the associated buggy code fragments for bugs inside the benchmark suite, we identify 30 performance bugs related to complexity problems (or complexity bugs). Developers usually fix these bugs by applying optimized algorithms to lower complexity or to reduce workloads processed by the buggy code. These results indicate that *complexity problems are popular and indeed one of the major reasons causing software slowdown.*

Performance bugs not related to complexity problems are caused by various reasons. For example, there are several Mozilla bugs related to GUI operations, such as drawing transparent figures or refreshing web pages too frequently. There are also bugs caused by misusing synchronizations, such as using busy wait or I/O operations inside critical sections. Table 1 shows the numbers of complexity bugs and non-complexity bugs.

We next analyze how long it takes to resolve performance bugs. On average, developers use 162 days to fix a complexity problem and use 103 days to fix a non-complexity problem. The results indicate that complexity problems are potentially more difficult to fix than non-complexity problems.

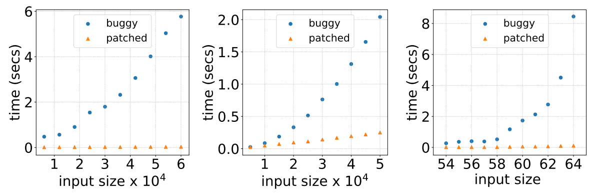
We also find that when reporting a performance bug, the end user often 1) compares the application's performance using inputs with different sizes [40], or 2) provides an input with repetitive patterns. There are 25 complexity bugs falling into one of the two cases, while for non-complexity problems, the number is only 8.

3.2 A Taxonomy of Complexity Problems

We categorize complexity problems based on their complexity types. We then study complexity problems in each type from the following

	Apache	Chrome	GCC	Mozilla	MySQL	Total
# of Non-Complexity and Complexity Bugs (Section 3.1)						
Non-Complexity Bugs:	12	2	1	9	11	35
Complexity Bugs:	4	3	8	10	5	30
Taxonomy of Complexity Bugs (Section 3.2)						
Complexity Type:						
$O(N)$:	1	0	0	4	2	7
$O(N^k)$ $(k > 1)$:	3	3	5	6	2	19
$O(e^N)$:	0	0	3	0	1	4
Root Cause:						
One-level Loop:	1	0	0	4	2	7
Nested Loops:	3	3	6	6	2	20
Recursive Function:	0	0	2	0	1	3
Fixing Strategies:						
Optimize Buggy Code:	3	3	4	9	5	24
Skip Workloads:	1	0	4	1	0	6

Table 1: Empirical Studies in Section 3.



(a) MySQL#27287 **(b) Apache#34464** **(c) GCC#27733**

Figure 2: How the execution time scales with input size for three complexity problems. ***We use 10 distinct inputs to plot a curve for each bug.***

three aspects: 1) what are their root causes[3]; 2) how they generate user-perceived slowdown; 3) how they are fixed by developers. Table 1 shows the results.

$O(N)$: linear complexity. As shown in Table 1, 7 out of 30 studied complexity problems are in linear complexity. All of them are caused by a buggy loop, whose iteration number scales with input size N.

Five of them are characterized by buggy loops that contain serialized I/O operations. Users could perceive these bugs, even though the loop iteration numbers are not large. Patching these 5 bugs involves aggregating I/O operations or completely eliminating unnecessary I/O operations. For example, Mozilla#490742 is caused by bookmarking tabs using individual database transactions. Even bookmarking 50 tabs can cause a timeout dialog window to pop up during a performance failure run. To fix this bug, Mozilla developers use one single aggregated transaction to bookmark all tabs.

For the other two bugs, their buggy loops execute many iterations during performance failure runs. They are fixed by adding shortcuts to completely skip the buggy loops. Take MySQL#33948 as an example, MySQL developers followed a common practice to keep all table entries in the same linked list, including both used ones and free ones. The buggy loop iterates the linked list and looks for a free entry. During performance failure runs, many table entries are used and the buggy loop must iterate excessively to find a free entry. To fix this bug, MySQL developers simply separate used entries and free entries into two separated linked lists.

$O(N^k)$: polynomial complexity (k>1). As shown in Table 1, more than half of the studied complexity bugs are in polynomial complexity. Similar to the bugs in linear complexity, the polynomial complexity is also caused by a buggy loop. However, **both** the loop execution number and the average loop iteration number scale as input size N.

[3]We refer root causes as code constructs conducting the inefficient computation, following previous works in this area [40, 41].

```
//indexOf: check if string src[] contains string tgt[]
//srcLen: the length of string src[]
//tgtLen: the length of string tgt[]
+ int i = -1 * tgtLen
- while(indexOf(src, 0, srcLen, tgt, tgtLen) < 0) {
+ while(i++ < 0|| indexOf(src, i, srcLen, tgt, tgtLen) {
          //append another character; try again
          src[srcLen++] = getchar();
  }
```

Figure 3: An Apache bug in $O(N^2)$ complexity. ***The execution cost scales polynomially as the number of chars from `getchar()`.***

