IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 11, 2019, accepted August 7, 2019, date of publication August 21, 2019, date of current version September 5, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2936599

Detecting Performance Bottlenecks Guided by Resource Usage

SHANSHAN LI, ZHOUYANG JIA, YUNFENG LI, XIANGKE LIAO, ERCI XU, XIAODONG LIU, HAOCHEN HE, AND LONG GAO
College of Computer Science, National University of Defense Technology, Changsha 410072, China

Corresponding author: Zhouyang Jia (jiazhouyang@nudt.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB1001802, and in part by the NSFC under Grant 61872373 and Grant 61872375.

ABSTRACT Detecting performance bottlenecks is critical to fix software performance issues. A great part of performance bottlenecks are related to resource usages, which can be affected by configurations. To detect configuration-related performance bottlenecks, the existing works either use learning methods to model the relationships between performance and configurations, or use profiling methods to monitor the execution time. The learning methods are time-consuming when analyzing software with large amounts of configurations, while the profiling methods can incur excessive overheads. In this paper, we conduct empirical studies on configurations, performance and resources. We find that 1) 49% performance issues can be improved or fixed by configurations; 2) 71% configurations affect the performance by tuning resource usage in a simple way; and 3) four types of resources contribute the main causes of performance issues. Inspired by these findings, we design *PBHunter*, a resource-guided instrumentation tool to detect configuration-related performance bottlenecks. *PBHunter* ranks configurations by resource usage and selects the ones that heavily affect resource usages. Guided by selected configurations, *PBHunter* applies the code instrumentation technique in resource-related code snippets. The evaluation shows *PBHunter* can effectively (36/50) expose the culprits of performance issues with minor overheads (5.1% on average).

INDEX TERMS Software performance, resource management, software tools.

I. INTRODUCTION

Performance is one of the most important metrics in software systems. Bad performance can negatively affect the software efficiency, thus damage the user experience. When software encounters performance issues, the maintainers need to diagnose the root causes and fix the problems in the very first time. Existing works show that a large proportion of performance issues are caused by misconfigurations [1]–[3]. This is because software configurations can easily affect its performance, while users with limited domain knowledge may frequently setup improper configurations. In this regard, automatic tools with the capability of detecting root causes of configuration-related performance issues can help the diagnosing and fixing processes.

There has been much research on addressing problems related to performance issues. Many techniques are targeted at detecting of performance bottlenecks by using profiling methods [4]–[9]. Taking HPROF as an example, HPROF is a widely-used Java profiling tool for heap and CPU profiling shipped with every JDK release. The profiling methods, however, may be limited in practice, since 1) the straightforward profiling strategy can incur excessive overheads; 2) performance issues may be only triggered under certain workloads or revealed in specific environments. There have been some works focusing on building performance models that describe the relationship between configurations and the software performance [10]–[18]. The building process are time-consuming, when analyzing software with large amounts of configurations. This is because the sheer number of software configurations and the complexity of the constraints among configurations make it difficult to test combinations in a brute-force manner.

Performance issues can be caused by 1) inefficient implementations of source code, also known as performance bugs; 2) resource conflicts or resource shortages, which lead to the performance bottlenecks. The performance bugs are well studied by many existing works [19]–[22], while the

The associate editor coordinating the review of this article and approving it for publication was Qiang Ni.

```
/* org\apache\hadoop\hdfs\BenchmarkThroughput.java */
private int BUFFER_SIZE;

public int run(String[] args) throws IOException{
      ...
      BUFFER_SIZE = conf.getInt
            ("dfsthroughput.buffer.size", 4*1024);
      ...
}

private Path writeLocalFile(String name,
            Configuration conf,
            long total) throws IOException{
      ...
      byte[] data = new byte[BUFFER SIZE];
      ...
}
```

FIGURE 1. The configuration *dfsthroughput.buffer.size* affects resource usage.

tools on performance bottlenecks are limited by overhead, workload and efficiency. In this paper, we focus on the performance bottlenecks. Our assumption is that as performance bottleneck is ultimately bounded by certain types of resources (e.g. allocated memory), studying resource-related configurations and a lightweight instrumentation inside the corresponding code segments are sufficient for exposing bottlenecks.

To verify our assumption, we investigate 150 real-world performance issues from 4 popular Java distributed projects, ZooKeeper, HDFS, Hadoop Common, and MapReduce, which require for high performances. We find that 49% of performance issues are related to configurations. This finding is close to the conclusion that 59% performance issues are related to configurations by previous study on C/C++ non-distributed projects [1]. By analyzing the source code and the bug reports, we also find configurations affect the performance through influencing the usage of four types of resources, and the most affections are implemented in a simple way. For example, as shown in Figure 1, *getInt()* reads the value of *dfsthroughput.buffer.size* and assign this value to BUFFER_SIZE in *run()*, then the value is used to allocate memory resource in *writeLocalFile*.

Based on our observations, we explore an approach to detect performance bottlenecks from resource perspective. First, we made a resource dependence analysis between configuration and each type of resources. And the configuration options with heavy impact on resources are selected as suspicious culprits for performance degradation. Then, we conduct a profiling method guided by the suspicious culprits for bottleneck detection. Our approach focuses on configurations that consume more software resources, thereby we need less instrumentations compared to traditional profiling methods. Traditional profiling methods instrument source code or byte-code to measure the running time of methods. Obviously, these instrumentations result in additional overheads which exacerbate performance issues and may also unintentionally mask the true performance bottlenecks.

There are two main challenges to address. First, it is difficult to identify the most resource-related configuration options since they may affect various types of resources in different ways. Second, it is hard to quantitatively analyze the impact of configuration options on the usage resources as configuration option affects not only data flow but also control flow.

To address these challenges, we analyze the relationship among configuration options, resources and performance and score the statements based on the extent of resource usage. To quantitatively analyze the impact of configurations on the usage of resources, we introduce a configuration ranking approach by accumulating the score of the related statements. We also explore an in-house testing approach that estimates the probability of branch and loop executions. We design and implement a tool, *PBHunter*, to automatically detect configuration-related performance bottleneck in software projects.

