Performance Bug Analysis and Detection for Distributed Storage and Computing Systems

JIAXIN LI, National University of Defense Technology, China
YIMING ZHANG, [1]National University of Defense Technology, [2]Xiamen University, China
SHAN LU, University of Chicago, USA
HARYADI S. GUNAWI, University of Chicago, USA
XIAOHUI GU, North Carolina State University, USA
FENG HUANG, National University of Defense Technology, China
DONGSHENG LI, National University of Defense Technology, China

This paper systematically studies 99 distributed performance bugs from five widely-deployed distributed storage and computing systems (Cassandra, HBase, HDFS, Hadoop MapReduce and ZooKeeper). We present the *TaxPerf* database, which collectively organizes the analysis results as over 400 classification labels and over 2,500 lines of bug re-description. TaxPerf is classified into six bug categories (and 18 bug subcategories) by their root causes, namely, resource, blocking, synchronization, optimization, configuration, and logic. TaxPerf can be used as a benchmark for performance bug studies and debug tool designs. Although it is impractical to automatically detect all categories of performance bugs in TaxPerf, fortunately we find that an important category of blocking bugs can be effectively solved by analysis tools. We analyze the cascading nature of blocking bugs and design an automatic detection tool called *PCatch*, which (i) performs program analysis to identify code regions whose execution time can potentially increase dramatically with the workload size; (ii) adapts the traditional happens-before model to reason about software resource contention and performance dependency relationship; and (iii) uses dynamic tracking to identify whether the slowdown propagation is contained in one job. Evaluation shows that PCatch can accurately detect blocking bugs of representative distributed storage and computing systems by observing system executions under small-scale workloads.

CCS Concepts: • **Information systems** → **Storage replication**; **Cloud based storage**; • **Software and its engineering** → *Cloud computing*.

Additional Key Words and Phrases: Storage and computing systems performance, blocking bugs.

This work is supported by the National Key Research and Development Program of China (2022YFB4500302), the Scientific Research Program of National University of Defense Technology (ZK20-03), the National Natural Science Foundation of China (61872376, 61932001), and the Natural Science Foundation of Hunan Province of China (2022JJ40555). Yiming Zhang is the corresponding author.
Authors' addresses: Jiaxin Li, National University of Defense Technology, Sanyi Rd., Changsha, Hunan, China, 410073, licyh@nudt.edu.cn; Yiming Zhang, [1]National University of Defense Technology, [2]Xiamen University, Siming Rd., Xiamen, Fujian, China, zhangyiming@nicexlab.com; Shan Lu, University of Chicago, Chicago, USA, shanlu@uchicago.edu; Haryadi S. Gunawi, University of Chicago, Chicago, USA, haryadi@cs.uchicago.edu; Xiaohui Gu, North Carolina State University, USA, xgu@ncsu.edu; Feng Huang, National University of Defense Technology, Sanyi Rd., Changsha, Hunan, China, 410073, fenghuang@nudt.edu.cn; Dongsheng Li, National University of Defense Technology, Sanyi Rd., Changsha, Hunan, China, 410073, dsli@nudt.edu.cn.

1553-3077/2023/1-ART $15.00
https://doi.org/10.1145/3580281

1 INTRODUCTION

Large-scale distributed storage and computing systems, such as distributed key-value stores, scalable file systems, synchronization services, data-parallel computing frameworks, streaming systems, and cluster management services, have become a backbone for the cloud. Unfortunately, the performance and QoS of cloud distributed systems is severely threatened by *performance bugs*. Distributed systems deal with large workloads upon many complicated distributed protocols running on hundreds or thousands of machines and have to deal with a variety of error handling and failure recovery. This makes distributed systems prone to performance bugs caused by various causes such as excessive resource consumption, blocking, synchronization, etc. These causes can lead to severe performance loss symptoms (implications) such as increased latency, wasted resources, and decreased throughput. These performance bugs are difficult to detect, diagnose, and fix, and cannot be directly tackled by existing single-node debugging techniques.

Combating real-world performance bugs in distributed systems is challenging due to the poor understanding of the complexity and diversity of these bugs. Designing effective techniques to address performance bugs requires a deep understanding of how performance bugs are introduced, exposed, diagnosed and fixed. Therefore, conducting an empirical and comprehensive performance bug study is needed. There have been a few past studies [6, 28, 47] examining performance bugs, most of which focus only on single-machine systems but not on large-scale distributed systems. Moreover, only few studies consider the characteristics (e.g., root causes) of performance bugs comprehensively. For example, the study [28] conducted by Jin et al. only looks at several common root causes of performance problems and inefficient code sequences that could be fixed by simple patches. SyncPerf [6] conducts an extensive study of categories and root causes only for synchronization issues.

The following characteristics of distributed performance bugs make them difficult to dissect, categorize, and diagnose. (i) Complex and diverse bug causes. Compared to single-node bugs, performance bugs in distributed systems are caused by a much wider variety of complex reasons and lead to more different types of symptoms as software systems are getting increasingly more complex. (ii) Big impact scope. The errors and failures caused by distributed performance bugs can propagate to other threads, nodes, and even the cluster.

This paper presents our in-depth analysis of 99 representative real-word distributed performance bugs (Section 2). These bugs are sampled from five popular distributed systems (Cassandra, HBase, HDFS, Hadoop MapReduce, and ZooKeeper). We collectively organize the analysis results as over 400 classification labels and over 2,500 lines of bug re-description in the *TaxPerf* database, classified into six bug categories (and 18 bug subcategories) by their root causes, namely, resource, blocking, synchronization, optimization, configuration, and logic. TaxPerf can be used as a bug benchmark for researchers to tackle distributed performance problems and evaluate the effectiveness of performance debugging tools.

Although it is impractical to automatically detect all categories of performance bugs in TaxPerf, an important category of blocking bugs can be effectively solved by analysis tools. We analyze the cascading nature of blocking bugs and design a automatic detection tool called *PCatch* (Section 3). The blocking bugs with performance cascading influence (shortly PCbugs) are widely discovered in our studied systems. Figure 1 illustrates a real-world PCbug [8] from Hadoop MapReduce. The figure shows a TaskTracker node, where threads belonging to different jobs may execute in parallel, and a JobTracker node. In the TaskTracker node, thread t_1 downloads Hadoop Distributed File System (HDFS) files for a job. Inevitably, a client may submit a job that demands a large file, which leads to a time-consuming file-copy loop in t_1, denoted by the red polyline. Unfortunately, sometimes the slowness of this file copy could propagate through lock contentions and eventually delay the sending of TaskTracker's heart-beats in thread t_4. Such a delay of heartbeats is catastrophic. Failing to receive the heartbeat on time, the JobTracker node would consider the TaskTracker node dead, causing all tasks, which belong to various jobs, running on that TaskTracker to restart on a different node. This PCbug has been fixed by changing thread t_2 on TaskTracker, so that when t_2 fails to acquire lock l_1, it immediately releases lock l_2. PCbugs are

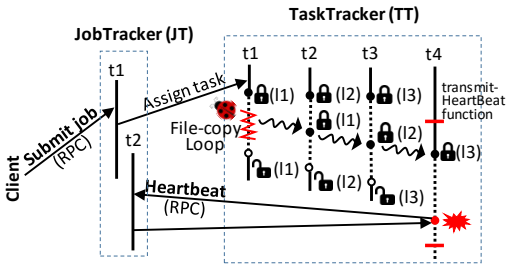

Fig. 1. A PCbug in MapReduce: the client's job delays heartbeats through lock contentions and causes the whole Tasktracker node to fail.

often triggered by large workloads and end up with global performance problems. The workload first causes non-scalable code to slow down, such as the file-download loop in Figure 1. The slowdown is inevitable and benign if well isolated. Unfortunately, software-resource contention, like the lock contention in Figure 1, may cause a local slowdown to propagate to a different job or a critical system routine, such as the heartbeat routine, and eventually lead to severe slowdowns affecting multiple jobs and sometimes multiple physical nodes in the system.

Although extensive studies have been conducted on detecting performance problems in single-node systems, such as loop inefficiency problems [42–44], cache false-sharing problems [37, 40], and lock contention problems [6, 15, 58], there are no effective solutions for detecting PCbugs in cloud systems. It is highly desirable to build automated PCbug detection tools that can *deterministically* predict PCbugs and pin-point the performance cascading chains by analyzing *small* workloads during in-house testing.

PCatch automatically detects PCbugs through a combination of static and dynamic program analyses. PCatch observes a run of a distributed system under a small-scale workload. It then automatically predicts PCbugs that can affect the performance of any specified user job or system routine (referred to as a *sink*). Specifically, PCatch identifies an $\langle L, C, S\rangle$ triplet for every PCbug: L is a code region like a loop, referred to as *source*, whose execution time could be greatly delayed under future workloads; C is a slowdown chain that could propagate the slowdown of L under certain timing; and S is the time-sensitive sink in a different job from L, whose execution time matters to users but is affected by the long delay from L propagating through Chain C. The detection is done by the three key components of PCatch, namely, performance cascading model and analysis, job identity analysis, and non-scalable source identification.

We evaluated PCatch on widely-used distributed systems (Section 4). We test small-scale workloads on these systems. By observing system execution under small-scale workloads, PCatch reports 33 PCbugs, with 22 of them being true PCbugs. Among these 22 PCbugs, 15 explain the 8 failures that we were aware of and the remaining 7 lead to failures we were unaware of. These PCbugs are difficult to catch without PCatch— even under workloads that are thousands or hundreds of thousands times larger, only 4 out of these 22 PCbugs manifest after 10 runs. The whole bug-detection incurs 3.7× ~ 5.8× slowdown, suitable for in-house use.

2 CATEGORIZED PERFORMANCE BUG ANALYSIS

2.1 Background of Bug Study

A distributed system runs on a collection of nodes interacting with each other. Each node can run multiple protocols in multiple threads. A local performance bug is a performance bug that happens locally within a node and leads to performance loss within a node. A distributed performance bug is a performance bug that

happens in distributed systems and leads to performance loss, due to the running or interaction of distributed components. We view local performance bugs as a subset of distributed performance bugs and refer to both of them as performance bugs in the rest of this paper.

To perform a comprehensive study of performance bugs for the cloud, we select five widely deployed open-source distributed systems that cover a diverse set of system architectures: Hadoop MapReduce (MR) [12] representing distributed computing frameworks, HDFS (HD) [3] representing distributed file systems, HBase (HB) [1] and Cassandra (CA) [31] representing different distributed key-value stores, and ZooKeeper (ZK) [2] representing distributed synchronization services. These are all big Java applications (144K–2.8M lines of code) with mature bug tracking systems.

For accurately identifying performance bugs, we start our study from the open-source cloud bug study (CBS) database [22], which contains over 3,000 sampled bugs reported to the JIRA bug databases of the above five cloud systems over a period of three years (1/1/2011-1/1/2014), and already labels issues related to performance bugs. From CBS, we finally randomly sample 99 performance bugs (Table 1) from over 2,500 bugs after excluding the bugs that are not real performance bugs but mistakenly labeled by CBS (e.g., zk1239) and that do not contain clear description information of root causes, symptoms and fixes, through manual inspection.

We study the characteristics of performance bugs along two key stages: root causes and implications. Our study is based on careful manual checking of the related materials of the 3,000+ bugs in CBS, including bug report description, bug discussion of developers and users, bug patch or confirmed fix strategies, and corresponding source code. Specifically, for each bug, we first ensure that the developers marked it as a real bug (not a false positive) and its bug description information is clear. We then re-describe the full step sequence of bug causality to a clearer and more concise description. To ensure the quality of the taxonomy in our study, each bug and the resultant statistics are carefully studied and examined with several rounds by more than one author who all have previous experience with these distributed systems.

2.2 Classification Overview

Performance bugs may be caused by both software and hardware faults. As hardware can fail, reliability and dependability must come from software. That is, error-handling code is a necessity to deal with hardware failure as well as software failure. Thus, in this paper, we study the root causes and other characteristics of performance bugs mainly at the software level.

Table 1 shows the classifications and counts of performance bugs in the five popular distributed systems. Performance bugs can be divided into six bug categories and 18 bug subcategories in total by their root causes. Note that, a performance bug may belong to two or more bug subcategories. We present each type of performance bugs below.