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

void mult_alg(... HOST_WIDE_UINT t, ...) {
  hash_index = hash (t);
  if (alg_hash[hash_index].t == t)
  { ... //fast path: reuse previous results
  } else {
    ... //slow path: expensive recursive computation
    t1 = bitwise-operations(t) (t1 < t)
    mult_alg (...t1, ...);
  }
}
```

Figure 4: A GCC performance problem in exponential complexity. ***How many times `mult_alg` is invoked scales exponentially with the number of 1s in the binary form of input `t`.***

To fix the majority (14/19) of bugs in this category, developers directly modify the buggy loops, whose total iteration numbers scale polynomially. Take MySQL#27287 as an illustration, to fix this bug, developers add an extra field to save parent `XML_NODE` and completely remove the buggy loop in Figure 1. How the execution time changes after patching for MySQL#27287 is shown in Figure 2a.

To fix the other bugs (5/19), developers reduce workloads processed by the loops scaling polynomially, instead of changing the loops directly. The buggy code fragment of Apache#34464 is shown in Figure 3. The `while` loop searches a string `src` for a target substring `tgt`. If the search is unsuccessful, a new character returned from `getchar()` will be appended to string `src`, and the loop will search `src` again from the beginning. There is an inner loop inside `indexOf()`, whose total iterations scale polynomially in terms of the number of characters returned from `getchar()`. After fixing this bug, the inner loop will only check the most recent `tgtLen` characters. The developers do not change the inner loop, while reducing the workload it processes. Figure 2b shows how the execution time scales after patching for this bug.

$O(e^N)$: exponential complexity. Four of the studied complexity problems are in exponential complexity. These complexity problems are fixed either by leveraging memoization to reuse previous results or by skipping the computation with exponential complexity for large workloads.

Three of the studied performance problems are caused by recursive function calls. Taking GCC#27733 in Figure 4 as an example, the recursive function `mult_alg` computes the best algorithm to multiply `t`. In each invocation, `mult_alg` will try a set of bitwise operations to change input `t` into a smaller number, `t'`, and recursively call itself. The number of times when `mult_alg` is invoked scales exponentially in terms of the number of 1s in the binary form of input `t`. To optimize this function, GCC developers use a hash table, `alg_hash`, to record which `t` has been processed before and its corresponding result. However, there is a type declaration error inside the hash table entry, causing `t` larger than the maximum unsigned integer to never hit cache. For large `t`, `mult_alg` is still in exponential complexity. After fixing the type declaration error, memoization is enabled for large `t`, as shown in Figure 2c.

3.3 Implications

We summarize the implications from our study, which can guide the design and implementation of ComAir.

Implication 1. To effectively profile algorithmic complexity, *a set of inputs providing similar code coverage with different input sizes are needed.* As we discussed in Section 3.1 (last paragraph), for majority (25/30) of complexity problems, it is fairly easy for developers to get these inputs based on users' descriptions and provided inputs when they report complexity problems.

Implication 2. Our study shows that all complexity problems are caused either by a loop or a recursive function. This finding suggests that algorithmic profiling should *focus on code constructs involving loops and recursive functions.* Our study also shows that approximately three-fourths of complexity problems (23/30) are caused by repeated executions of a loop or a recursive function. Previous work [40, 41] demostrates that sampling code constructs that execute many times in one program run can lower the runtime overhead while preserving the same diagnosis or detection capability. It is promising to *apply sampling to design production-run algorithmic profiling techniques.*

Implication 3. We also examine what types of data structures holding inputs processed by buggy loops or recursive functions. The most two common types of data structures are array (11/30) and linked list (9/30). Other types of data structures are either application-specific or can only cover one or two bugs. This fact inspires us to *design effective algorithms for loops that process an array or a linked list during algorithmic profiling.*

4 COMAIR DESIGN AND IMPLEMENTATION

Guided by the findings of the empirical study in Section 3, we design and implement ComAir, an algorithmic profiler to detect complexity problems. ComAir can be used in both production environment and in-house development. We first describe the production-run version of ComAir. The key benefits of ComAir include its *low runtime* overhead and capability of *ranking and localizing complexity problems in specific code constructs.* We then describe how to use ComAir as an in-house testing tool.

The design of ComAir used in production environment follows two steps. First, given a program under profiling, ComAir instruments the program and distributes it to end users to collect runtime profiles under different user inputs (e.g., workloads). To minimize runtime overhead, we design and implement several advanced instrumentation methods in ComAir. In the second step, ComAir computes cost functions for different code constructs based on the execution profiles and then reports a ranked list of code constructs in terms of their likelihood of containing performance bugs due to complexity problems. We next describe details of the two steps.

4.1 Using Light-Weighted Instrumentation Methods for Algorithmic Profiling

To obtain execution profiles, we need to select suitable *profiling targets* (i.e., code constructs), *input metrics*, and *cost metrics.*

ComAir supports code constructs in any granularity as profiling targets. However, one finding (Implication 2) in our empirical study is that all complexity problems are caused either by a loop or a recursive function. Therefore, we consider code constructs related to loops and recursive functions as profiling targets.

As for input metrics, our study found that the most two commonly involved data structures in complexity problems are array and linked list (Implication 3). Therefore, if a loop is chosen as a profiling target and it is to process an array or a linked list, we use Data Structure Size (DSS) as its input metric. Otherwise, we use Read Memory Size (RMS), since RMS is more generic as we discussed in Section 2. The difference between DSS and RMS is that we only need to instrument `read` instructions accessing elements in an array or a linked list with DSS, while RMS needs to monitor both `read` and `write` instructions to figure out whether a memory cell is firstly accessed by `read`. Therefore, DSS requires recording less dynamic information.

We use the executed BBs as cost metric. As discussed in Section 2, the number of executed BBs is a generic metric and it is more lightweight than executed instructions. We design a novel algorithm to reduce the number of instrumentation sites, while still computing the precise number of executed BBs.