In summary, our contributions are as follows:

- We carry out an empirical study on performance issues. The results verify that configuration options affect software performance by influencing the usage of resource and the most-related configuration options are more likely to be suspicious culprits of performance issues.
- We design and implement *PBHunter*, a resource-guided instrumentation tool to detect configuration-related performance bottlenecks. *PBHunter* ranks configurations by resource usage and selects the ones that heavily affect resource usage. Guided by selected configurations, *PBHunter* applies a lightweight instrumentation.
- We verify the effectiveness of *PBHunter* to detect performance bottleneck in software. The experimental results demonstrate that *PBHunter* can effectively (36/50) expose the culprits of performance issues. The extra overhead is less than 5.1%.

II. RELATED WORK

A. PROFILING SOFTWARE PERFORMANCE

Profiling tools are critical for understanding and diagnosing performance bugs. There is a large number of research on the detection of bottleneck [4]–[9]. Shen *et al.* [5] implemented GA-Prof that combines a search-based heuristic with contrast data mining of execution traces to accurately determine performance bottlenecks. Schur *et al.* [6] presented ProCrawl, a generic approach to mine behavior models from web applications. ProCrawl observes the behavior of the application through its user interface, generates and executes tests to explore unobserved behavior. Chilimbi *et al.* [7] described a statistical debugging tool called HOLMES that isolates bugs by finding paths that correlate with failure. HOLMES can use iterative, bug-directed profiling to lower execution time and space overheads. Liu *et al.* [8] designed and implemented AutoAnalyzer that automates the process of debugging performance problems of SPMD-style parallel programs, including data collection, performance behavior analysis, locating bottlenecks, and uncovering their root causes. Han *et al.* [9] proposed a novel approach StackMine that mines call-stack traces to help performance analysts effectively discover highly impactful performance bugs.

These profiling methods can incur excessive overheads (e.g., more than 200%). Our evaluation has shown that *PBHunter* can expose the culprits of performance issues with minor extra overheads (5.1% on average).

B. BUILDING PERFORMANCE MODELS

The performance model can describe the relationship between configuration options and the performance of the software. There is a lone line of research has been undertaken to establish such models in order to help developers predict performance through various sampling and machine learning methods [10]–[18]. Guo *et al.* [10], Sarkar *et al.* [11], and Siegmund *et al.* [12] predicted system performance based on learning influences of individual configuration options and combinations of configuration options. Nair *et al.* [13] improved the above works by measuring a few configurations of a configurable software system and to make statements about the performance of its other configurations. Siegmund et al. [14] proposed an approach that derives a performance-influence model for a given configurable system, describing all relevant influences of configuration options and their interactions. Medeiros *et al.* [15] presented a comparative study of 10 state-of-the-art sampling algorithms regarding their fault-detection capability and size of sample sets. Guo *et al.* [16] proposed a data-efficient learning approach, called DECART, that combines several techniques of machine learning and statistics for performance prediction of configurable systems. Grechanik *et al.* [17] proposed an adaptive, feedback-directed learning testing system for finding performance problems in applications automatically using black-box software testing. Jamshidi *et al.* [18] learned the model using samples from simulators that approximate performance of the real system at low cost. The learning methods are time-consuming when analyzing software with large amounts of configurations (e.g., tens of days). Our evaluation has shown that *PBHunter* can efficiently (less than 11 minutes) expose the culprits of performance issues.

C. DETECTING PERFORMANCE BUG

There are several techniques that leverage dynamic or static analysis to detect performance problems [19]–[22]. Nistor *et al.* [19] presented CARAMEL, a static technique that detects and fixes performance bugs that have non-intrusive fixes likely to be adopted by developers. Nistor *et al.* [20] presented TODDLER, an automated oracle for performance bugs, which enables testing for performance bugs to use the well established and automated process of testing for functional bugs. Song and Liu [21] designed a root-cause and fix-strategy taxonomy for inefficient loops, and a static-dynamic hybrid analysis tool, LDoctor, to provide accurate performance diagnosis for loops. Attariyan *et al.* [22] introduced performance summarization, a technique for automatically diagnosing the root causes of performance problems. Jin *et al.* [28] conducted a comprehensive study of 109 real-world performance bugs that are randomly sampled from five representative software suites. These works focus on troubleshooting the root causes of performance issues. *PBHunter* is used for detecting software performance bottlenecks, which may not be the root causes of performance issues. *PBHunter* is a complementary tool for these works.

D. DIAGNOSING MISCONFIGURATION

Much previous work has proposed using static program analysis to identify and fix incorrect or abnormal configurations [29]–[33]. Attariyan and Flinn [29] built a tool called ConfAid that instruments application binaries to monitor the causal dependencies introduced through control and data flow as the program executes. ConfAid uses these dependencies to link the erroneous behavior to specific tokens in configuration files. Yuan *et al.* [30] presented CODE, a tool that automatically detects software configuration errors. CODE is based on identifying invariant configuration access rules that predict what access events follow what contexts. Zhang *et al.* [31] presented EnCore that automatically detects software misconfigurations. EnCore takes into account two factors: the interaction between the configuration settings and the executing environment, and the correlations between configuration entries. Keller *et al.* [32] presented ConfErr, a tool for testing and quantifying the resilience of software systems to human-induced configuration errors. Xu *et al.* [33] built SPEX to automatically infer configuration requirements from software source code, and then use the inferred constraints to: expose misconfiguration vulnerabilities, and detect certain types of error-prone configuration design and handling. These methods are primarily aimed at function-related misconfigurations and do not apply to performance problems, while *PBHunter* helps users to detects performance bottlenecks.

III. EMPIRICAL STUDY

Several existing works [1]–[3] study real-world performance issues. For example, 59% performance issues are related to configurations [1]. Previous works only paid attention to the relationships between performance and configurations, but did not consider how configurations affect performance. Our assumption is that configurations can affect performance by tuning resource usage, since most performance bottlenecks are ultimately bounded by certain types of resources. In this regard, we conduct empirical studies on the relationships among performance, configurations and resources. Our studies include the following research questions:

RQ1: What is the percentage of performance issues related to configuration?

RQ2: How do the configurations affect resource usage?