2.3 Resource Bugs

We refer to performance bugs that improperly use computing resources of distributed systems as resource-related bugs, briefly, as resource bugs. Resource bugs make up 22% of bugs studied. From our study, resource bugs can be categorized into four types: excessive resource consumption, resource leak, resource overuse and resource underuse.

Besides common computing resources such as CPU, disk, memory, I/O bandwidth and network bandwidth, resources in distributed systems also include a variety of system resources, such as nodes, file descriptors, socket connection, processes/threads, queues, heap and stack, and system-specific resources (e.g., streams). For example, hd1865 is a case overusing the number of threads, where each DFSClient has its own worker thread for checking leases, however, this is not necessary. This bug is fixed by keeping only one worker thread shared by all DFSClients, because those worker threads essentially need to be synchronized to do their work.

Table 1. Classification and statistics of categorized performance bugs for five open-source systems including Cassandra (CA), HBase (HB), HDFS (HD), MapReduce (MR), and ZooKeeper (ZK). The data in "()" denotes proportions of corresponding bug categories (or subcategories) out of total bugs (or total bugs of each category). Subscripts denote the count of real unique studied bugs.

Categories	Subcategories	CA	HB	HD	MR	ZK	Total	
Resource	Resource excessive consumption		2	2	2		6 (24%)	25 (22%)
	Resource leak	2		1		1	4 (16%)	
	Resource overuse	7		2	1	2	12 (48%)	
	Resource underuse			1	2		3 (12%)	
Blocking	Cascading blocking	2	3	2	2		9 (75%)	12 (10%)
	Heavy-load blocking			2		1	3 (25%)	
Synchronization	Improper-grained locking		1				1 (11%)	9 (8%)
	Unnecessary locking			2			2 (23%)	
	Improper locking	1		1	1		3 (33%)	
	Swollen locking			2	1		3 (33%)	
Optimization	CPU excessive consumption			1		1	2 (6%)	30 (26%)
	CPU overuse	8	2	4	1	3	18 (60%)	
	CPU underuse	3				2	5 (17%)	
	Redundant operation	1		3		1	5 (17%)	
Configuration	Numeric configuration	2	2	3	3	1	11 (92%)	12 (10%)
	Non-numeric configuration	1					1 (8%)	
Logic	General logic	5	3	2	6	3	19 (70%)	27 (24%)
	Error unawareness			4	4		8 (30%)	
Total		32_{27}	13_{11}	32_{27}	23_{20}	15_{14}	115_{99}	

2.3.1 Excessive resource consumption. Excessively consuming any type of system resources, caused by high loads (i.e., high request or operational loads), can make the type of resource insufficient. For example, for disk consumption, a variety of large history data such as old error logs, commit logs and temporary outputs can easily take up disk space if not cleaned in timely fashion. Then, the contention on remaining insufficient resources can affect part of or all of system activities and lead to performance loss.

An interesting case (hb3813) is that an RPC queue of HBase stores client requests with data waiting for being handled. Since the default queue size is big and a large number of requests are waiting, this large queue can cause an out-of-memory (OOM) failure. The bug is fixed by reducing the queue size. Overall, excessive resource consumption is usually caused under high loads.

2.3.2 Resource leak. Resource leak is a form of bad resource management like forgetting to release occupied resources. We observed that, in cloud systems, almost all of the aforementioned resources can be leaked, such as memory, file descriptor, etc. Note that, resource leak bugs are also logic bugs (Section 2.8). For example, in hd3359, memory leaks happen because DFSClient does not close all cached sockets and leads to increasing number of unavailable threads. This bug is fixed by explicitly closing cached sockets.

Note that, resource leak bugs under high load can excessively consume resources, which are similar to excessive resource consumption bugs (Section 2.3.1). However, resource leak bugs mean an exceptional state while excessive resource consumption bugs are about normal resource requirements.

2.3.3 Resource overuse. Resource overuse issues mean that the code implementation of a relevant function or algorithm consumes more resource than it should. They might not cause severe performance problems on system daily running. For example, in hd5470, HDFS uses more memory (i.e., more object allocation) than it actually needs. The bug is fixed by using a better algorithm that uses less memory.

2.3.4 Resource underuse. Insufficient resource allocation for an implementation of function or algorithm can also cause performance problems. We refer to this type of bugs as resource underuse bugs. They are opposite to resource overuse bugs (Section 2.3.3). Note that, although both resource underuse bugs and excessive resource consumption bugs (Section 2.3.1) manifest the same symptom of insufficient resources, they are different. That is, resource underuse bugs are caused by insufficient resource allocation while excessive resource consumption bugs are caused by excessive resource usage leading to insufficient resources.

For example, in hd5497, the default initial size of a queue for holding under-replicated blocks is only 16, which is small. If there are many under-replicated blocks during HDFS startup, the queue would grow and shrink with the insertion and removal of under-replicated blocks, which is very expensive. The bug is fixed by allocating large initial queue size and disabling the queue growth/shrinkage option for users.

2.4 Blocking Bugs

Blocking bugs, which make up 10% of bugs studied, are prevalent in performance bugs. One or more operations (referred to as causer operations) whose executions are supposedly short but take a significantly long time to finish, blocking other operations (referred to as affected operations). Blocking is transitive through causality relationship and software/hardware resource contention such as contention of locks, handler thread pools for queues, and shared variable. Compared to traditional only causal relationship based bugs, blocking bugs are more complicated and difficult to dissect.

Generally speaking, a blocking bug consists of three elements: source, chain, and sink. In the source-code level, the operations in source or sink could be any statement, function calls or code regions that conduct time-consuming, expensive works. The sink must be an important, time-sensitive operation such as heartbeats, regular reads or writes of systems. The chain represents the propagation path of delay or the influence way from source to sink. Note that those bugs that propagate a delay only through causality relationship and occur inside one job slowing down the job itself are considered as optimization bugs (Section 2.6).

In a blocking bug, the delay of source is usually caused by large workload processing with a big size like cluster size, data size, and load size. Specifically, in distributed systems, a long delay is usually formed by (1) a non-scalable loop whose execution time may increase with the workload changes, (2) a handling of backlog of tasks such as RPC requests or events, or (3) high-concurrency request loads or heavy operational loads. Thus, blocking bugs are often relevant to scalability problems. In addition, the delay also could be generated by downtimes of Java GC under high loads, manual delays like timeouts or sleeps, or slowdowns caused by hardware failures.

Performance blocking bugs usually have a series of cascading causes and thus are referred to as performance cascading bugs or *PCbugs*. They usually have a single time-consuming causer operation that blocks other one or more important operations through causality relationship and software/hardware resource contention. PCbugs violate the performance isolation property of distributed systems. They can be divided into local and global cascading bugs based on intra-job or inter-job problems. A job means a user job or a system routine that can be identified in job-origin analysis by our performance blocking bug detection tool (PCatch), which will be introduced in the next section.

Local PCbugs mean that a time-consuming operation blocks other operations inside the same job. Although the causer and affected operations of the local PCbugs could be in different threads, they all belong to the same job. That is, this type of bugs are intra-job problems from the perspective of bug impact scope.

Global PCbugs propagate a local delay from one job to other jobs, which can affect multiple jobs and even multiple physical nodes in the system. For example, in ca6744, a slow clean-up loop in one job could slow down another job on a different Daemon node due to race on an event-handling thread. That is, an event queue is handled by a single handler thread that contains two event handlers from two different jobs respectively. The slow loop in one event handler blocking the other one affects the other job. This bug is fixed by increasing the number of handler threads.

A handling of backlog of tasks such as RPC requests or events also can lead to cascading blocking bugs like mr4749 and mr4088, and may lead to heavy-load blocking bugs below.

Heavy-load blocking. A subset of blocking bugs are caused by large numbers of operations from high-concurrency request loads or heavy operational loads that block other important operations. They are referred to as *heavy-load blocking bugs* which are essentially caused by the contention of resources. They are similar to excessive resource consumption bugs (Section 2.3.1) which excessively consume some resources listed in Section 2.3 and lead to contention of the insufficient remaining resources. However, we categorize them as blocking bugs, because they mainly focus on obvious blocking problems caused by contention of target systems' application resources like synchronization resources. In contrast, excessive resource consumption bugs focus on resources of host operating systems. For example, in hd5790, when a huge humber of leases need recovery, NameNode may receive tens of thousands of blockcommit request calls from DataNodes. Each request takes a long time to handle because the problematic algorithm requires holding NameNode's lock, which affects and slows down all regular operations in NameNode.

2.5 Synchronization Bugs

Synchronization bugs or lock-related problems, which make up 8% of bugs studied, are common in distributed systems as well as single-machine multiple threaded software. Also, distributed systems are more prone to synchronization bugs when writing synchronization related codes due to their complexity. Note that synchronization bugs are related to but different from blocking bugs (Section 2.4), in that blocking bugs have the cascading property and time-sensitive sinks.

2.5.1 Improper-grained locking. The granularity is an important property of a lock. It represents a measure of the amount of data a lock is protecting. Generally, using a coarse granularity (a small number of locks, each protecting a large segment of data) results in less lock overhead when a single process is accessing the protected data, but more lock contention (i.e., worse performance) when multiple processes are running concurrently. Conversely, using a fine granularity (a larger number of locks, each protecting a fairly small amount of data) increases the overhead of the locks themselves but reduces lock contention.

From our study, coarse-grained locking bugs are more common than fine-grained locking bugs in distributed systems. A coarse-grained locking bug means a lock (i.e., a big lock) used to protect the corresponding operations is at a higher level than it should be. Thus, the position of a coarse-grained lock needs to be changed.

For example, in hb3483, when a RegionServer hits its global memstore upper limit, all writes on it have to wait until memory has flushed down to the low water mark since this flushing process holds a relevant lock. This bug has been fixed by triggering flushes in advance in a daemon thread whenever memory size is above the low water mark.

2.5.2 Unnecessary locking. Unnecessary locking bugs can be viewed as over-synchronization problems already discussed in some previous studies [6, 28]. Over-synchronization indicates a situation where a synchronization becomes unnecessary because the corresponding computations do not require any protection or they are already protected by some other synchronizations.

For example, in hd4702, the function of DatanodeManager::fetchDatanodes holds the namesystem read lock while iterating through data nodes which contains operations requiring mutual exclusion. However, the operations are already synchronized in another internal function of fetchDatanodes. The bug is fixed by simply dropping the lock.

2.5.3 Improper locking. Developers can use a variety of synchronization primitives (e.g., atomic instructions, spin locks, try-locks, read/write locks, mutex locks etc.) to protect shared accesses. Different synchronization primitives have different use cases.

From our study, synchronization mechanisms such as synchronized methods, read/write locks, and custom synchronizations are more common and are prone to improper locking bugs. Also, improper locking bugs usually are seen in domain-specific scenarios.

For example, in hd5064, NameNode uses a fair read-write lock to handle read/write requests waiting in a queue. When handling a long read while a write is waiting at the head of queue, later concurrent short reads in the queue cannot go together with the long read, because their priority is lower than the concurrent write. That is, the concurrent write is waiting for the long read to finish; and those short reads are waiting for the write to finish. The bug is fixed by optimizing the read-write lock.

2.5.4 Swollen locking. Swollen locking mean the enclosed critical section of a lock involves large numbers of operations unnecessary to be synchronized. We differentiate swollen locking bugs from improper-grained locking bugs because of their prevalence in performance bugs. Unlike previous coarse-grained locking bugs (Section 2.5.1), the root causes of swollen locking bugs are in those operations unnecessary to be synchronized while the root causes of coarse-grained locking bugs are the locks themselves whose positions needed to be changed. In addition, swollen locking bugs are also different from redundant operation bugs (Section 2.6.4) whose contained redundant operations are really unnecessary.

2.6 Optimization Bugs

Optimization means making a program execute more rapidly, that is, getting shorter completion time or using less CPU resource (i.e., execute less number of CPU instructions). An optimization bug is a performance bug whose execution speed of corresponding buggy code regions needs to be improved, i.e., reducing the execution completion time of the buggy codes. Note that optimization bugs are specific to CPU-related optimizations.