As discussed in Section 2.2, existing algorithmic profiling techniques often incur significant overhead and thus cannot be used in production environment. To address this issue, ComAir proposes several methods to *enable practical online monitoring, which constitute the key novelty of ComAir over existing algorithmic profiling tools.* These methods include 1) minimizing instrumented code at each user's end; 2) optimizing instrumentation sites; 3) approximation of input metrics; and 4) sampling profiling targets. We next discuss these methods in detail.

4.1.1 Creating Multiple Instrumentation Versions. ComAir creates multiple program versions for a program under profiling, and for each version ComAir only instruments a small number of profiling targets. All different versions are distributed to different end users. Since there is a huge amount of end users, we anticipate that enough profiles can still be collected for each version to conduct algorithmic profiling.

To select profiling targets, we are largely guided by our empirical study in Section 3.2. Since all complexity problems involve a loop or a recursive function, we choose loops, recursive functions, functions containing loops, and functions invoked inside a loop directly or indirectly as profiling targets. We ignore loops whose iteration numbers are constant, since the execution of these loops does not scale with input and cannot cause complexity problems.

For each program version, we only instrument a small set of profiling targets to reduce runtime overhead. Given a profiling target, its direct and indirect callees are also instrumented. Therefore, we will not put code constructs with caller-callee relationship in the same set. If there are limited program versions we can distribute at the same time, we prioritize "simple" code constructs, such as functions without callees and inner loops of nested loops, and we gradually select more "complex" code constructs along call chains, if we cannot find complexity problems inside simple code constructs.

4.1.2 Optimizing Instrumentation Sites. We propose three instrumentation optimization strategies, two for obtaining input metric, and one for obtaining cost metric.

Input Metric. To measure RMS (i.e., the input metric), we need to instrument `read` and `write` instructions in a code construct (or a profiling target). Different from the algorithm described in previous work [9, 10], we do not maintain any global data structure, since we want to simplify online monitoring and reduce runtime overhead. Instead, we trace `read` and `write` instructions to record instructions' id and accessed addresses. We then analyze tracing log offline to calculate RMS.

In order to reduce the number of `read` and `write` instructions to be instrumented, we propose two optimization algorithms. The first algorithm works as follows. If a write on a memory cell happens earlier, following read on the same memory cell will not increase RMS, since RMS only counts memory cells whose first access is read. Similarly, if a read on a memory cell happens earlier, following read on the same memory cell will not increase RMS either, since RMS only considers distinct memory cells. Therefore, given two consecutive memory accesses on the same location, we do not need to instrument the second one. We design an intra-procedural analysis, based on dominator analysis, which can tell us when an instruction is executed whether another instruction has to be executed in advance. We focus on stack memory cells that hold scalar variables and only have read and write as uses (i.e., not having "address of" as uses), so that alias analysis is avoided. In the example of Figure 1, suppose there are four reads on variable `p` inside the loop — two are at line 7 ("`p>=items`" and "`p--`"), and the other two are at line 8 and there is one write on `p` ("`p--`"). In this case, variable `p` meets the requirement of the optimization, because "`p>=items`" is conducted in the loop header and it dominates all other accesses. Therefore, we only need to instrument the read access inside "`p>=items`".

For the second optimization algorithm, if a read or a write is conducted inside a loop and the accessed address is not changed across different loop iterations, we promote the instrumentation to the loop pre-header, instead of recording the accessed address in every iteration. For example, in Figure 1, the address of variable `p` is not changed inside the loop. Therefore, we can record the address of `p` in the loop's preheader for "`p>=items`".

Cost Metric. When counting the executed basic blocks (BBs) to compute cost metric, a naive way is to instrument every BB to increment a counter by one. To reduce the number of instrumentation sites that update the counter, we design an algorithm to selectively instrument part of BBs.

Our algorithm leverages the previous work on counting edge[4] events through selectively instrumenting counter update sites on control flow graphs (CFGs) [3], which has been proved to conduct path profiling efficiently [4, 36]. However, the original algorithm cannot be directly adopted in our case because it is designed to count edge events, not BB events. To address this problem, ComAir firstly splits each monitored BB into two, and the monitored BBs include BBs inside profiling targets and BBs inside functions called from profiling targets. Then, ComAir labels the event number as "1" for edges connecting a pair of split BBs, leaving the event number as "0" for other edges. After that, applying the algorithm [3] can inform ComAir where to update the counter and how to update the counter. For example, in Figure 1 there are in total six BBs inside the loop, but ComAir only instruments five of them and updates the counter by two for one BB and by one for the other four BBs, instead of updating the counter by one for all the six BBs. As another example, there are 141 BBs inside function `mult_alg()` in Figure 4, while ComAir only instruments 78 of them.

[4]An edge on CFG represents a jump from one basic block to another basic block.

4.1.3 Approximation of Input Metric.

Accurately recording dynamic input information for a profiling target may still incur a large runtime overhead. Therefore, we propose an approximation mechanism. If a profiling target is a loop and it is to process an array or a linked list, we use the DSS as input metric.

To apply the approximation, we are facing two challenges. The first one is to *identify array-processing loops*. To address this, given a loop, we analyze all pointer usage inside the loop. If a pointer is deferenced in every iteration of the loop, and its value is also increased or decreased by an integer number in every iteration, we consider the pointer points to array elements and the loop is an array-processing loop. For example, as shown in Figure 1, `p` points to array elements. It is deferenced in every iteration of the loop in Figure 1, and the pointer value is decreased by one in every loop iteration. For an array-processing loop, instead of recording all addresses of accessed array elements, we only record the addresses of the first and the last accessed elements. We calculate the address difference and use it as DSS. The reason is that array is organized as a consecutive memory, and the address difference between the first and last accessed elements is approximately in linear relationship with the number of accessed elements and DSS.