RQ3: Which types of resources contribute the main causes of performance issues?

These empirical studies are conducted on four widely-used software projects written in Java language: ZooKeeper, HDFS, Hadoop Common, and MapReduce. To answer RQ1, we analyze 150 performance issues from the issues tracking

systems and select the performance issues that could be improved or fixed by configurations. To answer RQ2, we analyze all 516 bool and numeric types of configurations of the target programs, and select the ones that can affect resource. To answer RQ3, we analyze bug reports and patches of configuration-related performance issues sifted by RQ1, and determine the type of resource leading to performance degradation.

A. RQ1: WHAT IS THE PERCENTAGE OF PERFORMANCE ISSUES RELATED TO CONFIGURATION?

The existing work [1] studied the percentage of performance issues related to configuration on three C/C++ non-ditributed programs, while we study this question on four Java ditributed programs. We search the issue tracking system for each software using a set of performance-related keywords (e.g., performance, slow, speed, regression, degradation, throughout, efficiency), and get 10534 bugs in total. We randomly select bugs to analyze whether it is a performance issue. We abandon the bugs without detailed descriptions and comments for reproducing the performance issue or pointing out the culprit. We manually check them by three authors until get 150 performance issues, which are enough to prove the reliability of our research compared to previous studies [1], [2].

We further analyze the bug reports and patches of the issues to determine whether these performance issues are configuration-related. We consider a performance issue is related to configurations, when its bug report or comment clearly points out that this bug is related to a certain configuration. On the other hand, we analyze the patches of the performance issue. If one patch modifies the value of the configuration option or the method related to the configuration option to fix the performance issue, we consider this as a configuration-related performance issues.

The results are shown in Table 1. For example, in ZooKeeper, we totally find 779 issues by keyword searching, and analyze 57 issues. Among them, 20 issues are related to performance, while eight are configuration-related performance issues. The configuration-related performance issues account for 49% on average, which is close to the conclusion of 59% of the previous survey [1].

> Finding 1: A great part of (49%) performance issues can be improved or fixed by configurations. Detecting configuration-related performance bottlenecks helps to fix about half of performance issues.

B. RQ2: HOW DO THE CONFIGURATIONS AFFECT PERFORMANCE?

In order to verify that whether a configuration affects performance through tuning resource usage, we analyze all 516 bool and numeric types of configurations from the four target programs to analyze how configuration affect performance.

TABLE 1. Performance issues.

Software	#Issues	#Used	#Perf	#Conf	#Conf/#Perf
ZooKeeper	779	57	20	8	55%
HDFS	4151	188	83	40	48%
Hadoop	3873	70	34	18	53%
MapReduce	1731	50	13	7	54%
Total	10534	365	150	73	49%

#Issues: the number of bug reports by searching key words;
#Used: the number of analyzed bug reports;
#Perf: the number of performance issues;
#Conf: the number of configuration-related performance issues.

TABLE 2. Partial resource-related methods.

Type	Examples
MEM	List, Set,Map,String,Byte,Arrays,Collection
NET	URL, Socket, SocketAddress, INetAddress
IO	Write, File,InputStream,OutputStream

TABLE 3. Resource-related configurations.

Software	#Config	#ResCongfig	Proportion
ZooKeeper	12	8	75%
HDFS	268	214	79%
Hadoop Common	137	94	68%
MapReduce	99	50	50%
Total	516	366	71%

#Config: the number of bool and numeric types of configurations;
#ResCongfig: the number of configuration options related to resource.

This is a common practice, since bool and numeric types of configurations are more likely to affect software performance. We check if a *resource-related method* is included in the statements affected by the configuration. First, we look for resource-related methods from JDK by analyzing the JDK documentation. We consider a method is resource-related, when the documentation shows that this method is related to resource usage (e.g, *Socket()* for network, *List()* for memory). Table 2 displays partial resource-related methods. Second, we obtain the statements dependent on the configuration by slicing. If these statements contain a resource-related method, the configuration is considered to be related to resource.

Table 3 shows the number of studied configurations and the number resource-related configurations for each program. For example, 8 of 12 configurations in ZooKeeper are resource-related. On average, 76% configurations can affect the performance through tuning resource usage.

> Finding 2: More than 50% of configurations can affect the performance through tuning resource usage. This finding verifies our assumption that configurations can affect performance by tuning resource usage.

For the resource-related configurations in Table 3, we manually analyze all related source code from the reading method of the configuration. We divide the way the configurations affect resources into two categories based on the

```
/* Example of confOne */

/* hadoop/io/compress/DefaultCodec.java */
public CompressionOutputStream createOutputStream
      (OutputStream out, Compressor compressor)
      throws IOException{

      return new CompressorStream(out, compressor,conf.
            getInt("io.file.buffer.size", 4*1024));
}

/* hadoop/io/compress/CompressorStream.java */
public CompressorStream (OutputStream out,
      Compressor compressor, int bufferSize){
      super (out);

      if (out==null || compressor==null)
            throw new NullPointerException();
      else if (bufferSize <= 0)
            throw new
IllegalArgumentException("Illegal...");

      this.compressor=compressor;
      buffer = new byte[bufferSize];
}
```

```
/* Example of confMul */

/* zookeeper/server/quorum/leader.java */
long start = System.currentTimeMillis();
long cur = start;
long end = start+self.getInitLimit()*
            self.getTickTime();

while(!electionFinished && cur<end){
      electingFollowers.wait(end-cur);
      cur=System.currentTimeMillis();
}
```

FIGURE 2. Examples of *confOne* and *confMul*.

number of configurations affect resource usage. 88% usage of resources are affected by only one configuration option (*confOne*), and 12% usage of resources are affected by two or more configuration options (*confMul*). *confOne* usually directly controls the execution path (e.g., *if, while, etc.*), size (e.g., *buffer*), or threshold (e.g., *thread count*). For example, as shown in Figure 2, the value of *bufferSize* (the size of the buffer of I/O) is determined by the value of the configuration option, *io.file.buffer.size*. On the other hand, *confMul* means multiple configurations affect a resource at the same time, and multiple configuration options interact with each other. And as shown in Figure 2, *end* = *start* + *self.getInitLimit()* * *self.getTickTime()*, where *getInitLimit* get the value of configuration *initLimit*, *getTickTime* get the value of configuration *tickTime* and *start* is an initial value of system time. When *electionFinished* is set to false, the loop waits until $cur \geqslant end$.

