PCatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems

Jiaxin Li[*]
NUDT, University of Chicago
licyh@nudt.edu.cn

Yuxi Chen, Haopeng Liu, Shan Lu
University of Chicago
{chenyuxi,haopliu,shanlu}@uchicago.edu

Yiming Zhang
NUDT
ymzhang@nudt.edu.cn

Haryadi S. Gunawi
University of Chicago
haryadi@cs.uchicago.edu

Xiaohui Gu
North Carolina State University
xgu@ncsu.edu

Xicheng Lu and Dongsheng Li
NUDT
{xclu,dsli}@nudt.edu.cn

ABSTRACT

Distributed systems have become the backbone of modern clouds. Users often expect high scalability and performance isolation from distributed systems. Unfortunately, a type of poor software design, which we refer to as performance cascading bugs (PCbugs), can often cause the slowdown of non-scalable code in one job to propagate, causing global performance degradation and even threatening system availability.

This paper presents a tool, PCatch, that can automatically predict PCbugs by analyzing system execution under small-scale workloads. PCatch contains three key components in predicting PCbugs. It uses program analysis to identify code regions whose execution time can potentially increase dramatically with the workload size; it adapts the traditional happens-before model to reason about software resource contention and performance dependency relationship; it uses dynamic tracking to identify whether the slowdown propagation is contained in one job or not. Our evaluation using representative distributed systems, Cassandra, Hadoop MapReduce, HBase, and HDFS, shows that PCatch can accurately predict PCbugs based on small-scale workload execution.

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; **Software reliability**; **Software testing and debugging**;

KEYWORDS

Performance Bugs; Cascading problems; Distributed Systems; Bug Detection; Cloud Computing

ACM Reference Format:
Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, and Xicheng Lu and Dongsheng Li. 2018. PCatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal,* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3190508.3190552

[*]The work was done when Jiaxin Li was a visiting student at University of Chicago.

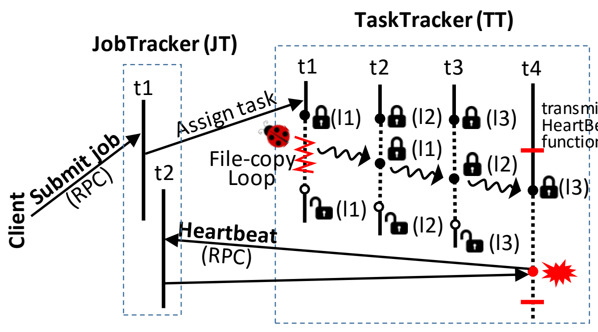

Figure 1: A PCbug in MapReduce: the client's job delays heartbeats through lock contentions and causes the whole Tasktracker node to fail.

1 INTRODUCTION

1.1 Motivation

Distributed systems, such as cloud storage, data-parallel computing frameworks, and synchronization services, have become a dominant backbone for modern clouds. Users often expect high *scalability* and performance *isolation* of these systems, given their inherent distributed and redundant nature. Unfortunately, poor software design could violate both properties at once, causing the slowdown of one job to propagate and affect other jobs or even the whole system. We refer to this type of design problem as a performance cascading bug, as a PCbug for short.

Figure 1 illustrates a real-world PCbug [5] 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.

As demonstrated by this example, PCbugs are often triggered by large workloads and end up with global performance problems, violating both scalability and performance-isolation expectations. 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.

The following properties of PCbugs make them difficult to tackle:

- Workload sensitive. The amount of slowdown incurred by a PCbug varies on workloads. Unfortunately, in most cases, only small workloads are available during in-house testing, which makes PCbugs difficult to expose and catch before software release.
- Non-deterministic. Software-resource competition and the corresponding slowdown propagation are often non-deterministic. Consequently, even running software under large workloads does *not* guarantee to expose PCbugs.
- Global symptoms. The cascading nature of PCbugs makes the performance-failure diagnosis difficult — it requires global analysis to identify the slowness propagation chains and the slowdown root causes.

Although extensive studies have been conducted on detecting performance problems in single-machine systems, such as loop inefficiency problems [31–33], cache false-sharing problems [26, 29], and lock contention problems [3, 10, 48], 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.

1.2 Contribution

In this paper, we present a tool PCatch that automatically detects PCbugs through a combination of static and dynamic program analysis. 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. We refer to such a user job or a system routine as a *sink* in the paper.

Specifically, PCatch identifies a $\langle L, C, S\rangle$ 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.

Performance cascading model and analysis. We model two types of relationships that could contribute to performance cascading. One is the traditional causality, also called must-happens-before, relationship [24, 25, 27, 30] that forces one operation to execute after another due to deterministic program semantics. The other is a non-deterministic relationship, which we call may-happens-before, that may cause one code region to slow down another code region due to non-deterministic software-resource contentions. Our model considers not only lock-based resource contention, but also event handling contention and RPC handling contention that are unique to asynchronous computation and inter-node communication in distributed systems. This model and the corresponding analysis enables us to systematically and accurately *predict* how the slowdown at one part of a distributed system could non-deterministically affect another part in *future* runs. The details are presented in Section 4.

Job identity analysis. We use dynamic analysis to automatically identify which code regions belong to the same user job or the same system routine. This analysis helps us differentiate local performance-cascading problems from global performance-cascading problems, with the latter relevant to PCbugs. The details are presented in Section 5.