Optimization bugs make up 26% of bugs studied. From the perspective of performance impact propagation, optimization bugs are also slowness propagation issues like blocking bugs (Section 2.4). However, they are performance bugs whose slowness propagation are through causality relationship and from blocking bugs. Optimization bugs can be categorized into the following types.

2.6.1 Excessive CPU consumption. Excessively consuming CPU resource caused by high loads can cause contention on remaining insufficient CPU resource and affect part of or all of system activities. That can lead to performance loss just like excessive resource consumption bugs. For example, in hd4992 mentioned below, the load balancing conducted by large numbers of worker threads in HDFS creates significant load on NameNode. This excessively consumes CPU resource. The bug is fixed by making the balancer thread numbers configurable, that is, uses can limit the number of threads manually.

2.6.2 CPU overuse. CPU overuse bugs are common optimization bugs whose code implementation of a relevant function or algorithm takes more time and consumes a little more CPU resource than it should be. They might not cause severe performance problems on daily running of systems.

For example, in ca5389, Cassandra deserializes the data to trees which consumes CPU cycles. However, this tree feature is useless, because on the write path the tree is not needed, and on the read path the deserialized data

will be cloned and merged to the new result. The bug is fixed by simply deserializing the data to list, now that it is only the intermediate product.

2.6.3 CPU underuse. CPU underuse bugs are a type of optimization bugs whose code implementation of a relevant function or algorithm consumes less CPU resource than it should be. They are opposite to CPU overuse bugs (Section 2.6.3).

For example, in mr4720, the web UI page of Job History Server, using a bad algorithm, takes long time to download and render with 20,000 jobs. The bug is fixed by using deferred rendering algorithm for speeding.

2.6.4 Redundant operation. Redundant operation (or unnecessary operation) bugs are bugs contain code units that conduct unnecessary work and generate unused results or do not always generate useful results, given the calling context. This is similar to "skippable functions" [28]. Note that, redundant operation bugs, in fact, are a special case of CPU overuse bugs and considered as an independent category in our study.

For example, in zk1642, the leader server needs to load the database for figuring out the zxid before running leader election. However, it loads the database again when it starts leading. This is problematic for larger databases. The bug is fixed by skipping the database load process if it is loaded before and using the zxid already calculated.

2.7 Configuration Bugs

Configuration bugs [10, 24], which make up 10% of bugs studied, have become significant deployment problems in cloud systems. They refer to unsuitable configuration parameter settings that negatively impact the performance. Note that, those bugs with logic problem inside code, which can be exposed by certain configuration values and cause hang or others that indirectly leads to performance loss, are considered as logic bugs (Section 2.8) rather than configuration bugs.

2.7.1 Numeric configuration. Numeric configuration bugs [50] are dominant (> 90%) in configuration bugs. The corresponding configuration parameter values are integers or floating-point. Numeric configuration bugs, which are performance sensitive to different configuration parameter values, affect the performance by wrong/unreasonable parameter value settings.

For example, in hb2149, the default value of hbase.regionserver.global.memstore.lowerLimit of 25% is too low. When the global memstore size reaches the upper limit (40%), the memory flushing starts and all write operations are blocked during the flushing. The bug is fixed by improving the lower limit value from 25% to 35%.

2.7.2 Non-numeric configuration. Generally, non-numeric configuration refers to (1) the configurations that are binary and determine whether a performance optimization is enabled, (2) the configurations that contain several configuration items for users to choose a suitable data structure or algorithm for a specific scenario (e.g., runtime environment).

2.8 Logic Bugs

Logic bugs, which make up 24% of bugs studied, can occur in any code and involve a wide variety of buggy codes, including incorrect or incomplete data structure, algorithm, error handling, failure recovery, etc. Logic bugs in performance bugs are a special bug category in which they are kind of side-effects of functional bugs. We also call logic bugs of performance bugs as pseudo-functional bugs, because they can manifest symptoms like functional bugs, such as crashes or hangs, but finally cause performance loss rather than functional misbehavior (i.e., incorrect software functionality) of distributed systems.

Note that, those bugs with logic problem inside code, which can be exposed by certain configuration values and cause hang or others that indirectly leads to performance loss, are considered as logic bugs instead of configuration bugs (Section 2.7). In addition, if a performance bug is a logic bug, we will not tag it with other bug

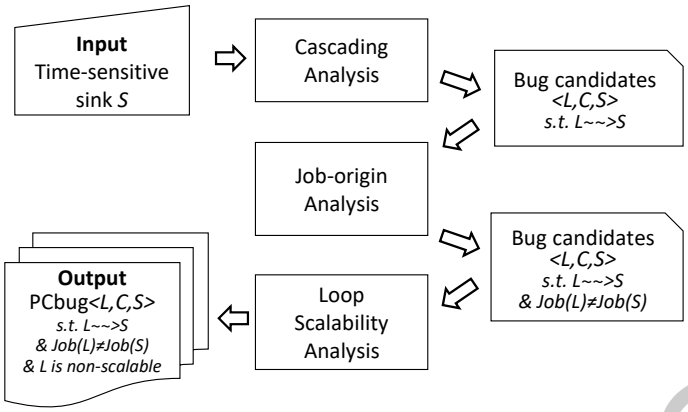

Fig. 2. PCatch overview

categories, except resource leak bugs (Section 2.3.2) that are caused by wrong logics of forgetting to normally release occupied resources.

2.8.1 General logic. General logic bugs refer to performance bugs involving incorrect logics in code implementation. They are related to a wide variety of bugs, covering all above four main bug categories (e.g., resource, blocking, synchronization, optimization). That is, general logic bugs may affect system resources such as the size of memory, the I/O bandwidth of disk, code execution time, etc.

For example, in mr3862, When a NodeManager attempts to shutdown itself cleanly, it is possible for it to appear to hang due to lingering DeletionService threads which are not correctly shut down. The bug is fixed by correctly shutting down DeletionService threads first. This bug is caused by wrong logic and leads to local delay propagation through causality relationship inside a job itself. This is related to optimization bugs (Section 2.6).

As shown in Section 2.3.2, ca5175 is a resource leak bug as well as a general logic bug caused by wrong logic. In addition, error-handling code dealing with hardware/software errors/failures always contains general logic bugs.

2.8.2 Error unawareness. Error unawareness bugs are a special bug category in logic bugs. They are unaware of (or ignore) systems' errors such as invalid values (e.g., mr4919), insufficient permission (e.g., hd4485), bad resources such as bad nodes (e.g., mr4599, hd3703), etc., and then continue to execute. This type of bugs are often caused by missing error detection, error handling, failure detection, failure recovery, etc.

For example, in hd4485, the MapReduce (MR) job does not run as the same user as HDFS when reading data from HDFS. This makes MR clients not able to use short-circuit read to directly access local DataNode (DN) data for reducing read/write time because of a "permission denied" error when connecting to the socket of local DN. The bug is fixed by setting r/w permission via chmod(666) automatically after DN creates the socket.

3 PCATCH: DETECTING CASCADING BUGS

Although it is impractical to automatically detect all categories of performance bugs in TaxPerf, fortunately we find that an important category of blocking bugs can be automatically detected by leveraging there cascading nature to design an efficient detection tool called *PCatch*.

3.1 Causality relationship

An important property of blocking bugs is the cascading impacts which can be modeled as the traditional causality relationship [32, 36, 38, 41]: an operation o_1 *happens before* another operation o_2 if there exists a logical relationship between them so that o_2 cannot execute unless o_1 has finished execution. We also refer to this relationship as o_2 *causally depends* on o_1, denoted as $o_1 \rightarrow o_2$. We refer to o_1 as the *causor* of o_2, denoted as $Causor(o_2)$. Causal relationship is transitive: if $o_1 \rightarrow o_2$ and $o_2 \rightarrow o_3$, then $o_1 \rightarrow o_3$.

PCatch directly uses the causal relationship model and some analysis techniques presented by DCatch [36]. PCatch traces causal operations and uses them to construct *happens-before (HB) graphs*. PCatch uses this graph to decide whether two operations are concurrent and what are the causors of a given operation.

Figure 2 shows the workflow of PCatch which can automatically detect a PCbug $\langle L, C, S \rangle$. The inputs to PCatch bug detection include the distributed system D to be checked, a small-scale workload that allows PCatch to observe a run of D, and code regions in D whose execution time matters to users (i.e., sinks). Developers can use PCatch APIs `_sink_start` and `_sink_end` to specify any code region as a sink.

The first step of PCatch is dynamic cascading analysis. At this step, PCatch analyzes run-time traces to identify a set of potential PCbug sources whose execution time could propagate to affect the duration of the sink through software-resource contention and (optionally) causality relationships. Here, we only consider loops as potential PCbug source candidates, as loops are most likely to become performance bottlenecks, as shown by previous empirical studies [28]. We analyze potential causality and software-resource contentions based on our cascading model, which will be presented in Section 3.2. At the end of this step, we get a set of PCbug candidates $\{\langle L, C, S \rangle\}$.

The second step of PCatch conducts dynamic job-origin analysis for every PCbug candidate. By analyzing the run-time trace, PCatch automatically judges whether the source L and the sink S belong to the same user/system job. Only those that belong to different jobs remain as PCbug candidates. The others are pruned, as they are at most local performance problems. This step is explained in Section 3.5.

The last step of PCatch conducts dynamic-static hybrid loop scalability analysis to check whether the source in a PCbug candidate has the potential to be time consuming under a future workload. We look for loops that satisfy two conditions: (1) each loop iteration takes time; and (2) the total number of loop iterations does not have a constant bound and can potentially increase with the workload. For the first condition, PCatch simply looks for loops whose loop body contains I/O operations. For the second condition, PCatch designs an algorithm that leverages loop-bound analysis and data-flow analysis, with the details in Section 3.6. After this step, all the remaining candidates are reported as PCbugs.

3.2 Cascading analysis

3.2.1 Goals. We consider two code regions to have a *performance dependency* on each other, if the slowdown of one region can lead to the slowdown of the other. Such a dependency can be established through two causes. (1) **Causality relationship** can force an operation to *deterministically* execute after another operation (e.g., message receiving after sending) or one region to *deterministically* contribute to the duration of the other (e.g., RPC execution time contributes to the RPC caller's time). (2) **Resource contention**[1] can cause a resource acquisition operation to *non-deterministically* wait for another party's resource release. Consequently, a delayed release will cause an extended waiting.

The PCatch cascading analysis aims to identify $\langle L, C, S \rangle$, where delays in code region L can propagate to slow down sink S through a chain C which contains resource contention and (optionally) causality relationships.

3.2.2 Challenges. There are three key challenges here.

[1]Hardware resource contention can also cause lack of performance isolation [62], but is out of the scope of PCbugs.

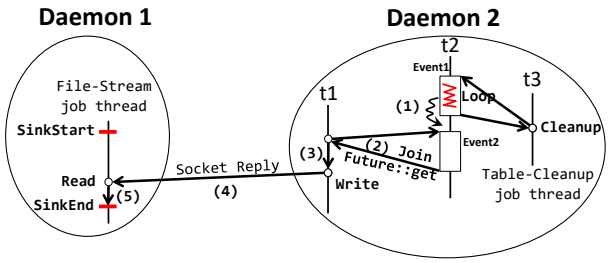

Fig. 3. A real PCbug in Cassandra, where a slow clean-up loop in one job could slow down another job on a different node due to competition on the event-handling thread t2. $\rightarrow$ represents must-HB and $\rightsquigarrow$ represents may-HB.

- Non-determinism. Resource contention is non-deterministic, as resource-acquisition order may vary from one run to another. PCatch needs to predict potential dependencies.
- Diversity. Distributed systems have a variety of communication and synchronization mechanisms (synchronous or asynchronous, intra- or inter-node) that can cause performance dependencies, such as locks, RPCs, events, and messages shown in Figure 1 and 3. Missing any of them would hurt coverage and accuracy of PCbug detection.
- Composition. Performance dependency between L and S may include contentions of multiple resources, such as the three locks in Figure 1, as well as multiple causality chains, as shown in Figure 3. PCatch needs to carefully handle not only individual causality/contention operations, but also their compositions.

To systematically and accurately reason about and predict execution delays caused by such complicated and entangled resource contentions and semantics dependencies, we will first build a performance-dependency model, and then design a dependency analysis algorithm based on that model.