> Finding 3: 88% of resource-related configurations affect the resource usage in a simple way. An automation tool can get the resource usage of one configuration by analyzing the source code dependent on that configuration.

C. RQ3: WHICH TYPES OF RESOURCES ARE THE MAIN CAUSES OF PERFORMANCE ISSUES?

We analyze bug reports and source code of each configuration-related performance issues to determine the types of resource leading to performance degradation. We classify these issues into four categories based on the type of resource: 1) MEM-related, 2) CPU-related, 3) IO-related, and 4) NET-related. For memory-related performance issues, performance bottleneck is ultimately bounded by a shortage in buffer/cache or lack of memory. For example, in MAPREDUCE-6551, dealing with many small files leads to many *map* tasks. In gengeral, a *map* task will use the default config *MRJobConfig#MAP_MEMORY_MB* to set its memory capacity. In this case, the *map* tasks cost so much memory resource for the massive small files. For CPU-related performance issues, invalid/inefficient calculations or lacks of CPU lead to software performance bottlenecks. For instance, in Hadoop-14216, the performance regression is caused by parsing the XML file. This wastes CPU resources and makes the software running time longer. The XML parsing performance can be improved by reusing and making changes in the XML parser (STAX). For IO-related performance issues, performance bottlenecks are related to the IO operations. While for NET-related performance issues, performance bottlenecks are caused by network bandwidth.

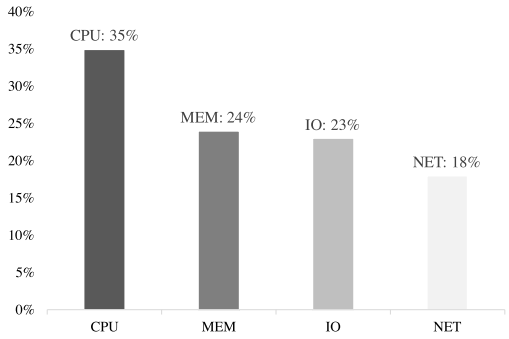


FIGURE 3. Resource classification.

Figure 3 shows the proportions of each type of resource. The configuration-related performance issues are related to the usage of CPU (35%), memory (24%), IO (23%), and NET (18%).

> Finding 4: Four types (i.e., CPU, IO, MEM and NET) of resources contribute the main causes of performance issues. For each configuration, we need to consider four types of resource usages.

IV. DESIGN

In this section, we introduce the design of *PBHunter*. Figure 4 illustrates its architecture. *PBHunter* consists of two main phases. The resource dependence analysis phase ranks all configurations according to their impact on the resources usage. First, *PBHunter* gets the statements affected by each configuration option. Then, for each configuration option, *PBHunter* accumulates the score of the statements (*w.r.t.* the impact of the statement on the resource) that the configuration option affects. Finally, according to the score of each configuration option, *PBHunter* ranks all configuration options and selects the top resource-related configuration options as suspicious culprits of performance issues. These suspicious

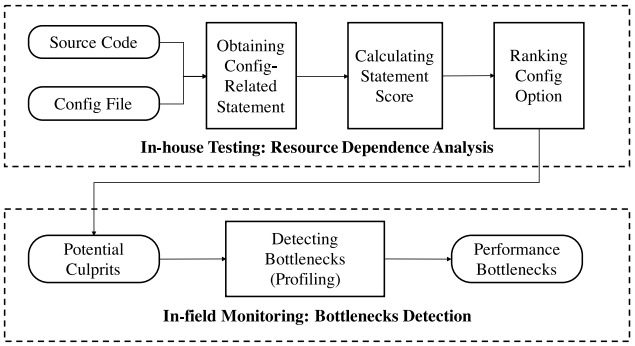


FIGURE 4. The architecture of *PBHunter*.

culprits serve as inputs in dynamic phase. In dynamic analysis phase, *PBHunter* detects performance bottlenecks related to these suspicious culprits obtained by resource dependence analysis. *PBHunter* monitors the resource consumption of statements affected by these suspicious culprits by profiling, then reports the bottlenecks as well as corresponding configuration options to the users.

A. RESOURCE DEPENDENCE ANALYSIS

In the resource-dependence-analysis-phase, we rank the configuration options according to their level of influence concerning the usage of resources, and select the top resource-related configuration options as suspicious culprits for performance degradation. To achieve this goal, we quantify the influence of all code blocks affected by the configuration option on resources, and denote this influence as the score of this option. We divide the analysis into two steps. First, we leverage inter-process slicing to obtain the statements that the configuration option affects. Second, we calculate the score of each statement by in-house testing approach, and rank the configuration options by adding up the scores of affected statements. This in-house testing approach does not depend on the real workload, meaning the process of ranking configuration options is workload-insensitive.

1) OBTAINING CONFIGURATION-RELATED STATEMENT

PBHunter uses the program slicing tool, WALA [23], to obtain the statements affected by a given option. This tool requires a seed statement as the start of slicing. *PBHunter* regards the statement that reads the option as the seed statement. More than 90% configuration options have *get* and *set* methods, and these *get* methods follow a good pattern. For example, in Figures 2, *getInitLimit* is the read method of *initLimit*, *getTickTime* is the read method of *tickTime* and *getInt("io.file.buffer.size", 4*1024))* is the read method of *io.file.buffer.size*. We summarize the following two patterns to recognize the read methods: (1) *get* + *Option Name*, (2)*get* + *Option Type (Option Name)*.

2) CALCULATING STATEMENTS SCORE

For each configuration, *PBHunter* accumulates the score of the statements (*w.r.t.* the impact of the statement on the resource) that the configuration affects. To achieve this, we transfer the statements to bytecode instructions with WALA(Shrike) [23]. First, we predefine the score of each base instruction by reading official documents of different processors [24], [25]. Since the NET-related resource can be regarded as a subtype of IO-related resource, we do not predefine the score for NET-related resource. Table 4 shows the scores of partial type of instructions for each resource. The score represents to what degree this instruction affects one type of resource. We use $OIns_{Type}$ to represent the score of different *Type* of each instruction. For example, a DIV operation is slower than an ADD operation, so the $OIns_{DIV}$ is larger than the $OIns_{ADD}$.