The second challenge is to *identify linked-list-processing loops*. To address this, we also analyze all pointer usage inside a loop under profiling, by checking whether a pointer satisfies the following three conditions. First, its base type is a `struct`. Second, it is deferenced in every iteration of the loop. Third, it is updated in every iteration with a new value from one field of the `struct` it points to. If a pointer satisfies the three conditions, we consider it points to elements in a linked list. We also consider the loop is a linked-list-processing loop.

4.1.4 Sampling Dynamic Instances of Profiling Targets.

Previous works [18, 24, 25, 40, 41] demonstrate that sampling can greatly reduce overhead for dynamic techniques, while still preserving their desired detection and diagnosis capability. This motivates us to apply sampling to algorithmic profiling. Our study in Section 3 shows that the majority of complexity problems are caused by repeated execution of a loop or a recursive function (Implication 2). This finding inspires us to sample some dynamic instances of a profiling target in one program run to record their input sizes and costs, and infer the whole input size and cost for all instances in the same run.

How we do sampling is similar to what is described in previous works on statistical debugging [18, 24, 25, 40, 41]. Given a profiling target, we clone it and we instrument the cloned version to record information for input size and cost. We also dump extra delimiters to log before each execution of the cloned version to differentiate information collected from different dynamic instances.

Before each execution of a profiling target, we choose between the cloned version and the original version. How many times the cloned version is executed depends on a tunable sampling rate. To make the choice between the two versions, we add a global counter to the monitored program. If the counter value is larger than zero, we choose the original version and decrease the counter value by one. If the counter value is equal to zero, we choose the cloned version and reset the counter value to a random number, whose expectation is equal to the inverse of the sampling rate. To guarantee we have something recorded in each profiled program run, we always sample the first dynamic instance.

After sampling, ComAir next *infers input and cost information for all instances of the same profiling target.* Cost can be simply inferred by multiplying recorded cost values with sampling rate. To infer the whole input size for all instances, we leverage the mark-and-recapture method [43]. Mark-and-recapture is a commonly used statistical method for estimating the size of an animal population. In this method, some of the animals are captured, marked, and released. Then, another group of the animals are captured. The size of the whole animal population is estimated based on the ratio of marked animals in the second captured sample.

To utilize the mark-and-recapture method, we first process our log to calculate a set of memory cells, which contribute RMS or DSS for each sampled dynamic instance. We then estimate the whole RMS or DSS for all instances using the following formula. We assume a sequence of m samples is sampled for a profiling target in one program run. Given the ith sample, we use M_i to represent the total number of distinct memory cells in the previous $i-1$ samples, C_i represents the number of distinct memory cells in the ith sample, and R_i represents the number of distinct memory cells in the ith sample that also appeared in one of the previous $i-1$ samples. The estimated RMS or DSS for the profiling target is:

$$N = \sum_{i=1}^{m} M_i * C_i \Big/ \sum_{i=1}^{m} R_i \tag{1}$$

4.2 Ranking Complexity Problems

After the profiles are collected, ComAir computes the cost function for each unique code construct (i.e., profile target) using the standard curve fitting method [6, 29].

As we discussed in Section 2.2, existing techniques [9, 10, 52] simply attribute complexity to each function, without providing any future analysis. Therefore, they fail to identify root causes for performance failures caused by complexity problems.

To address this problem, we design a ranking algorithm with the goal to identify root-cause code constructs. Our algorithm follows three intuitions. First, root-cause code constructs must have highest complexity. Second, root-cause code constructs must consume large computation cost. Third, since all direct and indirect callers of root-cause code constructs consume more cost and may have the same complexity, we should rank a callee higher than its caller, to reduce false positives.

Our algorithm works as follows. According to their complexity, we first divide profiling targets into three groups, exponential-complexity group, polynomial-complexity group and linear-complexity group. We always rank profiling targets with exponential complexity highest, profiling targets with polynomial complexity in the middle, and profiling targets with linear complexity lowest. For profiling targets within the same group, we then compute caller-callee relationship using static analysis and tracing logs. Since we record instructions' ids, tracing logs can help us solve function pointers. If there is direct or indirect caller-callee relationship between profiling target A and profiling target B, we rank B higher than A. Since a root-cause profiling target may cause all its callers to be in the same complexity and caller always has a larger cost than its callee, prioritizing callees over callers can help reduce false positives. If there is no caller-callee relationship, we rank the one with larger observed cost higher.

To illustrate our ranking algorithm, suppose we have four profiling targets, whose target ids (A to D), complexity and ever observed largest costs are listed as follows: (A, $O(n)$, 2000), (B, $O(n^2)$, 800), (C, $O(n^2)$, 1200), and (D, $O(n^2)$, 300). We also assume target C is an indirect caller of target B and target D, and there is no caller-callee relationship between target B and target D. According to our algorithm, we rank the four profiling targets as B, D, C, and A. Targets B, C and D have higher complexity than target A, so that these three are ranked higher. Target C is has caller-callee relationship with targets B and D, and thus it is ranked lower. Target B has larger cost than target D and it has no caller-callee relationship with target D, so that it is ranked as No. 1.

4.3 In-house Algorithmic Profiling

During in-house testing, ComAir profiles the whole program, since runtime overhead is less of a concern. Existing testing inputs can be used to launch ComAir, and testing input generation techniques [5, 7, 14] can also be utilized to create more inputs. For testing inputs with repetitive patterns, developers can change these inputs by themselves and create inputs with the same functionality but different sizes. ComAir automatically merges input and cost information for the same code constructs collected from multiple runs with different inputs and infer cost functions.