3.3 Cascading model

We will first discuss causality relationships and resource-contention relationships separately below in Section 3.3.1 and Section 3.3.2, followed by a discussion in Section 3.3.3 about how to compose them together to reason about performance cascading relationships in distributed systems.

3.3.1 Causality (must-HB) relationships. Under causal relationship $o_1 \rightarrow o_2$, the delay of o_1 would delay the start of o_2. For example, the delay of message sending would always delay the start of the message handling on the message-receiving node.

3.3.2 Resource contention (may-HB) relationships. Under resource contention, one resource-acquisition operation could non-deterministically wait for another resource-release operation. For example, an attempt to acquire lock l has to wait for whoever *happens to* acquire l earlier to release l. This non-deterministic relationship is a crucial ingredient of performance cascading in the context of PCbugs — the slowdown of any operation o while holding a resource l would lead to an extended wait at a corresponding resource acquisition a, denoted as $o \rightsquigarrow a$.

PCatch models two key types of software-resource contention: locks and thread pools. They play crucial roles in coordinating execution, which inevitably brings contention, in synchronous thread parallel execution, asynchronous event-driven concurrent execution, synchronous RPC processing, and asynchronous socket-message handling.

Locks. For two critical sections that are concurrent with each other and protected by the same lock, the delay inside one could postpone the start of the other, and vice versa. Consider Figure 4(*left*). For two pairs of lock-unlock operations, {$\texttt{lock}_1\texttt{(l)}$, $\texttt{unlock}_1\texttt{(l)}$} and {$\texttt{lock}_2\texttt{(l)}$, $\texttt{unlock}_2\texttt{(l)}$}, if $\texttt{lock}_1\texttt{(l)}$ // $\texttt{lock}_2\texttt{(l)}$, we

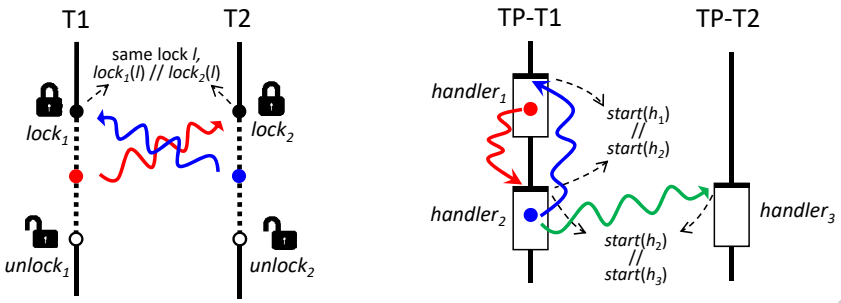

Fig. 4. May-HB relationship.

can infer that $o_2 \rightsquigarrow \texttt{lock}_1\texttt{(l)}$ and $o_1 \rightsquigarrow \texttt{lock}_2\texttt{(l)}$, where o_1 and o_2 are any operations inside the {$\texttt{lock}_1\texttt{(l)}$, $\texttt{unlock}_1\texttt{(l)}$} and {$\texttt{lock}_2\texttt{(l)}$, $\texttt{unlock}_2\texttt{(l)}$} critical sections, respectively, which consequently result in May-HB relationship. Note that, such a delay could directly affect another node in a distributed system when the critical section is inside a message/RPC handler.

In practice, there are different variations of this rule, depending on different lock primitives. For example, given a reader-writer lock, the may-HB relationship has to involve a writer-lock critical section, and does not exist between two reader-lock critical sections.

Thread-Pool. Distributed systems often maintain a fixed number of threads that are responsible for executing handlers of events inserted into a specific queue or RPCs delivered to a specific node. In these cases, the handler threads become a target of resource contention. If one event (RPC) happens to be inserted into the handling queue (or dispatched to the thread pool) later than another event (RPC), the start of its handler function inside the thread (e.g., t_2 in the Daemon-2 node in Figure 3) could be unexpectedly delayed by the execution of the other event/RPC handler. Consequently, we have the following may-HB rule related to thread pools. For two event/RPC handler functions h_1 and h_2 that are processed by the same thread pool, if the start of function h_1 is concurrent with the start of function h_2, denoted as $\texttt{start}_{h_1} \,//\, \texttt{start}_{h_2}$, we can infer that $o_1 \rightsquigarrow \texttt{start}_{h_2}$ and $o_2 \rightsquigarrow \texttt{start}_{h_1}$, where o_1 and o_2 are any operations inside handler functions h_1 and h_2, respectively. Figure 4(*right*) shows an example of may-HB relationship where there are three handlers (h_1, h_2, h_3) inside two handler threads of the same thread pool. If the starts of h_1 and h_2 are concurrent, then operations inside h_1 may-happen-before the start of h_2. Similarly, if the starts of h_2 and h_3 are concurrent, then operations inside h_2 may-happen-before the start of h_3. Both result in may-HB relationship.

The may-HB relationship is transitive — if $A \rightsquigarrow B \rightsquigarrow C$, we know that $A \rightsquigarrow C$ as long as $A//C$.[2] That is, a slowdown of executing A would indirectly cause a slowdown of executing C. For example, in Figure 1, a slowdown of the file-copy loop in thread t_1 of the TaskTracker node could cause extra wait time in thread t_4's lock acquisition through three pairs of lock contention: file-copy loop $\rightsquigarrow \texttt{lock}(l_1)$ in t_2 $\rightsquigarrow \texttt{lock}(l_2)$ in t_3 $\rightsquigarrow \texttt{lock}(l_3)$ in t_4.

Clearly, resource contention caused by thread pools is the most intense when the thread pool contains only one thread, and is much less intense when multiple threads are available. Our modeling does not consider the number of threads in the pool, because many thread pools in real-world systems leave the number of threads configurable — the number of threads could become one even if it is currently not. Furthermore, even when there are multiple threads in the pool, the slow processing of one handler function could still delay the start of later handler functions due to decreased handling throughput.

[2] $A//B$ and $B//C$ does not guarantee $A//C$.

Input: A sink region S
Output: A set of loops $\mathbb{L} = \{L | L \rightsquigarrow S\}$
Set<Loop> $\mathbb{L} \leftarrow$ Null;
Queue<Pair> $\mathbb{R} \leftarrow \{S\}$;
while $R = \mathbb{R}.pop()$ **do**
  for L *in* R **do**
    if $L \,//\, S_{end}$ **then**
      $\mathbb{L}.add(L)$
  end
  $\mathbb{R}.push(mustHB(R))$;
  $\mathbb{R}.push(mayHB(R))$;
end
return $\mathbb{L}$;

Algorithm 1: Cascading analysis algorithm

3.3.3 Composing must-HB and may-HB relationships. Now, we can reason about the performance dependency between any two code regions R_1 and R_2 considering both must-HB and may-HB relationships.

May-HB relationships directly translate to performance dependencies. That is, if we find an operation o_1 inside R_1 and an operation o_2 in R_2 so that $o_1 \rightsquigarrow o_2$, we know that slowdowns in R_1 could lead to slowdowns in R_2.

The must-HB relationship does *not* always mean performance dependencies. For example, the thread-creation operation in a parent thread happens before every operation inside the child thread. However, delays in the parent thread can**not** lead to extra execution or waiting time inside the child thread through this must-HB relationship.

The must-HB relationship leads to performance dependencies in the following situation. Suppose the last operation of R_1 is o_{end1} and the first operation of R_2 is o_{start2}. If we can find an operation o_2 in R_2 so that $o_{end1} \rightarrow o_2$ and $o_{end1} \not\rightarrow o_{start2}$, then we know that slowdowns in R_1 would delay the execution of o_{end1}, which would delay o_2 but not o_{start2} and hence extend the execution time of R_2. For example, in Figure 3, since the message `Write` in t_1 of Daemon-2 is concurrent with `SinkStart` on Daemon-1 and happens-before the message `Read` on Daemon-1, slowdowns in t_1 on Daemon-2 before the `Write` could cause slowdowns in the sink on Daemon-1.

We can now compose may-HB and must-HB relationships together, as performance dependencies among code regions are clearly transitive: if slowdowns in region R_1 could lead to slowdowns in region R_2 and slowdowns in R_2 could lead to slowdowns in R_3, naturally R_1 could slow down R_3. For example, in Figure 3, we can infer that the slowness of the cleanup loop in Daemon-2 could unexpectedly propagate to delay the ending of the file-stream job in Daemon-1 through the following propagation chains: (1) the cleanup loop inside Event_1 could slow down the execution of Event_2, as cleanup-loop $\rightsquigarrow$ start of Event_2, (2) Event_2 could slow down thread t_1's execution before `Write`, as the end of $\text{Event}_2 \rightarrow$ the event-join on t_1 and $\not\rightarrow$ the start of t_1, (3) t_1's execution before `Write` could slow down the sink, as `Write` $\rightarrow$ `Read` and $\not\rightarrow$ `SinkStart` on Daemon-1.

3.4 Analysis algorithm

PCatch analyzes run-time traces to identify every loop L in the trace upon which sink S has contention-related performance dependencies, denoted as $L \rightsquigarrow S$. Following the performance-dependency model discussed above, for every L, PCatch needs to identify a sequence of code regions $R_1, R_2, ..., R_k$, where R_1 has performance dependencies on L, R_{i+1} depends on R_i, and finally S on R_k, with at least one of such dependencies based on resource contention.

In this section, we assume that the trace contains all the information needed by PCatch's analysis. We also assume any loop could execute for a long time under some workload, and we will discuss how to prune out those loops whose execution time is guaranteed not to increase with any workload in Section 3.6.

Algorithm. The outline of our algorithm is shown in Algorithm 1. At a high level, we maintain a working region set $\mathbb{R}$. The sink S has a performance dependence on every member region R in this working set. We process one region R in $\mathbb{R}$ at a time, until $\mathbb{R}$ becomes empty. For every R, we check if it contains a loop L that is concurrent with the end of the sink (S_{end}). If so, we conclude that $L \rightsquigarrow S$. We then add to the working set regions that R has a performance dependence upon based on must-HB analysis (i.e., the `mustHB(R)` function in Algorithm 1) and may-HB analysis (i.e., the `mayHB(R)` function in Algorithm 1).

The `mustHB` function works as follows. Given an input region R, PCatch iterates through every node in the HB graph that corresponds to an operation in R. For every node v, PCatch checks if there is an edge on the HB graph that reaches v from a node v' that belongs to a different thread or handler. Once such a v' is identified, its corresponding region R' is added to the output of `mustHB(R)` — R' ends at v' and starts at p so that p and v' belongs to the same thread/handler and $p \not\rightarrow R_{start}$. Tracing backward along the must-HB graph could produce many code regions. In general, the farther away the less likely is performance impact, because any part of the chain may not be on the critical path. Consequently, we set a threshold to limit the number of cross-handler/thread/node must-HB edges in the traversal. We currently use a threshold of 2.

The `mayHB` function works as follows. Given an input region R, the analysis goes through every operation in R. Whenever a lock acquisition or an event/RPC handler start o is encountered, we will identify from the trace all the lock critical sections or handler functions that compete on the same lock or the same handler thread with o and are concurrent with o, and put them all into the output set.

For example, in Figure 3, our analysis starts from the sink in the file-stream job thread. The `mustHB` analysis will discover the code region in thread t_1 on Daemon-2 that ends with the socket write, and then discover the Event_2 handler in thread t_2. While applying `mayHB` analysis to the latter, we discover the Event_1 handler also in thread t_2. Finally, analyzing the Event_1 handler will reveal the cleanup loop that has a cascading relationship with the sink.

3.5 Job-origin Analysis

PCatch analyzes and prunes out PCbug candidates whose source and sink belong to the same job, as these do not reflect global performance-cascading problems.

To figure out which job an instruction belongs to, PCatch identifies the corresponding node of this instruction in the HB graph and searches backward along causality edges on the graph. For example, in Figure 1, the file-copy loop is inside an RPC function in TaskTracker thread t_1; its caller is inside another RPC function in JobTracker's thread t_1; its caller is from the client, which ends the back tracing. At the end of the back tracing, instructions that belong to different jobs will end up with different nodes in the HB graph.