TABLE 4. The scores of partial type of base instruction we used in our experiment for different types of resources.

Type	Score			Type	Score		
	CPU	MEM	IO		CPU	MEM	IO
Store	2	1	1	Load	2	1	1
ArrayLoad	2	10	10	Get	2	1	1
LoadIndirect	1	1	1	Pop	2	5	1
Conversion	3	1	1	New	4	2	1
ArrayStore	2	10	10	Swap	5	5	1
StoreIndirect	1	1	1	Dup	2	5	1
Goto	5	1	1	DIV	10	2	1
Throw	10	1	1	MUL	5	2	1
REM	15	2	1	ADD	5	2	1
SUB	5	2	1	AND	3	2	1
OR	3	2	1	XOR	3	2	1
Monitor	2	2	1	SHL	1	2	1
USHR	1	2	1	SHR	1	2	1
CMPG	3	2	1	CMP	3	2	1
CMPL	3	2	1	NEG	1	2	1
Throw	5	2	1	NE	5	2	1
Return	5	1	1	GE	5	2	1
TypeTest	10	2	1	LE	5	2	1
Switch	3	2	1	GT	5	2	1
ArrayLength	5	2	1	EQ	5	2	1
Constant	5	2	1	LT	5	2	1
Instanceof	5	2	1	-	-	-	-

The score of an instruction can also be affect by its context. For example, the instruction is located in a loop or branch statement. In this case, we need to estimate the number of instruction executions, which has a significant influence on the score of each configuration option. To achieve this, we use the instrumentation technique to count the number of loop executions and to monitor the branch path based on in-house testing [26]. We test each software with an existing test set or testing tools. Then, we take the averaged iteration times as the execution number for each loop, and calculate the average probability as the execution number for each branch.

Loop: We insert one counter into each nesting loop for each method to record the number of loop executions. Then, we run all test cases in the test set to traverse as many as paths. Based on the output, we take the averaged value of each loop execution as the base value of the loop. For example, the 5th-16th instructions are *Loop* structures, and the execution number of these instructions is 10 (*NumberOfLoop*) in Figure 5.

Branch: We consider the following two branch control structures: (1) *if else*, (2) *switch case*. We WALA's bytecode library to convert these branch control structures to bytecode

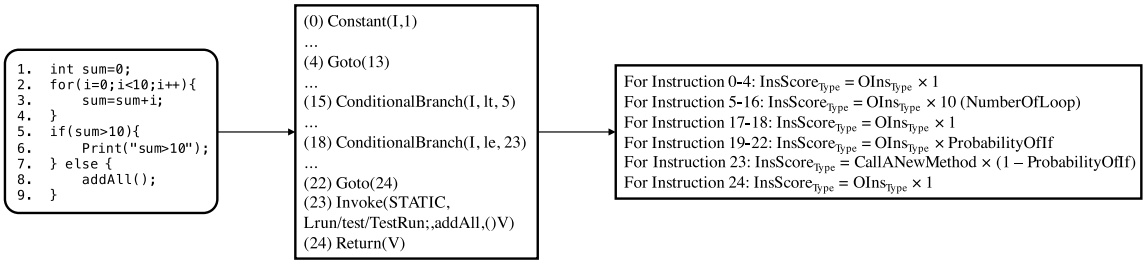


FIGURE 5. Scoring a code block affected by configuration option (*InsScore$_{Type}$* represents the score of different *Type* of instruction; *OIns$_{Type}$* represents the score of different *Type* of base instruction; In this figure, *Type* is the type corresponding to each instruction.).

and instrumentation on bytecode. For each branch, we instrument additional bytecode to monitor which path is executed. Furthermore, based on the execution numbers of different branches, we calculate the probability of each branch. For example, the 19th-23th instructions are *if* branch structures, and the number of executions of these instructions needs to be multiplied by the probability (*ProbabilityOfif*) of the *if* branch in Figure 5.

According to these results, we can calculate the score of each instruction which is located in the loop or branch as a description in (1). Where $InsScore_{Type}$ is the score of an instruction, $OIns_{Type}$ represents the score of this *Type* of base instruction, $\prod_{i=1}^{n} P_i$ is the probability of each nested branch (n represents the nesting layers of control conditional branches), and $\prod_{j=1}^{m} T_j$ is the number of loop executions if this statement is in a loop (m represents the nesting layers of loops). For example, for the 0^{th} instruction *Constant*, the number of executions of this instruction is 1, then the score ($InsScore_{Constant}$) of this instruction for the CPU resource is $OIns_{Constant} \times 1 = 5 \times 1 = 5$ by looking up the Table 4. For the 15^{th} instruction *ConditionalBranch*, the type of this instruction is *LTInstruction* and the execution number is 10, so the score ($InsScore_{LT}$) of this instruction is $OIns_{LT} \times 10 = 5 \times 10 = 50$.

$$InsScore_{Type} = OIns_{Type} \times \prod_{i=1}^{n} P_i \times \prod_{j=1}^{m} T_j \tag{1}$$

3) RANKING THE CONFIGURATION OPTIONS

In order to reduce the number of configuration options that need to be analyzed, *PBHunter* ranks all configuration options according to their scores (the impact of configuration option on resources). Based on the scores of each instruction in previous section, we accumulate the scores of all the statements affected by the configuration option as the score of that configuration option. Thus we score each configuration option as described in (2), where $Score_{conf}$ is the score of a configuration option, $InsScore_i$ is the score of the i^{th} instruction in all instructions affected by the configuration options. Finally, according to the scores, we rank all configuration options.

$$Score_{conf} = \sum_{i=1}^{n} InsScore_i \tag{2}$$

B. BOTTLENECKS DETECTION

In the resource-dependence-analysis phase, we rank all configuration options, and select top ranked ones as suspicious culprits for performance degradation. In this phase, *PBHunter* use profiling [27] to identify bottlenecks related to the suspicious culprits. Profiling is a form of dynamic program analysis that measures the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of method calls. One way to implement profiling is instrumentation [27]. Instrumentation refers to an ability to monitor or measure the level of a product's performance, to diagnose errors, and to write trace information.