The in-house version of ComAir uses RMS as input metric and BB as cost metric. It is configured by enabling only the three instrumentation site optimizations (Section 4.1.2), without enabling approximation (Section 4.1.3) and sampling (Section 4.1.4). The reason is that the instrumentation site optimizations intend to speed up ComAir, while still providing the same results, but enabling the approximation and sampling will sacrifice accuracy. Similar to Aprof [9, 10], the in-house version of ComAir leverages several global data structures to preprocess memory access information to lower memory and disk overhead, instead of directly tracing read and write information into log, as what we did for the production-run version. To keep the implementation simple, we choose function as profiling granularity. Otherwise, we need to implement more complex mechanisms to judge whether a given code construct has finished its execution.

Different from the production-run version, the in-house version of ComAir considers each individual dynamic instance of a function as a distinct point during inferring cost function, while not merging all dynamic instances of the function from the same program run. This is because if we want to merge all dynamic instances for a given function, we need to track accessed memory cells by previous dynamic instances, which will incur a large memory overhead. As we discussed in Section 2, given a complexity problem with polynomial complexity (e.g. Figure 1), the polynomial complexity can be inferred when profiling the function containing the outer loop by the in-house version.

Like the production version of ComAir, the in-house version also provides a ranked list of functions to help localize complexity problems. However, existing profilers (e.g., Aprof [9, 10]) simply attribute complexity to each executed function, without providing any postmortem analysis.

5 EVALUATION OF COMAIR

Our experiments aim to evaluate whether ComAir can provide the desired profiling capability, accuracy and performance. Specifically, we answer the following three research questions:

RQ1: How effective is ComAir at identifying complexity problems in both production and in-house environments?

RQ2: How accurate is the production-run version of ComAir after equipped with various instrumentation methods?

RQ3: Can the production-run version of ComAir be deployed in production environment?

The first research question allows us to evaluate whether ComAir is effective at detecting real-world complexity problems. The second research question investigates whether the instrumentation methods used in the production-run version of ComAir can lower runtime overhead while remaining the accuracy in detecting complexity problems. The third research question explores whether the overhead generated by the production-run version of ComAir is negligible enough for being used in production environment.

5.1 Experimental Setup

We implement ComAir using LLVM-5.0.0 [23]. All our experiments are conducted on a Linux machine, with Intel Xeon(R) 4110 CPU and 3.10.0 kernel. ComAir and all the data from the experiments are publicly available at https://github.com/ComAirProject/ComAir.

5.1.1 Benchmarks. We collect evaluated complexity problems from two sources. The first one is from a public performance-bug benchmark suite [17, 40, 41], which is studied in Section 3. We used 17 bugs, and the others cannot be successfully reproduced, because they depend on special hardware or software, which is not available to us. Among the 17 reproduced bugs, six are from original Java or JS programs, and we re-implement them using C/C++. The reason is that our current implementation of ComAir only supports C/C++ programs. Our re-implementation is based on our thorough understanding of these complexity problems. In our re-implementation, we maintain their original algorithms, data structures and caller-callee relationships.

The second source is from Toddler [35] and LDoctor [41]. Toddler is a dynamic technique that can detect inefficient loops in Java programs. In total, developers confirmed 21 bugs detected by Toddler. These bugs were later re-implemented in C/C++ by LDoctor. Thus, we use all the 21 bugs from LDoctor in our evaluation.

In total, we evaluate ComAir on 38 benchmark programs. All benchmark information is shown in Table 2. Our evaluated original bugs are all large, and seven of them are more than 1 million lines of code. Our evaluated re-implemented bugs are in small to middle sizes, ranging from 91 to 1094 lines of code. The majority of bugs used in our experiments are in $O(N^2)$ complexity. We also have two bugs in $O(2^N)$ complexity and four in O(N) complexity.

5.1.2 Production-run and In-house Inputs. For the production-run version of ComAir, we create inputs used in our evaluation by simulating real user inputs. For each of the 38 benchmarks, we create ten distinct inputs, following previous practice in evaluating statistical debugging [2, 40]. To obtain such inputs, we use bug reports, which contain detailed information about how real-world users perceive the complexity problems. For 14 bugs, the users explicitly explained how to change input sizes to reproduce the described scaling problems in the bug reports. We follow the users' descriptions to create new inputs. For the other bugs, their bug-triggering inputs provided in the bug reports contain repetitive patterns, so that it is not difficult for us to create new inputs. For example, when reporting GCC#46401, the user provided a C file, containing an expression with thousands of strings as operands. We change the number of strings to create new inputs. Finally, we create ten distinct inputs for each bug, and we set their sizes to be an arithmetic sequence with the largest input size equal to the input in the bug report and the smallest input size equal to 1/10 of the largest input. We use the same inputs to evaluate the in-house version of ComAir to fairly compare accuracy and runtime overhead between the two versions of ComAir.

5.1.3 Experimental Settings. For each benchmark, we conduct algorithmic profiling on ten program runs with the ten distinct inputs for the two versions of ComAir, following the practice in previous works [2, 40]. After collecting a set of (input, cost) pairs for a code construct, we only keep the pair with largest cost for pairs with the same input size, since we want to infer the worst-case complexity. We use 1/100 as default sampling rate for the production-run version of ComAir, which is also the default sampling rate in previous works on statistical debugging [18, 24, 41].

To answer RQ1 (i.e., the effectiveness of ComAir in identifying complexity problems), we measure the ranking number (position) of code constructs containing complexity problems. To determine whether a code construct matches the known complexity problem (i.e., ground truth), we manually examined the solution discussed in the corresponding issue report and the patch used for fixing the issue. To apply the production-run version of ComAir on each bug, we first use a PIN tool [28] to record executed iteration numbers for all loops using the bug-triggering input, and then we instrument the top five loops with the largest loop iteration numbers to create five distinct program versions deployed in production, since instrumenting all loops is expensive. For benchmarks with less than five executed loops during buggy runs, we instrument all executed loops. For the in-house version, since the whole program is profiled, we rank all executed functions during buggy runs.