PCatch distinguishes the origins between user jobs and system routines. For an instruction that belongs to a user job, such as the loop in Figure 1, the origin tracing usually would end up at an RPC function or a message handler that was invoked by client, such as the `submitJob` RPC call in Figure 1. On the other hand, for an instruction that belongs to a system routine, the origin tracing usually ends up at the main thread of a process that was started during system start-up (i.e., not by interaction with clients). For example, in Figure 1, when we search for the origin of the heartbeat function in thread t_4 of the TaskTracker node, we find that it is not inside any event/RPC/message handler and is inside the main thread of the TaskTracker process.

If the origin is a main thread, PCatch uses the process ID paired with the thread ID as the job ID. If the origin is an RPC function or a message handler, PCatch uses the handler's process ID paired with the RPC/message ID as the job ID. Then, we can easily tell whether two (groups of) instructions belong to the same job.

```
1 vector[2] = 1;
2 for (i=0; i < vector.size(); i++) {
3     ...
4 }
```

Fig. 5. Data-dependence may not affect loop bound (Line 1 affects the content but not the size of `vector`, which defines the loop count).

3.6 Loop Scalability Analysis

3.6.1 Goals. As discussed in Section 3.1, we aim to identify loops satisfying the following two conditions as potential slow-down sources of PCbugs: (1) each iteration of the loop is time-consuming, conducting I/O operations, such as file system accesses, network operations, explicit sleeps, etc; (2) the number of loop iterations does not have a constant upper bound and can potentially increase together with the workload. We refer to such loops as *non-scalable*.

Identifying loops that satisfy the first condition (time-consuming iterations) can be conducted by watching specific API calls which will be introduced in Section 4.1. Identifying loops that satisfy the second condition (non-scalable loops) is more challenging and is the focus of this section.

3.6.2 Challenges. Identifying non-scalable loops is related to but different from loop complexity analysis [20, 21] and program slicing [52], and thus demands new analysis algorithms.

Traditional loop-complexity analysis tries to figure out the complexity of a loop L in a program variable V, which is both unnecessary and insufficient for PCbug detection. It is unnecessary because we only need to know whether the loop count could scale up with the workload, but not the exact scaling relationship — whether it is $O(V^2)$ or $O(V^3)$ does not matter. It is also insufficient because it does not tell whether the value of V can increase with the workload or not.

Traditional analysis for program slicing tries to figure out what affects the content of a variable V at a program location P. Clearly, it alone is insufficient to identify non-scalable loops, because we first need to identify which variable V can approximate the loop count (e.g., `vector` in Figure 5). Furthermore, even if V has data dependence upon an operation O, O may not be able to affect the upper bound of V, as illustrated by Figure 5. Meanwhile, even if V has no data dependence upon an operation O, O may actually affect the upper bound of V, as we will explain later in Section 3.6.5 and Figure 6 (in Figure 6, the size of `toAdd` approximates the loop count at line 21; `toAdd` has no data dependence on `report.NumBlocks()` on line 17, but its size is decided by the latter).

3.6.3 Our Solution. Since accurately computing the loop count and identifying all non-scalable loops is impractical [20, 21], we give up on soundness and completeness, and instead aim to identify some common types of non-scalable loops with good efficiency and accuracy.

To choose those common types, we focus on the two important aspects of every loop exit condition — the loop index (e.g., `i` in Figure 5) and the loop index bound (e.g., `vector.size()` in Figure 5), which clearly have a large impact on when a loop exits and hence on the loop count. Thinking about what types of loop index variables and loop index-bound variables might lead to non-scalable loops produces the following three patterns:

(1) Loops with a loop-invariant index. The loop cannot exit until another thread updates a shared variable with a loop-terminating value. The durations of these synchronization loops are non-deterministic and could be affected by workloads (e.g., Figure 7).
(2) Loops with a workload-sensitive index. Return values of I/O operations define the loop index variable and hence determine the loop count (e.g., Figure 8).

```
1  File[] blockFiles = dir.listFiles();  // Java I/O API
2  for (i=0; i < blockFiles.length; i++) {
3    ...
4    blockSet.add(new Block(..)); //augmentation
5  }
6  ...
7  BlockListAsLongs(blockSet.toArray(alist));
8  ...
9  BlockListAsLongs(long[] list) {
10     blockList = list ? list : new long [0];
11 }
12 ...
13 NumBlocks() {
14     return blockList.length/LONGS_PER_BLOCK;
15 }
16 ...
17 for (i=0; i < report.NumBlocks(); i++) {
18     toAdd.add(block);           //augmentation
19 }
20 ...
21 for (Block block: toAdd) {
22     ...
23 }              //toAdd has type Collection<Block>
```

Fig. 6. A simplified loop example from HDFS

```
1  while (rjob.localing) {
2      rjob.wait();
3  }
```

Fig. 7. Synchronization loop in MapReduce (`rjob.localing` is a loop invariant if not consider other threads)

```
1  while (bytesRead >= 0) {
2      ...
3      bytesRead = in.read(buf);
4  }
```

Fig. 8. MapReduce Loop with workload-sensitive index

(3) Loops with a workload-sensitive bound. Return values of I/O operations contribute to the loop bound and hence affect the loop count (e.g., in Figure 6, the return value of an I/O operation at line 1 affects the loop bound at line 21).

Our analysis goes through three steps. The first step (Section 3.6.4) conducts static analysis inside a loop body to identify non-scalable loops of type 1 and type 2. It also identifies a variable V that can approximate the bound of the loop count. The second step conducts global static analysis to identify I/O operations O that may contribute to the value of V (Section 3.6.5). The last step runs the system again to see whether operations O can indeed be executed and hence finishes identifying non-scalable loops of type 3. The last step is straightforward and hence is skipped below.

3.6.4 Loop-local analysis. Given a loop L, we first identify all the exit conditions of L. That is, PCatch uses WALA the Java byte code analysis infrastructure [26] to identify all the loop exit instructions and the branch conditions

```
1 for (int idx = 0; idx < volumes.length; idx++) {
2     volumes[idx].getBlockInfo(blockSet);
3 }
```

Fig. 9. A loop with constant stride in HDFS

```
1 for (Block block: toAdd) {
2     addStoredBlock(block, ...);
3     ...
4 }
```

Fig. 10. Collection-related loop in HDFS

that predicate these exit instructions' execution. In WALA, every condition predicate follows the format of $AopB$, where A and B are numerical or boolean typed, and op is a comparison operator. In the following, we first present a baseline algorithm, and then extend it to handle more complicated loops.

Baseline algorithm. Suppose we identify a loop-exit condition $V_{lower} < V_{upper}$ or $V_{lower} \leq V_{upper}$. We will then analyze how V_{lower} and V_{upper} are updated inside the loop.

If both are loop invariants, we consider this loop as a synchronization loop and hence type-1 non-scalable loop. For example, in the loop shown in Figure 7, the exit condition is `rjob.localing == FALSE`. Since `rjob.localing` is a loop invariant variable and `FALSE` is a constant, this loop has to rely on other threads to terminate it.

If neither of them is a loop invariant, we give up on analyzing the loop and simply consider it as scalable — this design decision may introduce false negatives, but will greatly help the efficiency and accuracy of PCatch.

If only one of them is a loop invariant, we check how the other one $V_{variant}$ is updated in the loop.

If $V_{variant}$ is updated with a constant increment or decrement in every iteration of the loop, such as `idx` in Figure 9, we consider this loop as a possibly type-3 non-scalable loop and consider the value of V_{upper} at the start of the loop, such as `volumnes.length` in Figure 9, as an approximation for the loop-count's upper bound. The global analysis in Section 3.6.5 will then analyze how the value of V_{upper} is computed before the loop to decide whether loop L is scalable.

If $V_{variant}$ is not updated with a constant stride, we then check whether it is updated with content returned by an I/O function in the loop, such as `in.read(buf)` shown in Figure 8. If so, we identify a type-2 non-scalable loop. Otherwise, we stop further analysis and exclude the loop from PCbug-source consideration.

Handle collections. Many loops iterate through data collections like the one shown in Figure 10. In WALA, exit conditions of this type of loops contain either `hasNext()` (i.e., `Iterator::hasNext() == FALSE`) or `size()` (e.g., `i≤list.size()`). We then adapt the baseline algorithm to analyze these container-related loops.

We consider Java `Collection` APIs like `next()`, `remove (obj)`, and `add(obj)` as a constant-stride decrement or increment to the iterator, and consider APIs like `addAll(...)` and `removeAll(...)` as non-constant update. If the loop contains no such update operation, it is considered as a synchronization loop (i.e., non-scalable); if it contains only constant-stride increment or only constant-stride decrement in every loop iteration, the loop is considered as potentially non-scalable and the size of the corresponding collection, $C.size()$ will be identified for further analysis later.

Handle equality conditions. When the loop-exit condition is $V_1 \neq V_2$ or $V_1 == V_2$, our analysis above that identifies type-1 and type-2 loops still applies. If the loop exits when $V_1 == V_2$, type-3 loop analysis is conducted similarly as above. That is, if one of V_1 and V_2 is a loop invariant and one has a constant stride inside the loop, we will move on to analyze if the initial value of either V_1 or V_2 at the beginning of the loop is affected by I/O

```
Input: A loop L and its loop-count approximation V
Output: A set of I/O operations 𝕆 = {O|O contibutes to V}
Set<Variable> ℂ ← {V};
Set<I/O Operation> 𝕆 ← Null;
while c = ℂ.pop() do
    if exp_c == constant then
        continue
    if exp_c == I/O function then
        𝕆.add(exp_c)
        continue
    for x in ContributorOf(exp_c) do
        ℂ.add(x)
    end
end
return 𝕆;
```

Algorithm 2: Global loop count analysis algorithm

operations. If the loop exits when $V_1 \neq V_2$, we give up type-3 loop analysis, as its loop bound is too difficult to efficiently approximate.

Handle multiple exit-conditions. A loop may contain more than one exit and hence may have more than one exit condition. In principle, we consider a loop to be scalable, unless the analysis on every exit condition shows that the loop is non-scalable.

3.6.5 Global loop count analysis. After loop-local analysis, every type-3 non-scalable loop candidate L and its loop-count approximation V are passed on for global analysis, which checks whether there exists an I/O operation O that can contribute to the value of V.

This checking could be done dynamically — instrumenting every I/O operation and every memory access, and checking the dependency on-line or through a trace. Unfortunately, the overhead of such dynamic slicing analysis is huge [4, 63]. This checking could also be done statically — analyzing the program control and data flow graphs to see whether any I/O operation O could contribute to the value of V along any path p that might reach L. However, such a path p may not be feasible at run time. Consequently, we use a hybrid analysis. We first conduct static analysis, which will be explained in detail below, and then run the software to see whether any such path p is feasible under the testing workloads.

Algorithm 2 shows an outline of our analysis. At the high level, our analysis maintains a *contributor* working-set $\mathbb{C}$ — every member element of $\mathbb{C}$ is a variable c whose value at program location l_c contributes to the value of V. $\mathbb{C}$ is initialized as a set that contains V. In every iteration of the working loop shown in Algorithm 2, we pop out an element c, such as V at the first iteration. We analyze backward along the data-flow graph to find every update to c, `c = exp`$_c$, that might be used by the read of c. Then three different situations could happen: (1) if `exp`$_c$ is a constant, our analysis moves on to analyze the next element in the working set $\mathbb{C}$; (2) if `exp`$_c$ is an I/O function, such as `listFiles()` in Figure 6, we add `exp`$_c$ into the output set; (3) otherwise, we analyze `exp`$_c$ to potentially add more contributors into $\mathbb{C}$ before we move on to analyze the next contributor in $\mathbb{C}$. When $\mathbb{C}$ becomes empty, we check the output set $\mathbb{O}$. If it is empty, we conclude that L is scalable; if it is not empty, we move on to dynamic analysis to see whether these contributing I/O operations in $\mathbb{O}$ can indeed execute at run time.

Next, we discuss how we analyze expression `exp`$_c$ to find potential I/O operations that contribute to its value.

Intra-procedural baseline analysis. Given an expression `c = exp`, we simply put all operands in *exp* into the working set $\mathbb{C}$ (e.g., `blockList.length` and `LONGS_PER_BLOCK` for line 14 of Figure 6).