PBHunter makes use of instrumentation to monitors the resource consumption (*running times for CPU, Memory consumption for MEM, IO request for IO*) of the code blocks affected by suspicious culprits. First, *PBHunter* obtains the code blocks affected by configuration options using static slicing. Then, *PBHunter* instrument these code blocks to monitor the resource consumption. Third, *PBHunter* runs test cases for each software, and obtains the resource consumption of all code blocks affected by suspicious culprits. Based on the amount of resource consumption, *PBHunter* ranks code blocks in a descending order and generates a list of bottlenecks. Finally, the bottlenecks, *w.r.t* methods, and the corresponding configuration options are reported to the users. It would be better if we were able to get the real input and workload that caused the software performance bottleneck. In the experiment, we set the input and load according to the description of the bug report. For the settings not mentioned, we use the default parameters.

V. EVALUATION

In this section, we conduct experiments to evaluate the effectiveness and the overhead of *PBHunter* on detecting software performance bottlenecks. We first evaluate the effectiveness of the resource-dependence-analysis phase on recognizing suspicious culprits, i.e., configurations that heavily affect the resource usage. Then, we evaluate the effectiveness of the bottleneck-detection phase, which uses of instrumentation to monitors the resource consumption of suspicious culprits. After that, we evaluate the overhead of *PBHunter*, and compare *PBHunter* with baseline tools. Finally, we evaluate the parameter in *PBHunter*, i.e., the threshold used to select top ranking configurations.

TABLE 5. Target performance issues and the rankings of bottlenecks (*i.e.*, problem method).

Perdormance Bugs	Resource Type	Culprit(Configuration Option)	*PBHunter* Ranking	HPROF Ranking
ZooKeeper-1583	CPU	maxClientCnxns	20	176
HDFS-3738	CPU	dfs.client.socket.timeout	-	40
HDFS-11412	MEM	dfs.namenode.maintenance.replication.min	8	114
MapReduce-6551	MEM	mapred.job.map.memory.mb	50	317
Hadoop-13263	MEM	hadoop.security.groups.cache.background.reload	14	204
Hadoop-8021	I/O	mapred.output.compres mapred.output.compression.codec	33	283

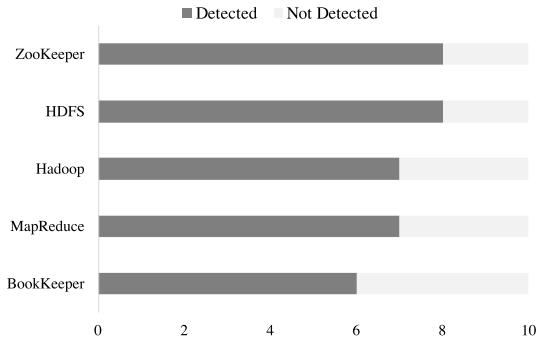


FIGURE 6. The number of performance issues detected by ***PBHunter***.

A. EFFECTIVENESS OF RESOURCE DEPENDENCE ANALYSIS

PBHunter uses resource dependence analysis to select suspicious culprits. For instructions affected by a given configuration, we predefine their scores for each type of resource, which can be affected by contexts like loop and branch. To count the loop number and the probability of branch, we select four widely used benchmark (TestDFSIO, mrbench, WordCount and zk-smoketest) for in-house testing. The TestDFSIO can test the speed of reading and writing the files of HDFS. We select 10 to 100 files (10-100MB each file) for different workload. The mrbench is a MapReduce benchmark, we run 10 to 100 times jobs. The WordCount is the most commonly used test program of Hadoop, and we select *.txt* files as the input. zk-smoketest is the standard benchmark of ZooKeeper.

To illustrate the effectiveness of recognizing suspicious culprits, we use the same method described in Section II to select 10 configuration-related performance issues for ZooKeeper, HDFS, Hadoop Common and MapReduce. These performance issues are different from those we studied in Section II. At the same time, we randomly select 10 configuration-related performance issues of Bookkeeper to further validate whether *PBHunter* is effective for new software.

We select top 5% configuration options as the culprits causing performance degradation, and determine whether the configuration options causing the performance issue is in ranking (top 5%). Figure 6 shows the results. Black indicates that *PBHunter* detects the configuration option that causes the performance issue, and white is the opposite. The results show that *PBHunter* detect 36 culprits of performance issues in 50 cases, and the averaged percentage of five software reaches 72%. The results indicate that *PBHunter* can effectively detect the configurations that cause software performance issues.

B. EFFECTIVENESS OF BOTTLENECK DETECTION

To evaluate the effectiveness of detecting bottlenecks, we need to reproduce the performance issues to obtain the real bottleneck as an oracle. We are still in the process of reproducing the issues in Section V-A. So far, we have successfully reproduced 6 configuration-related performance issues as shown in Table 5, and apply them to *PBHunter*. We compare *PBHunter* to the standard Java profiling tool HPROF, which is a widely-used standard JAVA profiling tool in JDK and we can use it very easily. To reduce the impact of human factors on HPROF, we use default setting of HPROF. The results are shown in Table 5. The number is the ranking of root bottleneck. Taking HDFS-11412 as an example, the root cause is ranked 8^{th} in the ranking by *PBHunter* and 114^{th} by HPROF. The ranking of bottleneck of HDFS-11412 in *PBHunter* is improved by 114/8 times compared with the ranking in HPROF.

PBHunter only instruments the methods that are affected by the suspicious culprits obtained by resource dependence analysis. This strategy can significantly increase the efficiency of diagnosing configuration-related performance issues. The ranking of bottleneck in *PBHunter* can be improved by 10 times compared with the ranking in HPROF on average.