To answer RQ2 (i.e., the accuracy of the production-run version of ComAir), we calculate the R^2 value [42] for the two groups of input sizes (and costs) reported by the production-run version of ComAir with and without instrumentation methods discussed in Section 4.1. R^2 represents how well one variable can be predicted by another variable, and a R^2 value close to one means the two variables are in strong linear relationship [8]. Therefore, if the computed R^2 values for input sizes and costs are both close to one, the inferred two cost functions with and without instrumentation methods must be in the same order, which means that both sampling and approximation mechanisms do not hurt the accuracy.

To answer RQ3 (i.e., whether the production-run version of ComAir can be really deployed in production runs), given a benchmark, we measure the runtime overhead for the program version with the buggy loop instrumented, compared with non-instrumented version. We run each version **ten** times and compute the overhead based by average execution time.

5.2 Experimental Results

All experimental results are shown in Table 2. We discuss how these results can answer the previous three research questions as follows.

	Benchmark Information			Production-run Version							In-house Version	
BugID	KLOC	P. L.	Comp.	Ranking	Cost Function	R^2-Func	R^2-Input	R^2-Cost	Overhead w/ ins.	Overhead w/o ins.	Ranking	Overhead
Mozilla#347306	88	C	$O(N^2)$	1_5	$y = 9.54 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	2.34%	8.07X	1_{465}	79X
Mozilla#416628	105	C	$O(N^2)$	1_5	$y = 7.20 * 10^{-3}x^2$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	3.77%	9.17X	1_{394}	1252X
Mozilla#490742	0.157^*	JS	$O(N)$	1_4	$y = 5.00x$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{1.00}$	0.22%	5.37%	1_8	54.46%
Mozilla#35294	0.195^*	C++	$O(N^2)$	1_5	$y = 1.06 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	0.12%	9.58X	1_{12}	98X
Mozilla#477564	0.116^*	JS	$O(N^2)$	1_5	$y = 6.44 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	2.79%	11.50X	1_5	141X
MySQL#27287	995	C++	$O(N^2)$	1_5	$y = 8.74 * 10^{-3}x^2$	$\checkmark_{0.98}$	$\checkmark_{0.99}$	$\checkmark_{0.98}$	3.59%	78.49%	1_{1544}	10X
MySQL#15811	1127	C++	$O(N^2)$	1_5	$y = 1.34 * 10^{-1}x^2$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	4.44%	93.33%	1_{223}	57.77%
Apache#37184	0.092^*	Java	$O(N)$	1_3	$y = 7.00x$	$\checkmark_{1.00}$	$\checkmark_{1.00}$	$\checkmark_{1.00}$	4.34%	43.48%	1_7	5X
Apache#29743	0.257^*	Java	$O(N^2)$	1_3	$y = 1.56 * 10^{-4}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	1.56%	18.43%	1_{110}	54.11%
Apache#34464	0.16^*	Java	$O(N^2)$	1_4	$y = 5.64 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	0.18%	2.14X	1_9	38X
Apache#47223	0.162^*	Java	$O(N^2)$	1_4	$y = 1.56 * 10^{-4}x^2$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	4.14%	5.39X	1_{12}	81X
GCC#46401	5521	C	$O(N^2)$	1_5	$y = 3.12 * 10^{-3}x^2$	$\checkmark_{0.99}$	$\checkmark_{0.98}$	$\checkmark_{0.99}$	0.37%	7.36X	1_{4249}	32X
GCC#1687	2099	C	$O(2^N)$	1_5	$y = 2.13 * 10^2 e^{0.005x}$	$\checkmark_{0.98}$	$\checkmark_{0.94}$	$\checkmark_{0.93}$	4.57%	6.94X	1_{1481}	96X
GCC#27733	3217	C	$O(2^N)$	1_5	$y = 2.33e^{0.003x}$	$\checkmark_{0.97}$	$\checkmark_{0.95}$	$\checkmark_{0.98}$	4.34%	5.17X	1_{3573}	48X
GCC#8805	2538	C	$O(N^2)$	1_5	$y = 2.08 * 10^{-5}x^2$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	3.28%	1.78X	1_{1884}	48X
GCC#21430	3844	C	$O(N^2)$	1_5	$y = 1.02 * 10^{-3}x^2$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	4.63%	1.61X	1_{3370}	23X
GCC#12322	2341	C	$O(N^2)$	1_5	$y = 2.43 * 10^{-5}x^2$	$\checkmark_{0.99}$	$\checkmark_{0.98}$	$\checkmark_{0.99}$	1.75%	79.65%	1_{116}	87X
Apache#53622	1.094^*	Java	$O(N^2)$	1_5	$y = 6.64 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.98}$	0.36%	75.03%	1_{41}	27X
Apache#53637	0.937^*	Java	$O(N^2)$	1_5	$y = 1.03 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	0.49%	3.52X	1_{27}	68X
Apache#53803	0.421^*	Java	$O(N^2)$	1_4	$y = 4.37 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	1.27%	3.59X	1_{17}	51X
Apache#53821	0.417^*	Java	$O(N^2)$	1_4	$y = 4.36 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	0.08%	7.21X	1_{16}	97X
Apache#53822	0.417^*	Java	$O(N^2)$	1_4	$y = 5.66 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	2.18%	6.96X	1_{16}	91X
Collections#406	0.253^*	Java	$O(N^2)$	1_5	$y = 1.17 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{1.00}$	2.29%	1.63X	1_8	92X
Collections#407	0.766^*	Java	$O(N^2)$	1_5	$y = 4.36 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	0.97%	6.21X	1_{21}	93X
Collections#408	0.742^*	Java	$O(N^2)$	1_5	$y = 8.07 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	0.08%	6.86X	1_{28}	95X
Collections#409	0.781^*	Java	$O(N^2)$	1_5	$y = 1.03 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	2.52%	1.17X	1_{27}	86X
Collections#410	0.769^*	Java	$O(N^2)$	1_5	$y = 1.03 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	2.26%	4.19X	1_{31}	78X
Collections#412	0.284^*	Java	$O(N^2)$	1_5	$y = 4.31 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	1.31%	4.77X	1_{10}	90X
Collections#413	0.923^*	Java	$O(N^2)$	1_5	$y = 2.69 * 10^{-3}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	3.88%	4.15X	1_{23}	117X
Collections#425	0.758^*	Java	$O(N^2)$	1_5	$y = 4.31 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	2.43%	6.74X	1_{21}	94X
Collections#426	0.783^*	Java	$O(N^2)$	1_5	$y = 4.36 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	3.53%	7.16X	1_{23}	95X
Collections#427	0.756^*	Java	$O(N^2)$	1_5	$y = 8.05 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	1.99%	6.68X	1_{26}	94X
Collections#429-0	0.668^*	Java	$O(N)$	1_5	$y = 2.52 * 10^2 x$	$\checkmark_{0.98}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	2.75%	11.41X	1_{19}	148X
Collections#429-1	0.536^*	Java	$O(N)$	1_5	$y = 6.61 * 10^2 x$	$\checkmark_{0.92}$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	3.39%	2.08X	1_{18}	8X
Collections#429-2	0.416^*	Java	$O(N^2)$	1_5	$y = 2.17 * 10^{-3}x^2$	$\checkmark_{0.99}$	$\checkmark_{1.00}$	$\checkmark_{0.99}$	4.63%	4.53X	1_{21}	55X
Collections#434	0.336^*	Java	$O(N^2)$	1_5	$y = 3.64 * 10^{-3}x^2$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	4.82%	4.99X	1_{17}	118X
Groovy#5739-0	0.745^*	Java	$O(N^2)$	1_5	$y = 2.54 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	$\checkmark_{0.98}$	2.88%	5.82X	1_{18}	93X
Groovy#5739-1	0.756^*	Java	$O(N^2)$	1_5	$y = 2.71 * 10^{-2}x^2$	$\checkmark_{0.99}$	$\checkmark_{0.99}$	$\checkmark_{0.98}$	1.32%	6.22X	1_{17}	90X

Table 2: Benchmark Information and evaluation results for ComAir. ***In the "Benchmark Information" columns, x^*: x thousands of lines of code for re-implemented benchmarks. In the "Production-run Version" columns, x_y: the root-cause loop is ranked as xth in y evaluated loops; Cost Function: we only report the part with the highest order; R^2-Func: R^2 values comparing observed cost with predicted cost by cost function; R^2-Input: R^2 values comparing input sizes reported with and without instrumentation methods; ins. is short for instrumentation methods; $\checkmark_x$: the computed R^2 value x is larger than 0.92; In the "In-house Version" columns, x_y: the root-cause function is ranked as xth in y executed functions.***

5.2.1 *RQ1: Effectiveness.*