We need to pay attention to augmentation operations like `c += exp` or `Collection::add(..)`, such as line 18 in Figure 6. In addition to adding variables involved in *exp* into the working set, we also analyze whether this statement is enclosed in a loop L'. If it is, we add the loop-count variable of L' into the working set — if `x++` is conducted for N times, with N being workload-sensitive, we consider the value of `x` also workload-sensitive.

Intra-procedural collection analysis. We adapt the above algorithm slightly to accommodate for loops and operations related to collections and arrays.

Given a loop L whose loop count is approximated by the size of a collection or an array C, we search backward along the data-flow graph for update operations that can affect the collection size, such as `Collection::add(ele)`, `Collection::addAll (set)`, or the array length, such as `new String[len]`, instead of operations that update the content but not the size of C, such as `Collection::fill(..)`, `Collection:: reverse()`, etc.

For every size update, we add corresponding variables or collection sizes into $\mathbb{S}$, such as `len` for `new String[len]` (i.e., 0 for `new long [0]` on line 10 of Figure 6), and the size of `set` for `Collection:: addAll(set)`. We also distinguish augmentation operations such as `Collection ::add(ele)` from regular assignment such as `new String[len]`, and analyze the enclosing loop for augmentation operations as discussed earlier.

Inter-procedural loop-count analysis. If a contributor v is assigned by a function return, `v = foo(..)` (e.g., `NumBlocks()` on line 17 of Figure 6), we will put the return value of `foo` into the contributor working set $\mathbb{C}$ and then analyze inside `foo` to see how its return value is computed (e.g., line 14 of Figure 6). Of course, if `foo` is an I/O function itself, we directly add `foo` into the output set $\mathbb{O}$.

When a contributor variable v is the parameter of function `foo` or is used as an argument in invoking a callee function $\texttt{foo}_*$, we conduct inter-procedural analysis. In the former case, we identify every caller function $\texttt{foo}^*$, and add the corresponding function-call argument into the contributor working set. In the latter case, since $\texttt{foo}_*$ can modify the content of v when v is passed as a reference, we trace reference variables through WALA's default alias analysis for further data dependence analysis.

4 PCATCH IMPLEMENTATION AND EVALUATION

4.1 Implementation

We have implemented the PCatch static analysis tool using the WALA Java byte code analysis infrastructure [26], and the PCatch dynamic tracing tool using Javassist, a dynamic Java bytecode transformation framework [27]. PCatch does not require large-scale distributed-system deployment to detect PCbugs — different *nodes* of a distributed system can be deployed on either different physical machines or different virtual machines in a small number of physical machines. Next, we explain some implementation details such as how PCatch identifies I/O operations and how PCatch implements run-time tracing.

In the current prototype of PCatch, we consider the following API calls as I/O operations: file-related APIs, including both Java library APIs like `java.io.InputStream::read` and Hadoop library APIs `fs.rename`, network Java APIs like `java.net.InetAddress::getByName`, and RPC calls. As mentioned in Section 3.6, PCatch uses static analysis to identify loops that conduct time-consuming operations in loop bodies. We consider all the I/O operations mentioned above as expensive, and also consider `sleep` related functions as expensive. Our analysis checks whether any such expensive API is called in the loop body, including the callee functions of a loop.

The PCatch cascading analysis (Section 3.2) is conducted upon run-time traces, which are generated for every thread of the target distributed system. Every trace records the following three types of operations.

First, synchronization and communication operations related to must-HB analysis, such as RPC calls, message sending/receiving, event enqueue/dequeue, thread creation and join, and other causal operations.

Second, resource acquisition and release operations that are related to may-HB analysis. For lock contention, we record all object-level and class-level lock and unlock operations, such as synchronized method entrances and

BugID	Workload	Sinks
CA-6744	write table + cleanup table	Streaming
HB-3483	write table	Region assignment, RegionServer write
HD-2379 HD-5153	write file	Heartbeat, DataNode write, NameNode write
MR-2705 MR-4088 MR-4576 MR-4813	wordcount	Heartbeat, Map task

Table 2. Evaluation benchmarks. (Different HD and MR bugs affect different versions.)

exits, synchronized block entrances and exits, explicit locks and unlocks, etc. For thread-pool contention, we record the start and end of every RPC/event/message handler and which thread it is running on, which in fact is already recorded for must-HB analysis.

Third, PCbug source and sink candidates. Specifically, we identify every loop in the program and record the start of every loop. We also record the execution of every instance of `_sink_start` and `_sink_end`.

Every trace record contains three pieces of information: (1) the type of the recorded operation, (2) the callstack of the recorded operation, and (3) an ID that uniquely identify the operation or the resource under contention.

For causal operations, the IDs will allow PCatch trace analysis to correctly apply may-HB and must-HB rules. For every event, thread, or lock operation, the ID is the object hashcode of the corresponding event, thread, or lock object. For an inter-node communication operation, PCatch tags each RPC call and each socket message with a random number generated at run time.

For source and sink candidates, IDs uniquely identify them in bug analysis and report. Every ID is a combination of keyword `source/sink`, `start/end`, and a mix of the corresponding class name, method name, and line numbers.

4.2 Evaluation methodology

4.2.1 Benchmarks. We evaluate PCatch using four widely used open-source distributed systems: the Hadoop MapReduce distributed computing framework (MR); the HBase distributed key-value stores (HB); the HDFS distributed file system (HD); the Cassandra distributed key-value stores (CA). These systems range from about 100 thousand lines of code to more than three million lines of code.

4.2.2 Workloads. Our experiments use common workloads, such as word-count for MapReduce, writing a file for HDFS, updating a table for HBase, and updating and then cleaning a table for Cassandra. We were aware of 8 different reports in corresponding bug-tracking systems, as shown in Table 2, where severe performance slowdowns or node failures were observed by users under similar types of workloads at *large* scales.

Note that, although triggering PCbugs requires large-scale workloads and sometimes special timing among threads/events/messages. PCatch **predicts** PCbugs by monitoring execution under a **small-scale** workload **without** any requirements on special timing. More details about PCatch bug detection workloads are in Section 4.6.

4.2.3 Time-sensitive sinks. PCatch provides APIs to specify sinks – code regions whose execution time users/developers care about. A naive way to specify a sink is to put every client request into a sink. However, in practice, finer-granularity sinks would work better, as finer-granularity performance cascading analysis is not only faster but also more accurate.

BugID	Detected?	#Static $\langle L, S \rangle$s		#Static $\langle L, C, S \rangle$s	
		Bugs	FalsePositives	**Bugs**	FalsePositives
CA-6744	✓	$\mathbf{2}_2$	0	$\mathbf{2}_2$	0
HB-3483	✓	$\mathbf{2}_1$	0	$\mathbf{3}_2$	0
HD-2379	✓	$\mathbf{6}_2$	5	$\mathbf{8}_4$	8
HD-5153	✓	$\mathbf{5}_5$	3	$\mathbf{5}_5$	4
MR-2705	✓	$\mathbf{2}_1$	0	$\mathbf{4}_1$	0
MR-4088	✓	$\mathbf{1}_1$	1	$\mathbf{1}_1$	2
MR-4576	✓	$\mathbf{2}_1$	1	$\mathbf{2}_1$	2
MR-4813	✓	$\mathbf{2}_2$	1	$\mathbf{2}_2$	1
Total		$\mathbf{22}_{15}$	11	$\mathbf{27}_{18}$	17

Table 3. PCatch bug detection results (Subscript denotes bug reports related to the known bug listed in Column 1.)

In our experiments, we specify important system routines (e.g., heartbeats) and main tasks within a client request (e.g., region assignment and region server write task in HBase) as sinks, as shown in Table 2. We mostly simply surround a function call with `_sink_start` and `_sink_end`. Take the heartbeat sink illustrated in Figure 1 as an example. The *TaskTracker* process sends heartbeats to the *JobTracker* process by invoking the `transmitHeartBeat` function. Consequently, we simply put a `_sink_start` right before the function call and a `_sink_end` right after the function call. In practice, developers can put `_sink_start` and `_sink_end` anywhere, as long as they make sure that `_sink_start` executes before `_sink_end`.

4.2.4 Experiment settings. We run each node of a distributed system in a virtual machine, and run all VMs in two physical machines that use Intel® Xeon® CPU E5-2620, and 64GB of RAM. In practice, these systems are mostly deployed on more than one physical machine. Fortunately, PCatch can predict bugs using small-scale workloads without relying on time profiling.

We report PCbug counts by both the unique number of static source-sink pairs, short as $\langle L, S \rangle$), and the unique number of static triplets $\langle L, C, S \rangle$. These two counts' results are mostly similar. All our performance numbers are based on an average of 5 runs.

To confirm whether a PCatch bug report is indeed a PCbug, we increase the corresponding workloads following the loop-source reported by PCatch. We then measure (1) whether the execution time of the source indeed increases; and (2) whether the execution time of the sink has a similar and sufficiently large increase. The detailed results will be presented in Table 7.

4.3 Bug detection evaluation

Overall, PCatch successfully detects PCbugs for all benchmarks while monitoring *correct* execution of these applications, as shown by the ✓ in Table 3. In addition, PCatch found a few truly harmful PCbugs we were unaware of. PCatch is also accurate: only about one third of all the PCatch bug reports are false positives.

4.3.1 True bugs. PCatch successfully predicts the 8 PCbug benchmarks using small-scale workloads. These 8 benchmarks were originally noticed by users and developers under much larger workloads and many runs as we will discuss in Section 4.6.

PCatch also found a few harmful PCbugs, 7 unique source-sink pairs (9 unique source-chain-sink triplets), that we were unaware of, as shown in Table 4. We have triggered all of them successfully, and then carefully checked

BugID	What does the source loop do?	Sink	Chains	Loop Job	Sink Job	Loop
CA-6744$_1$	table-cleanup	Streaming	1P	Table Cleanup$_U$	File Streaming$_S$	Type3
CA-6744$_2$	metadata-scan	Streaming	1P	Table Cleanup$_U$	File Streaming$_S$	Type3
HB-3483$_1$	region-flush	RegionServer Write	1L/1L	HBase Write-1$_U$	HBase Write-2$_U$	Type2
HB-3483$_2$	**region state-check**	**RegionServer Write**	**1L**	**Timeout Monitor$_S$**	**HBase Write$_U$**	**Type3**
HD-2379$_1$	blocks-scan	DateNode Write	1L/1L	BlockReport Scanner$_S$	HDFS Write$_U$	Type3
HD-2379$_2$	**blocks-scan**	**HeartBeat**	**1L**	**BlockReport Scanner$_S$**	**HeartBeat$_S$**	**Type3**
HD-2379$_3$	blockreport-generate	DataNode Write	1L/1L	BlockReport Scanner$_S$	HDFS Write$_U$	Type3
HD-2379$_4$	**blockreport-generate**	**HeartBeat**	**1L**	**BlockReport Scanner$_S$**	**HeartBeat$_S$**	**Type3**
HD-2379$_5$	**removedblock-process**	**NameNode Write**	**1L**	**BlockReport$_S$**	**HDFS Write$_U$**	**Type3**
HD-2379$_6$	**addedblock-process**	**NameNode Write**	**1L**	**BlockReport$_S$**	**HDFS Write$_U$**	**Type3**
HD-5153$_1$	first-blockreport-process	NameNode Write	1L	BlockReport$_S$	HDFS Write$_U$	Type3
HD-5153$_2$	removedblock-process	NameNode Write	1L	BlockReport$_S$	HDFS Write$_U$	Type3
HD-5153$_3$	addedblock-process	NameNode Write	1L	BlockReport$_S$	HDFS Write$_U$	Type3
HD-5153$_4$	underconstructionblock-process	NameNode Write	1L	BlockReport$_S$	HDFS Write$_U$	Type3
HD-5153$_5$	corruptedblock-process	NameNode Write	1L	BlockReport$_S$	HDFS Write$_U$	Type3
MR-2705$_1$	file-download	Map Task	1P	MapReduce-1$_U$	MapReduce-2$_U$	Type2
MR-2705$_2$	**file-download**	**HeartBeat**	**2L/2L/3L**	**MapReduce-1$_U$**	**HeartBeat$_S$**	**Type2**
MR-4088$_1$	job init-wait	Map Task	1P	MapReduce-1$_U$	MapReduce-2$_U$	Type1
MR-4576$_1$	file-download	HeartBeat	3L	MapReduce-1$_U$	HeartBeat$_S$	Type2
MR-4576$_2$	**job init-wait**	**Map task**	**1P**	**MapReduce-1$_U$**	**MapReduce-2$_U$**	**Type1**
MR-4813$_1$	files-commit	HeartBeat	1L	MapReduce-1$_U$	HeartBeat$_S$	Type3
MR-4813$_2$	files-move	HeartBeat	1L	MapReduce-1$_U$	HeartBeat$_S$	Type3