C. EVALUATION OF OVERHEAD

PBHunter uses the instrumentation technique to detect performance bottlenecks, which introduces additional overhead. To measure the overhead, we count the number of instrumentations of *PBHunter* and HPROF. As shown in Table 6, the number of instrumentations inserted by *PBHunter* are 182 in average, while the number of HPROF is 794.

We further monitor the running time of three programs (WordCount, DFSIO-Write, and DFSIO-Read, which are widely used for testing Hadoop projects) in three cases:

TABLE 6. Number of instrumentation.

Software	*PBHunter*	HPROF	Proportion
ZooKeeper	137	805	17.0%
Hadoop	185	727	25.4%
MapReduce	256	941	27.2%
HDFS	150	704	21.3%
Average	182	794	22.9%

TABLE 7. The average and the worst overheads of *PBHunter* and HPROF.

Program	*PBHunter*		HPROF	
	Average	Worst	Average	Worst
WordCount	4.7%	9.0%	119.0%	153.4%
DFSIO-Write	4.7%	10.0%	23.0%	59.1%
DFSIO-Read	5.1%	8.7%	32.1%	69.6%

1) *PBHunter*, 2) HPROF, 3) original source code. For WordCount, we take the *.txt* files as the input and increase the size of the input with the experiment times. For DFSIO-Write and DFSIO-Read, we leverage their example program to test the read/write speed of HDFS with 10, 20, 30, and 40 files (10, 50, 100, 150MB every file). All tests are run 20 times and we take the average time as the last result. As shown in Table 7, *PBHunter* costs 4.7% overhead on average, while HPROF doubles the original execution time in the WordCount cases. As for the DFSIO-Write cases, *PBHunter* also has 4.7% overhead on average, and the worst case is 10%. By contrast, HPROF leads to 23% overhead on average and 59.1% for the worst case. DFSIO-Read is similar to DFSIO-Write, *PBHunter* introduces a 5.1% overhead on average, and the worst case is 8.7%. While the overheads of HPROF are 32.1% and 69.6%. In summary, *PBHunter* outperforms HPROF in these three cases with different workloads, and the expected overhead of *PBHunter* is 15% compared with the overhead of HPROF.

We also measure the efficiency of the resource dependence analysis. We run the resource-dependence-analysis phase on each of the four target programs to obtain their running times. The averaged running time is seven minutes, while the in-house testing is less than four minutes. This result suggests that *PBHunter* is more efficient compared the tools training machine learning models.

D. EVALUATION OF THRESHOLD

The resource-dependency-analysis phase sifts through configuration options that are highly resource-related by using a threshold. We select the top x% configuration options as suspicious culprits. Noise configuration options may be introduced when the threshold is too large. On the other hand, *PBHunter* may miss problem configuration options when the threshold is too small. We set x% to 5%, 10%, 15% and 20% and judge whether the real culprits of 73 configuration-related performance issues (in Section II) are selected as suspicious culprits. Top 10% configuration options detect 50 culprits while top 20% configuration options detect 55. Although 20% can obtain better result, it will introduce more noise configuration options which do not cause performance issues. In this regard, we select top 10% configuration options as suspicious culprits in resource dependence analysis.

We apply *PBHunter* on the target programs, and select top 10% configuration options as suspicious culprits. We manually verify the results by analyzing source code and descriptions of configuration options. The accuracy of resource dependence analysis is more than 75%.

E. DISCUSSION

In this section, we discuss the limitations of *PBHunter*. First, *PBHunter* cannot directly troubleshoot the root causes of performance issues. *PBHunter* monitors software performance bottlenecks based on instrumentation, but performance bottlenecks may not be the root causes of performance issues. When a method takes a long time, it will be identified as a performance bottleneck by *PBHunter*. If this method should have taken a long time and cannot be optimized, *PBHunter* may report a false positive. Second, the experiment results of *PBHunter* will be affected by benchmark. Different inputs and workloads will lead to different ranking of configuration options. Our experiments can not enumerate all potential scenarios. Third, *PBHunter* can be affected by poor coding style. *PBHunter* detects the read methods of configuration options based on the method names. We evaluate the accuracy of identifying read methods of configuration options. The result shows 80% configuration options can map their read methods. In this regard, *PBHunter* provides interface for users to manually specify the read methods.

VI. CONCLUSION

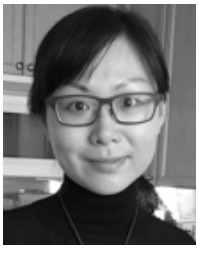
A highly configurable system always allows users to customize their system. However, users often set inappropriate configurations, leading to software performance deterioration. Identifying performance bottlenecks is critical to improve performance of software and troubleshoot performance issues. We study 150 performance issues as well as 516 bool and numeric types of configuration options, and analyze how configuration options affect performance. We find that performance issues can be correlated to resource-related configurations. Based on our study, we design and implement a tool, *PBHunter*, to detect the bottleneck that the configuration options affect by using resources dependence analysis. Our evaluation has shown that *PBHunter* can effectively (36/50) and efficiently (less than 11 mins) expose the culprits of performance issues with minor overheads (5.1% on average).

REFERENCES

[1] X. Han and T. Yu, "An empirical study on performance bugs for highly configurable software systems," in *Proc. 10th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, 2016, pp. 23:1–23:10. [Online]. Available: http://doi.acm.org/10.1145/2961111.2962602

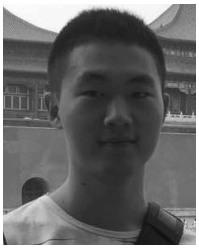
[2] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2018, pp. 154–168. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173206