As shown by the ranking numbers in Table 2, ComAir can effectively identify buggy loops and buggy functions among all evaluated loops and executed functions. The buggy loop and the buggy function are ranked as No. 1 for **all** benchmarks by the production-run version and the in-house version respectively. The numbers of evaluated or executed code constructs are shown as subscripts of the two "Ranking" columns. When evaluating the production-run version, the evaluated loops range from 3 to 5 for each bug. When evaluating the in-house version, there are 11 bugs whose executed functions are more than 100. GCC#46401 has 4249 executed functions, which is the largest number. Our ranking mechanism can effectively identify root causes among these code constructs.

Our ranking mechanism can indeed save developers' efforts and help identify root causes. Take GCC#8805 as an example, this bug is caused by a nested loop, whose inner loop's total iterations are in polynomial complexity (e.g., $O(N^2)$). To fix this bug, GCC developers significantly reduce the workload processed by the inner loop. Among the five evaluated loops, three of them are in superlinear complexity, and the production-run version successfully ranks the inner loop as No. 1. There are in total 1373 executed functions during buggy runs, and 87 of them are in superlinear complexity. The buggy function containing the outer loop is ranked as No. 1 by the in-house version. After referring the results of ComAir, developers can avoid manually inspecting the large number of suspicious loops or functions in superlinear complexity.

5.2.2 *RQ2: Accuracy.*

As shown in Table 2 (the "Cost Function" column), the production-run version of ComAir successfully reports the correct complexity for the buggy loops of all evaluated benchmarks. We compute the R^2 value between observed cost values and cost values predicted by the inferred cost function for each bug (the "R^2-Func" column). There are only five bugs, whose computed R^2 value is less than 0.99. The minimum R^2 value is 0.92 for

Collections#429-1. Previous work considers R^2 value larger than 0.92 as a good fitting [8]. Our results show that inferred cost functions can well represent observed (input size, cost) pairs. We also compute R^2 to compare the reported input sizes and cost values by the two versions of ComAir. For 31 out of 38 benchmarks, the R^2 values are larger than 0.99 for both input sizes and cost values. For the left seven benchmarks, the R^2 values are all larger than 0.92. These results show that *the production-run version of ComAir is as accurate as the in-house version.*

The R^2 values computed by comparing input sizes and cost values reported by the production-run version with and without instrumentation methods (Section 4.1) are shown in Table 2 (the "R^2-Input" and "R^2-Cost" columns). For 31 out of 38 benchmarks, R^2 are larger than 0.99 for both input sizes and cost values. For the other seven bugs, the computed R^2 values are not low, and all of them are larger than 0.92, which is considered as a good fitting. The minimum R^2 value is 0.93 for GCC#1687. Applied instrumentation methods include sampling and approximation, and these two can potentially influence profiling results. However, our experimental results demonstrate that *sampling and approximation applied in the production-run version of ComAir do not hurt accuracy.*

5.2.3 RQ3: Overhead. The runtime overhead for the production-run version of ComAir is shown in Table 2 (the "Overhead w/ ins." column). In general, the performance is good. The runtime overhead is constantly under 5% for all evaluated bugs. There are night bugs, whose overhead is less than 1%, and 15 bugs, whose overhead is in the range from 1% to 3%. The largest overhead is 4.82% for Collections#434. The results imply that *the production-run version of ComAir only incurs a negligible runtime overhead, and it can be deployed in production environment.*

We also measure the runtime overhead for the production-run version without enabling instrumentation methods, including sampling, instrumentation-site optimizations and approximation (the "Overhead w/o ins." column). The overhead is significantly larger, compared with when instrumentation methods enabled. For 31 bugs, the overhead is larger than 1X. For two bugs, the overhead is larger than 10X. In summary, our designed instrumentation methods can indeed lower the runtime overhead. The overhead for the in-house version of ComAir is also measured (the "Overhead" column). The overhead can be as large as 1252X for Mozilla#416628. For 32 out of 38 bugs, the overhead is larger than 10X. These results indicate that *the production-run version of ComAir can generate results as accurate as the in-house version, but with a significantly lower overhead.*

5.3 Discussion and Limitations

The current design and implementation of ComAir only consider complexity problems in single thread. To profile multi-threaded programs, ComAir needs extra synchronizations on all utilized global data structures, and these synchronizations may generate more overhead. ComAir also needs more advanced input metrics to precisely profile programs in the multi-threaded producer-consumer model discussed in previous work [10].

Although the performance of the production-version is good on all evaluated benchmarks, if a sampled dynamic instance conducts a lot of computation, ComAir may encounter a performance corner case and can still incur non-tolerable slowdown. For example, if a profiling target is a loop and one sampled instance of the loop contains a lot of iterations, ComAir may cause observable slowdown. To handle this problem, the next version of ComAir could be enhanced by recording dynamic information for limited number of iterations of each sampled loop instance.

6 RELATED WORK

Empirical studies on performance problems. Many previous studies were conducted on different types of real-world performance bugs [16, 17, 27, 34, 40, 41, 49, 51]. Similar to our work in Section 3, these studies have important findings and implications that can guide technical design to combat performance bugs from various aspects, such as detecting previously unknown bugs [17, 27, 49], diagnosing performance failures [40, 41] and optimizing existing programs [50]. However, our work is the first empirical study focusing on complexity problems, and it provides important supplement to existing studies.

Traditional and algorithmic profilers. Traditional profilers are the most widely used tools during performance optimization and debugging. After collecting runtime information, traditional profilers associate performance metrics to executed instructions, functions, or calling context [1, 15, 38]. Many research works are proposed to improve accuracy of profilers [19, 31, 39], or to reduce runtime overhead [53] and memory overhead of profilers [11]. However, traditional profilers can only analyze one single program run, but cannot connect results from multiple runs and cannot predict results for inputs not seen before. Aprof [9, 10] and AlgoProf [52] are existing algorithmic profiling techniques. As discussed in Section 2, these two techniques suffer from three limitations. ComAir enhances existing techniques through significantly reducing runtime overhead and providing an effective ranking mechanism. ComAir can accurately point out root causes and can be deployed in production environment.