Table 4. True PCbugs reported by PCatch. New bugs outside the benchmarks are in bold fonts. The "Chains" column lists the number of resource contentions involved in performance cascading and the type of resources (P: thread pool; L: lock); different chains for the same pair of source and sink are separated by "/". Subscript U: user-submitted jobs; Subscript S: system background or periodic jobs.

the change log of each software project, and found that 6 out of these 7 $\langle L, S\rangle$ pairs (7 out of 9 $\langle L, C, S\rangle$ triplets) have been patched in the latest versions of these systems.[3]

Our experiments judge whether a harmful PCbug is successfully triggered in the following way. We run software using a workload, designed based on the PCatch bug report (see Section 4.6), that is larger than the one used during PCatch bug detection but still has reasonable size. We then monitor the duration of the sink T_{sink}. For every PCbug mentioned above, we have observed that T_{sink} increases from less than 1 second to around 10 seconds or more, or from a few seconds to a few minutes, once the non-deterministic performance-propagation chain reported by PCatch is hit, indicating that delays are indeed unintentionally propagated from one job to

[3]HB-3621, HADOOP-4584, HD-5153, MR-1895, MR-2209, MR-4088

BugID	Alternate may-HB		No Origin Analysis		Alternative Loop Analysis	
	#LS	#LCS	#LS	#LCS	#LS	#LCS
CA-6744	4	7	2	4	7	8
HB-3483	1	3	1	2	5	5
HD-2379	10	53	2	2	6	16
HD-5153	6	8	0	0	12	21
MR-2705	7	9	2	5	8	25
MR-4088	9	12	3	6	3	3
MR-4576	9	12	4	9	6	7
MR-4813	7	17	0	0	9	14

Table 5. Extra false positives in $\langle L, S\rangle$ count and in static $\langle L, C, S\rangle$ count for alternative designs.

another through resource contention and causal operations. More bug-triggering details will be presented in Section 4.6.

Table 4 shows all the true bugs reported by PCatch. As we can see, PCatch can detect PCbugs caused by a wide variety of non-scalable loops and resource-contention chains.

4.3.2 False-positive bug reports. PCatch reports 11 false positives. 3 of them are caused by inaccurate static analysis of time-consuming operations. For example, some I/O wrapper functions in these systems use Java I/O APIs in an asynchronous way and hence do not introduce slowdowns at the call site. For 8 of them, their source loops' bounds are indeed determined by content returned by file systems, network, or users' commands. However, program semantics determine that those contents have a small upper-bound.

In our triggering experiments, we easily identify these false positives — when we increase the workload size based on PCatch bug report, we observe that the duration of the source loop, and consequently the corresponding sink, increases little, if at all.

4.4 Comparison with alternative designs

PCatch contains three key components. To evaluate the effectiveness of each component, we tried removing part of each component while keeping the other two components unchanged, and measure how many extra false positives would be introduced. Specifically, for cascading analysis, we slightly revised the may-HB rules (discussed in Section 3.3.2), so that two lock critical sections or two event/RPC handlers do not need to be concurrent with each other in order to have a may-HB relationship; for job origin analysis (Section 3.5), we tried simply skip this check; for non-scalable loop analysis, we tried declaring every loop that contains I/O operations in the loop body as a potential PCbug candidate source. As we can see in Table 5, the above three alternative designs all lead to many extra false positives, even when analyzing exactly the same traces as those used to produce results in Table 3. Clearly, all three components of PCatch are important in PCbug detection.

4.5 Performance results

As shown in Table 6, PCatch performance is reasonable for in-house testing. In total, PCatch bug detection incurs 3.7X–5.8X slowdown compared with running the software **once** under the **small-scale** bug-detection workload. Note that, without timing manipulation, many of these bugs do **not** manifest even after running the software under **large-scale** bug-triggering workloads of many runs. In comparison, PCatch can greatly improve the chances of catching these PCbugs in a much more efficient way during in-house testing.

Table 6 also shows the three most time-consuming components of PCatch: run-time tracing, trace-based cascading analysis, and static loop scalability analysis. As we can see, PCatch tracing consistently causes 1.1X

BugID	Base	Total PCatch bug detection	Tracing	Cascading Analysis	Scalability Analysis	Trace Size
CA-6744	120	703 (5.8x)	150	103	319	31MB
HB-3483	28	123 (4.5x)	47	11	30	9.4MB
HD-2379	73	268 (3.7x)	78	20	96	4.5MB
HD-5153	118	433 (3.7x)	132	17	158	11MB
MR-2705	44	223 (5.1x)	58	22	96	18MB
MR-4088	65	250 (3.8x)	85	26	67	11MB
MR-4576	83	373 (4.5x)	134	38	106	31MB
MR-4813	71	383 (5.4x)	93	55	156	24MB

Table 6. Performance of PCatch (total) and main components, base is the run time without PCatch. Unit: seconds.

BugID	Detection Run		Triggering Run		
	Workload Size	Sink Time	Workload Size	Sink Time !Buggy	Sink Time Buggy
CA-6744	1-row table	1.44s	10,000-row table	1.80s	8.6s
HB-3483	1K data	0.02s	~300M data	0.05s	62s
HD-2379	3 blocks	0.22s	~835,000 blocks	0.22s	19s
HD-5153	3 blocks	0.05s	~786,000 blocks	0.06s	14s
MR-2705	1KB sidefile	3.83s	1GB sidefile	4.29s	12m21s
MR-4088	1KB sidefile	4.37s	1GB sidefile	4.98s	12m32s
MR-4576	1KB sidefile	0.03s	1GB sidefile	0.03s	12m42s
MR-4813	1 reducer	0.07s	1000 reducers	0.12s	22s

Table 7. Detection run vs. triggering run (!Buggy: measured when the resource contention did not occur; Buggy: measured when the resource contention occurred)

– 1.7X slowdowns across all benchmarks. The cascading analysis is relatively fast. In comparison, static loop scalability analysis is the most time consuming, Note that, more than half (up to 80%) of the analysis time is actually spent for WALA to build the whole-program dependency graph (PDG) in every benchmark except for CA-6744. This PDG building time actually can be shared among all analyses on the same software project. Future work can also speed up the static analysis using parallel processing — analyzing the scalability of different loops in parallel. The execution time of the dynamic part of loop-scalability analysis and job-original analysis is part of the PCatch total bug-detection time, but is much less than the three components presented in Table 6.

4.6 Triggering results

We consider a PCbug to be triggered, if we observe (1) the sink executes much slower in a run when resource contention occurs than when contention does not occur;[4] and (2) the amount of slowness increases with the workload size.

We have successfully triggered all the true PCbugs detected by PCatch (i.e., every one in Table 4), with the details of the 8 benchmark bugs' triggering listed in Table 7.

Note that, without the guidance of PCatch bug reports, triggering these PCbugs is extremely difficult. In fact, without carefully coordinating the timing, which we will discuss below, only **4 out of 22** PCbugs manifest themselves after **10** runs under **large** workloads listed in Table 7. In these 4 PCbugs, the source can affect the

[4]During triggering runs, we use featherweight instrumentation to check if the reported resource contention happens.

heartbeat through contention of just one lock. Since heartbeat function is executed periodically (usually once every 3 seconds in MapReduce and HDFS), there is a good chance that the resource contention would happen to at least one heartbeat instance.

To trigger the remaining 18 PCbugs, we have to carefully coordinate the workload timing (i.e., when to submit a client request) in order to make the resource contention occur. This is true even for bug MR-2705_2 and MR-4576_1 in Table 4, which, although they have periodic heartbeat functions as sinks, require multiple lock contentions to cause the buggy performance to cascade.

These results show that PCatch can greatly help discover PCbugs during in-house testing — it can catch a PCbug (1) under regular timing without requiring rare timing to trigger resource contentions; (2) under small-scale workload that runs many times faster than large workloads.

The current triggering process is not automated yet, although it benefits greatly from PCatch bug reports. It mainly requires two pieces of manual work. First, based on the result of the PCatch loop-scalability analysis, we figure out which aspect of the workload size can affect the execution time of the loop source (e.g., should we increase the number of mappers or should we increase the size of a file), and prepare a larger workload accordingly, such as those shown in Table 7. Second, we monitor whether the bug-related performance propagation chain is triggered or not at run time, and conduct time coordination to help trigger that chain. Among these two pieces of manual effort, the second one can be automated in the future by techniques similar to those that automatically trigger concurrency bugs [36, 39]; the first is more difficult to automate, and may require advanced symbolic execution and input-generation techniques.

Note that, PCatch is capable of detecting PCbugs that can only be triggered by large clusters through experiments on small clusters (e.g., the bug's source loop count is determined by the number of cluster nodes). However, we did not encounter such bugs in our experiments and hence none of the triggering workload requires different cluster sizes.

4.7 Discussion

PCatch could have false positives and false negatives for several reasons.

(1) Performance-dependence model: PCatch cascading analysis is tied with our may-HB and must-HB models. It may miss performance dependencies caused by semaphores, custom synchronizations, and resource contentions currently not covered by our must-HB and may-HB models, resulting in false negatives.

(2) Workload and dynamic analysis: PCatch bug detection is carefully designed to be largely oblivious to the size of the workload and the timing of the bug-detection run. However, PCatch would still inevitably suffer false negatives if some bug-related code is not executed during bug-detection runs (e.g., the loop sources, I/O operations inside a loop source, causal operations, resource-contention operations, sinks), which is a long standing testing coverage problem.

(3) Static analysis: PCatch's scalability analysis intentionally focuses on common patterns of non-scalable loops in order to scale to analyzing large distributed systems, but it could miss truly non-scalable loops that are outside the three types discussed in Section 3.6, and hence lead to false negatives. On the other hand, some loops that scale well may be mistakenly reported as non-scalable loops and lead to false positives.

5 RELATED WORK

5.1 Performance bug detection

Many tools have been built to detect performance problems common in single-machine systems, such as performance-sensitive API misuses [28], inefficient and redundant loop computation [42–44], object bloat [18, 55], low-utility data structures [56], cache-line false sharing [37, 40], problematic inputs [17], etc. In general, these

tools have a different focus and hence a completely different design from PCatch. Particularly, previous loop-inefficiency detectors [42–44] analyze loops to detect inefficient computation inside a loop, such as redundant computation among loop iterations or loop instances, and dead writes performed by the majority of the loop iterations. The main difference between PCatch and these studies is that PCatch's loop analysis is focuses not on efficiency but on scalability (i.e., how the execution time of a loop scales with the workload size).

MacePC [30] detects non-deterministic performance bugs in distributed systems developed upon MACE [29], a language with a suite of tools for building and model checking distributed systems. It essentially conducts model checking for MACE systems – while traversing the system state space, it reports cases where a special timing can cause significant slowdown of the system. The model checking technique and the program analysis techniques used by PCatch complement each other: model checking provides completeness guarantees if finished, but suffers from state explosion problems and may only work for systems built upon a special framework like MACE.

5.2 Causality analysis in distributed systems

Many studies have been done to model, trace, and analyze causal relationships in distributed systems [11, 32, 36, 38], and leverage the knowledge of causal relationships to detect functional bugs [36].

PCatch used the previous DCatch model and techniques [36] in its causal relationship analysis. However, as discussed in Section 3.3.1, analyzing such causal relationships is only one component of the cascading analysis in PCatch. To detect cascading performance bugs in distributed systems, PCatch has to go much beyond causal relationship analysis: PCatch has to also model resource-contention (may-HB) relationships and compose them together with traditional causal relationships (must-HB); furthermore, PCatch needs to conduct job-origin analysis, which is not needed in DCatch concurrency-bug detection, and loop-scalability analysis, which is completely unrelated to traditional causal relationship analysis.