[3] C. Yan, A. Cheung, J. Yang, and S. Lu, "Understanding database performance inefficiencies in real-world Web applications," in *Proc. ACM Conf. Inf. Knowl. Manage. (CIKM)*, 2017, pp. 1299–1308. [Online]. Available: http://doi.acm.org/10.1145/3132847.3132954
[4] Oracle. (2018). *HPROF*. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr008.html
[5] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik, "Automating performance bottleneck detection using search-based application profiling," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2015, pp. 270–281. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771816
[6] M. Schur, A. Roth, and A. Zeller, "Mining behavior models from enterprise Web applications," in *Proc. 9th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, 2013, pp. 422–432. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491426
[7] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," in *Proc. 31st Int. Conf. Softw. Eng. (ICSE)*, May 2009, pp. 34–44. doi: 10.1109/ICSE.2009.5070506.
[8] X. Liu, J. Zhan, K. Zhan, W. Shi, L. Yuan, D. Meng, and L. Wang, "Automatic performance debugging of SPMD-style parallel programs," *J. Parallel Distrib. Comput.*, vol. 71, no. 7, pp. 925–937, Jul. 2011. doi: 10.1016/j.jpdc.2011.03.006.
[9] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 145–155. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337241
[10] J. Guo, K. Czarnecki, S. Apely, N. Siegmundy, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *Proc. 28th IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE)*, Nov. 2013, pp. 301–311. doi: 10.1109/ASE.2013.6693089.
[11] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems (t)," in *Proc. 30th IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE)*, Nov. 2015, pp. 342–352. doi: 10.1109/ASE.2015.45.
[12] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 167–177. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337243
[13] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Using bad learners to find good configurations," in *Proc. 11th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, 2017, pp. 257–267. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106238
[14] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proc. 10th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, 2015, pp. 284–294. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786845
[15] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, 2016, pp. 643–654. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884793
[16] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu, "Data-efficient performance learning for configurable systems," *Empirical Softw. Engg.*, vol. 23, no. 3, pp. 1826–1867, Jun. 2018. doi: 10.1007/s10664-017-9573-6.
[17] M. Grechanik, C. Fu, and Q. Xie, "Automatically finding performance problems with feedback-directed learning software testing," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, 2012, pp. 156–166. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337242
[18] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar, "Transfer learning for improving model predictions in highly configurable software," in *Proc. 12th Int. Symp. Softw. Eng. Adapt. Self-Manag. Syst. (SEAMS)*, 2017, pp. 31–41. doi: 10.1109/SEAMS.2017.11.
[19] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "CARAMEL: Detecting and fixing performance problems that have non-intrusive fixes," in *Proc. 37th Int. Conf. Softw. Eng. (ICSE)*, vol. 1, 2015, pp. 902–912. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818863
[20] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, 2013, pp. 562–571. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486862
[21] L. Song and S. Lu, "Performance diagnosis for inefficient loops," in *Proc. 39th Int. Conf. Softw. Eng. (ICSE)*, 2017, pp. 370–380. doi: 10.1109/ICSE.2017.41.
[22] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Proc. 10th USENIX Conf. Operating Syst. Design Implement. (OSDI)*, 2012, pp. 307–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387910
[23] (2018). *WALA*. [Online]. Available: http://wala.sourceforge.net/wiki/index.php/Main_Page
[24] (2018). *Alldatasheet*. [Online]. Available: http://pdf1.alldatasheet.com/datasheet-pdf/view/174758/ATMEL/ATMEGA16-16AU.html
[25] Intel. (2018). *Intel 64 and IA-32 Architectures Software Developer Manuals*. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm
[26] Wikipedia1. (2018). *Software Testing*. [Online]. Available: https://en.wikipedia.org/wiki/Software_testing
[27] Wikipedia2. (2018). *Computer Programming*. [Online]. Available: https://en.wikipedia.org/wiki/Profiling_(computerprogramming)
[28] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2012, pp. 77–88. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254075
[29] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *Proc. 9th USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, 2010, pp. 237–250. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924960
[30] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf. (USENIXATC)*, 2011, p. 28. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002181.2002209
[31] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "EnCore: Exploiting system environment and correlation information for misconfiguration detection," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2014, pp. 687–700. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541983
[32] L. Keller, P. Upadhyaya, and G. Candea, "ConfErr: A tool for assessing resilience to human configuration errors," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. FTCS DCC (DSN)*, Jun. 2008, pp. 157–166.
[33] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proc. 24th ACM Symp. Oper. Syst. Princ. (SOSP)*, 2013, pp. 244–259. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522727



SHANSHAN LI received the M.S. and Ph.D. degrees from the College of Computer Science, National University of Defense Technology, Changsha, China, in 2003 and 2007, respectively, where she is currently a Full Professor. She was a Visiting Scholar with the Hong Kong University of Science and Technology, in 2007. Her main research interests include distributed computing, social networks, and software reliability.

ZHOUYANG JIA received the B.S. and M.S. degrees from the College of Computer Science, National University of Defense Technology, Changsha, China, in 2013 and 2015, respectively. He is currently pursuing the Ph.D. degree with the National University of Defense Technology. His main research interests include software reliability, software availability, and so on.



YUNFENG LI received the B.S. and M.S. degrees from the College of Computer Science, National University of Defense Technology, Changsha, China, in 2016 and 2018, respectively. He is currently pursuing the Ph.D. degree with the National University of Defense Technology. His main research interest includes software engineering.

XIANGKE LIAO received the B.S. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 1985, and the M.S. degree from the National University of Defense Technology, Changsha, China, in 1988. He is currently a Full Professor and the Dean of the College of Computer Science, National University of Defense Technology. His research interests include parallel and distributed computing, high-performance computer systems, operating systems, cloud computing, and networked embedded systems.

ERCI XU is currently pursuing the Ph.D. degree with the College of Computer Science, National University of Defense Technology, Changsha, China. His main research interests include software engineering, software reliability, and storage systems.

XIAODONG LIU received the M.S. and Ph.D. degrees from the College of Computer Science, National University of Defense Technology, Changsha, China, in 2009 and 2014, respectively. He is currently an Assistant Professor with the School of Computer Science, National University of Defense Technology. His main research interests include software engineering, software reliability, operating systems, and so on.

HAOCHEN HE received the B.S. degree from the College of Computer Science, National University of Defense Technology, Changsha, China, in 2017. He is currently pursuing the Ph.D. degree with the National University of Defense Technology. His main research interest includes software engineering.

LONG GAO received the Ph.D. degree from the School of Computer Science, National University of Defense Technology, Changsha, China, in 2007. He is currently an Associate Professor with the College of Computer Science, National University of Defense Technology. His main research interests include real-time embedded systems and operating systems.