Performance bug detection. There are many performance bug detectors, leveraging static or dynamic techniques to identify bugs matching specific inefficiency patterns [12, 17, 26, 32, 33, 35, 37, 44–48]. Some detected bugs by them overlap with complexity problems. For example, all confirmed performance bugs detected by Toddler are in $O(N^2)$ complexity and used in our evaluation in Section 5. However, static detectors cannot provide any indication about bugs' performance impact. Dynamic detectors usually incur more than 10× overhead, and cannot be applied in production environment. ComAir can detect complexity problems in production runs and also provide information about workloads and bugs' performance impact to help developers make better priority decisions.

7 CONCLUSION

In this paper, we present ComAir, an automated tool that can effectively conduct algorithmic profiling under both in-house and production-run settings. ComAir enhances existing algorithmic profiling techniques by (1) providing a ranking mechanism to effectively identify root causes for performance failures due to complexity problems, and (2) significantly lowering runtime overhead for enabling algorithmic profiling in production environment. The evaluation on 38 benchmark programs shows that, with our designed instrumentation methods, ComAir can accurately identify the root causes of real complexity problems while incurring a negligible overhead under the production-run setting.

REFERENCES

[1] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *PLDI*.

[2] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-run software failure diagnosis via hardware performance counters. In *ASPLOS*.

[3] Thomas Ball. 1994. Efficiently Counting Program Events with Support for On-line Queries. *ACM Trans. Program. Lang. Syst.* (1994).

[4] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *MICRO*.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*.

[6] John M. Chambers, William S. Cleveland, Beat Kleiner, and Paul A. Tukey. 1983. *Graphical Methods for Data Analysis.* Wadsworth.

[7] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS*.

[8] Emilio Coppa. 2014. An interactive visualization framework for performance analysis. *EAI Endorsed Trans. Ubiquitous Environments* (2014).

[9] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-sensitive Profiling. In *PLDI*.

[10] Emilio Coppa, Camil Demetrescu, Irene Finocchi, and Romolo Marotta. 2014. Estimating the Empirical Cost Function of Routines with Dynamic Workloads. In *CGO*.

[11] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2011. Mining Hot Calling Contexts in Small Space. In *PLDI*.

[12] Yufei Ding and Xipeng Shen. 2017. GLORE: Generalized Loop Redundancy Elimination Upon LER-notation. In *OOPSLA*.

[13] Lu Fang, Liang Dou, and Guoqing Xu. 2015. PerfBlower: Quickly Detecting Memory-Related Performance Problems via Amplification. In *ECOOP*.

[14] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI*.

[15] gprof. [n. d.]. GNU gprof. http://sourceware.org/binutils/docs/gprof/. ([n. d.]).

[16] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. 2014. Performance Regression Testing Target Prioritization via Performance Risk Analysis. In *ICSE*.

[17] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *PLDI*.

[18] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. 2010. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*.

[19] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch Me if You Can: Performance Bug Detection in the Wild. In *OOPSLA*.

[20] Jyoti Bansal. 2013. Why is my state's ACA healthcare exchange site slow? http://blog.appdynamics.com/news/why-is-my-states-aca-healthcare-exchange-site-slow/. (2013).

[21] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. 2010. Finding latent performance bugs in systems implementations. In *FSE*.

[22] Chung Hwan Kim, Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, Xiangyu Zhang, and Dongyan Xu. 2014. IntroPerf: Transparent Context-sensitive Multi-layer Performance Inference Using System Stack Traces. *SIGMETRICS*.

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*.

[24] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. 2003. Bug Isolation via Remote Program Sampling. In *PLDI*.

[25] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation. In *PLDI*.

[26] Tongping Liu and Emery D. Berger. 2011. SHERIFF: Precise Detection and Automatic Mitigation of False Sharing. In *OOPSLA*.

[27] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *ICSE*.

[28] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*.

[29] Catherine McGeoch, Peter Sanders, Rudolf Fleischer, Paul R. Cohen, and Doina Precup. 2002. Experimental Algorithmics. Chapter Using Finite Experiments to Study Asymptotic Performance.

[30] Mozilla bugzilla. [n. d.]. Mozilla#35294. https://bugzilla.mozilla.org/show_bug.cgi?id=35294. ([n. d.]).

[31] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the Accuracy of Java Profilers. In *PLDI*.

[32] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. In *FSE*.

[33] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *ICSE*.

[34] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, Reporting, and Fixing Performance Bugs. In *MSR*.

[35] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-Access Patterns. In *ICSE*.

[36] Peter Ohmann and Ben Liblit. 2013. Lightweight Control-flow Instrumentation and Postmortem Analysis in Support of Debugging. In *ASE*.

[37] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *PLDI*.

[38] oprofile. [n. d.]. OProfile – A System Profiler for Linux. http://oprofile.sourceforge.net. ([n. d.]).

[39] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. AppInsight: Mobile App Performance Monitoring in the Wild. In *OSDI*.

[40] Linhai Song and Shan Lu. 2014. Statistical Debugging for Real-world Performance Problems. In *OOPSLA*.

[41] Linhai Song and Shan Lu. 2017. Performance Diagnosis for Inefficient Loops. In *ICSE*.

[42] Wikipedia. [n. d.]. Coefficient of determination. https://en.wikipedia.org/wiki/Coefficient_of_determination. ([n. d.]).

[43] Wikipedia. [n. d.]. Mark and recapture. https://en.wikipedia.org/wiki/Mark_and_recapture. ([n. d.]).

[44] Xusheng Xiao, Shi Han, Tao Xie, and Dongmei Zhang. 2013. Context-Sensitive Delta Inference for Identifying Workload-Dependent Performance Bottlenecks. In *ISSTA*.

[45] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: profiling copies to find runtime bloat. In *PLDI*.

[46] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding low-utility data structures. In *PLDI*.

[47] Guoqing Xu and Atanas Rountev. 2010. Detecting inefficiently-used containers to avoid bloat. In *PLDI*.

[48] Guoqing Xu, Dacong Yan, and Atanas Rountev. 2012. Static Detection of Loop-invariant Data Structures. In *ECOOP*.

[49] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *CIKM*.

[50] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *CIKM*.

[51] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2012. A Qualitative Study on Performance Bugs. In *MSR*.

[52] Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic Profiling. In *PLDI*.

[53] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. 2006. Accurate, Efficient, and Adaptive Calling Context Profiling. In *PLDI*.