5.3 Lock contention and impact analysis

Many tools have been proposed to profile lock contention [15, 45, 60]. Recently proposed SyncPerf [6] profiles runtime lock-usage information, such as how frequently a lock is acquired and contended, to diagnose performance problems related to frequently acquired or highly contended locks, including load imbalance, asymmetric lock contention, over synchronization, improper primitives, and improper granularity.

Past research also looked at how to identify high-impact performance bottlenecks in multi-threaded software. Coz [14] tries to insert delays at various places in software to measure the performance impact at different program locations. Coz only considers waits that are incurred by lock contention and that *have already* been recorded in the trace. Coz builds graphs to represent such wait relationship. By analyzing the graph and other information, Coz can figure out which code regions or critical sections have the biggest impact on a past run. Similarly, SyncProf [58] and work by Yu et al. [59] tries to identify performance bottlenecks by analyzing traces of many runs.

PCatch is related to previous works that analyze inefficient lock usages and lock contention. However, PCatch has fundamentally different goals and hence designs from the above works. First, PCatch aims to *predict* performance problems that may happen in the future, *not* to diagnose problems that have been observed. Consequently, the cascading analysis in PCatch models potential dependence and performance cascading, instead of lock contention that has already happened. Second, PCatch looks at resource contention inside distributed systems, which goes beyond locks. Third, PCatch focuses on PCbugs that violate scalability and performance-isolation properties, which is different from pure resource contention problem. For example, many locks in PCbugs may be neither highly contended nor frequently acquired, different from those in the bugs found by SyncPerf [6].

TaxDC [33] is a comprehensive taxonomy of non-deterministic concurrency bugs that studies distributed concurrency (DC) bugs from Cassandra, Hadoop MapReduce, HBase and ZooKeeper. Our analysis (Section 3) follows the methodology of TaxDC to present the taxonomy of performance bugs in distributed systems.

5.4 Scalability problems in software systems

Profiling techniques have been proposed to help developers discover code regions that do not scale well with inputs [13, 19, 54, 61]. Previous work has proposed techniques to better evaluate scalability of distributed systems by co-locating many distributed nodes in a single machine [23, 51]. These profiling and testing techniques are orthogonal to the performance-problem prediction conducted by PCatch.

5.5 Performance anomaly diagnosis

Many tools have been built to diagnose system performance anomalies [7, 9, 16, 46–48, 53], including many built for diagnosing and reasoning about performance problems in distributed systems [5, 11, 38, 49, 57, 64]. Some of them [11, 64] can approximate some performance dependency relationships by analyzing a huge number of traces together. PerfScope [25] conducts lightweight performance risk analysis (PRA) to improve performance regression diagnosis via testing target prioritization. All these tools focus on performance diagnosis based on large number of run-time traces, which is orthogonal to the bug-prediction goal of PCatch.

6 CONCLUSIONS

This paper systematically studies 99 distributed performance bugs from five widely-deployed distributed storage and computing systems (including Cassandra, HBase, HDFS, Hadoop MapReduce and ZooKeeper). We collectively organize the analysis results as over 400 classification labels and over 2,500 lines of bug re-description in the *TaxPerf* database. We further analyze the cascading nature of an important category of blocking bugs and design a automatic detection tool called *PCatch*, which enables users to detect PCbugs under small workloads and regular non-bug-triggering timing. We believe PCatch is just a starting point in tackling performance problems in distributed systems. Future work include (i) extending PCatch to detect other categories of performance bugs and (ii) automatically generating patches [35] for performance bugs.

ACKNOWLEDGMENTS

This research is supported by the National Key Research and Development Program of China (2022YFB4500302), the Scientific Research Program of National University of Defense Technology (ZK20-03), the National Natural Science Foundation of China (61872376, 61932001), and the Natural Science Foundation of Hunan Province of China (2022JJ40555). This journal submission presents substantial improvements of a previous version published in ACM EuroSys 2018 (PCatch) [34]. Specifically, the journal submission first conducts studies of 99 distributed performance bugs from five widely-deployed cloud distributed systems (Cassandra, HBase, HDFS, Hadoop MapReduce and ZooKeeper), and then introduces PCatch (of the EuroSys 2018 submission) as the solution for an important category (blocking bugs) of all performance bugs studied in this paper. Yiming Zhang is the corresponding author.

REFERENCES

[1] [n.d.]. Apache HBase Project. http://hbase.apache.org.
[2] [n.d.]. Apache ZooKeeper Project. http://zookeeper.apache.org.
[3] [n.d.]. HDFS Architecture. http://hadoop.apache.org/common/docs/current/hdfs_design.html.
[4] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic program slicing. In *PLDI*. 246–256.
[5] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP* (Bolton Landing, NY, USA). 74–89. https://doi.org/10.1145/945445.945454

[6] Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. In *EuroSys* (Belgrade, Serbia). 298–313. https://doi.org/10.1145/3064176.3064186

[7] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. 2010. Performance Analysis of Idle Programs. In *OOPSLA* (Reno/Tahoe, Nevada, USA). 739–753. https://doi.org/10.1145/1869459.1869519

[8] Apache. [n.d.]. MapReduce-4576. https://issues.apache.org/jira/browse/MAPREDUCE-4576.

[9] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-cause Diagnosis of Performance Anomalies in Production Software. In *OSDI*. 307–320. http://dl.acm.org/citation.cfm?id=2387880.2387910

[10] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*.

[11] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *OSDI*. 217–231. http://dl.acm.org/citation.cfm?id=2685048.2685066

[12] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell C Sears. 2009. MapReduce Online. UC. Berkeley Technical Report No. UCB/EECS-2009-136.

[13] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-sensitive Profiling. In *PLDI* (Beijing, China). 89–98. https://doi.org/10.1145/2254064.2254076

[14] Charlie Curtsinger and Emery D Berger. 2015. C oz: finding code that counts with causal profiling. In *SOSP*. 184–197.

[15] Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2014. Continuously measuring critical section pressure with the free-lunch profiler. In *OOPSLA*. 291–307.

[16] Daniel Joseph Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. 2014. PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures. In *SOCC*. 1 – 13.

[17] Dongdong Deng, Wei Zhang, and Shan Lu. 2013. Efficient concurrency-bug detection across inputs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 785–802. https://doi.org/10.1145/2509136.2509539

[18] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2008. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *FSE*. 59–70. https://doi.org/10.1145/1453101.1453111

[19] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. 2007. Measuring Empirical Computational Complexity. In *ESEC-FSE*. 395–404. https://doi.org/10.1145/1287624.1287681

[20] Sumit Gulwani. 2009. SPEED: Symbolic Complexity Bound Analysis. In *CAV*. 51–62.

[21] Sumit Gulwani and Florian Zuleger. 2010. The Reachability-bound Problem. In *PLDI*. 292–304.

[22] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. cbs. In *SoCC*.

[23] Diwaker Gupta, Kashi Venkatesh Vishwanath, and Amin Vahdat. 2008. DieCast: testing distributed systems with an accurate scale model. In *NSDI*. 407–421.

[24] Herodotos Herodotou, Fei Dong, and Shivnath Babu. 2011. No One Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*.

[25] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. 2014. Performance regression testing target prioritization via performance risk analysis. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 60–71. https://doi.org/10.1145/2568225.2568232

[26] IBM. [n.d.]. Main Page - WalaWiki. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[27] jboss javassist. [n.d.]. Javassist. http://jboss-javassist.github.io/javassist/.

[28] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *PLDI*. 77 – 88.

[29] Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. 2007. Mace: Language support for building distributed systems. In *PLDI*. 179 –188.

[30] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. 2010. Finding Latent Performance Bugs in Systems Implementations. In *FSE*. 17 – 26.

[31] Avinash Lakshman and Prashant Malik. 2009. Cassandra - a decentralized structured storage system. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*.

[32] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.

[33] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, Tom Conte and Yuanyuan Zhou (Eds.). ACM, 517–530. https://doi.org/10.1145/2872362.2872374

[34] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. 2018. Pcatch: automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, Rui Oliveira, Pascal Felber, and Y. Charlie Hu (Eds.). ACM, 7:1–7:14. https://doi.org/10.1145/3190508.3190552

[35] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 715–726. https://doi.org/10.1145/2950290.2950309

[36] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. 2017. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *ASPLOS*. 677–691.

[37] Tongping Liu, Chen Tian, Ziang Hu, and Emery D. Berger. 2014. PREDATOR: Predictive False Sharing Detection. In *PPoPP*. 3–14.

[38] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *SOSP*. 378–393.

[39] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*. 267–280.

[40] Mihir Nanavati, Mark Spear, Nathan Taylor, Shriram Rajagopalan, Dutch T. Meyer, William Aiello, and Andrew Warfield. 2013. Whose Cache Line is It Anyway?: Operating System Support for Live Detection and Repair of False Sharing. In *EuroSys*. 141–154.

[41] Robert H. B. Netzer and Barton P. Miller. 1991. Improving the accuracy of data Race detection. In *PPoPP*. 133–144.

[42] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *ICSE*. 902–912.

[43] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: detecting performance problems via similar memory-access patterns. In *ICSE*. 562–571.

[44] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *PLDI* (Portland, OR, USA). 369–378. https://doi.org/10.1145/2737924.2737966

[45] Oracle. [n.d.]. HPROF: A heap/cpu profiling tool. http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html.

[46] Kai Shen, Ming Zhong, and Chuanpeng Li. 2005. I/O System Performance Debugging Using Model-driven Anomaly Characterization. In *FAST*. 309–3222.

[47] Linhai Song and Shan Lu. 2014. Statistical Debugging for Real-World Performance Problems. In *OOPSLA*. 561–578.

[48] Christopher Stewart, Ming Zhong, Kai Shen, and Thomas O'Neill. 2006. Comprehensive Depiction of Configuration-dependent Performance Anomalies in Distributed Server Systems. In *HOTDEP*.

[49] Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. 2010. Visual, Log-Based Causal Tracing for Performance Debugging of MapReduce Systems. In *ICDCS*. 795–806. https://doi.org/10.1109/ICDCS.2010.63

[50] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. 2018. Understanding and Auto-Adjusting Performance-Sensitive Configurations. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 154–168. https://doi.org/10.1145/3173162.3173206

[51] Yang Wang, Manos Kapritsos, Lara Schmidt, Lorenzo Alvisi, and Mike Dahlin. 2014. Exalt: empowering researchers to evaluate large-scale storage systems. In *NSDI*. 129–141.

[52] Mark Weiser. 1981. Program slicing. In *ICSE*. 439–449.

[53] Alexander Wert, Jens Happe, and Lucia Happe. 2013. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *ICSE*. 552–561.

[54] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. 2013. Context-sensitive Delta Inference for Identifying Workload-dependent Performance Bottlenecks. In *ISSTA* (Lugano, Switzerland). 90–100. https://doi.org/10.1145/2483760.2483784

[55] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: profiling copies to find runtime bloat. In *PLDI* (Dublin, Ireland). ACM, New York, NY, USA, 419–430. https://doi.org/10.1145/1542476.1542523

[56] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding low-utility data structures. In *PLDI*. 174–186.

[57] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting Large-scale System Problems by Mining Console Logs. In *SOSP* (Big Sky, Montana, USA). 117–132. https://doi.org/10.1145/1629575.1629587

[58] Tingting Yu and Michael Pradel. 2016. SyncProf: detecting, localizing, and optimizing synchronization bottlenecks.. In *ISSTA*. 389–400.

[59] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. 2014. Comprehending Performance from Real-world Execution Traces: A Device-driver Case. In *ASPLOS* (Salt Lake City, Utah, USA). 193–206. https://doi.org/10.1145/2541940.2541968

[60] Piotr Zalewski and Jinwoo Hwang. [n.d.]. IBM thread and monitor dump analyze for Java. https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=2245aa39-fa5c-4475-b891-14c205f7333c.

[61] Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic Profiling. In *PLDI* (Beijing, China). 67–76. https://doi.org/10.1145/2254064.2254074

[62] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. 2014. Heading off Correlated Failures Through Independence-as-a-service. In *OSDI*. 317–334.
[63] Xiangyu Zhang, Sriraman Tallam, and Rajiv Gupta. 2006. Dynamic slicing long running programs through execution fast forwarding. In *FSE*. 81–91.
[64] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *OSDI*. 603–618.