Non-scalable source identification. Our analysis adapts traditional loop-bound analysis and program slicing to efficiently identify time-consuming loops whose execution time may increase with the workload changes. This analysis helps us to identify potential *sources* of performance cascading problems without running the software under large-scale workloads. The details are presented in Section 6.

We evaluated PCatch on widely used distributed systems, including Cassandra, HBase, HDFS and Hadoop MapReduce. We test small-scale workloads on these systems. We knew that these workloads, when at a much larger scale, triggered 8 cascading performance failures reported by users. 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 3.7X–5.8X slowdown, suitable for in-house use.

2 BACKGROUND

As mentioned in Section 1.2, one component of our performance cascading model is the traditional causality relationship [24, 25, 27, 30]: 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 previous work, DCatch [25]. We briefly present them below. More details can be found in the DCatch paper.

2.1 Causal relationship model

The DCatch model mainly includes the following causality rules that reflect a wide range of communication and concurrency-control operations in real-world distributed cloud systems. We will refer to those operations discussed below that directly lead to causal relationships as *causal operations.*

2.1.1 Inter-node rules. Inter-node causality relationships reflect various types of inter-node communication.

Synchronous RPC rules. A thread in node n_1 could call an RPC function r implemented by node n_2. This thread will block until n_2 sends back the RPC execution result, and thus the invocation of r on n_1 happens before the beginning of the RPC execution on n_2; the end of the RPC execution on n_2 happens before the return from r on n_1.

Asynchronous Socket rules. A thread in node n_1 sends a message m to node n_2 through network sockets. Unlike that in RPC, the sender does not block, so we only have the relationship that the sending happens before the receiving.

Custom synchronization rules. The DCatch model also contains a rule that reflects the notification mechanism provided by synchronization services like ZooKeeper [18], and a rule that represents distributed while-loop synchronization.

2.1.2 Intra-node rules. Two types of concurrent computation often co-exist in a distributed system node, synchronous multi-threading and asynchronous event processing, and contribute to different types of causality relationships.

Synchronous multi-threaded rules. These are the most traditional types of causality relationship: the creation of a thread (or process) t in the parent p happens before the execution of t starts; the end of t's execution happens before a successful join of t inside p; a signal operation happens before a corresponding wait operation.

Asynchronous event-driven rules. Events could be enqueued by any thread, and then processed by pre-defined handlers in event-handling thread(s). This process contributes to several types of causal relationships: (1) the enqueue of an event e happens before the handler-function of e starts; (2) when two events e_1 and e_2 are sent to the same single-worker FIFO queue, the start of e_1 happens before the start of e_2 if the enqueue of e_1 happens before the enqueue of e_2; (3) the end of an event's handler function happens before a successful join of this event in any thread (e.g., `Future::get()`)[1].

Finally, an operation o_1 happens before an operation o_2 from the same thread t that occurs later than o_1 in t. When t is an event-handler thread, the above relationship only applies when o_1 and o_2 are from the same event handler.

[1]This rule did not appear in the original DCatch paper and is added in the PCatch implementation.

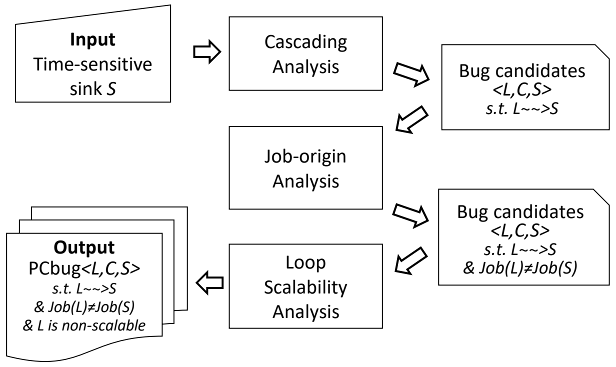
Input
Time-sensitive sink S
Cascading Analysis
Bug candidates <L,C,S> s.t. L~~>S
Job-origin Analysis
Bug candidates <L,C,S> s.t. L~~>S & Job(L)≠Job(S)
Loop Scalability Analysis
Output
PCbug<L,C,S> s.t. L~~>S & Job(L)≠Job(S) & L is non-scalable

Figure 2: PCatch overview

2.2 Causal relationship analysis

Given the causality rules above, once we know all the causal operations, the causal relationship between any pair of program operations can be computed.

DCatch analyzes causal relationships by building and analyzing a **happens-before (HB) graph**, a technique also used by much previous work [13, 24, 35, 36]. Specifically, the DCatch run-time tracer records all the causal operations and other operations useful for DCatch concurrency-bug detection. It then constructs the DAG happens-before graph, with every vertex v representing an operation $o(v)$ in the trace and every edge reflecting exactly one causality rule described in Section 2.1. In this HB graph, a node v_1 can reach a node v_2 if and only if $o(v_1)$ happens before $o(v_2)$. When there are neither paths connecting v_1 to v_2 nor paths connecting v_2 to v_1, their corresponding operations are concurrent with each other, denoted as $o(v_1)//o(v_2)$.

PCatch traces causal operations and uses them to construct **HB graphs** exactly as DCatch does. PCatch uses this graph to decide whether two operations are concurrent and what are the causors of a given operation.

3 PCATCH OVERVIEW

This section provides an overview of how PCatch predicts a PCbug $\langle L, C, S \rangle$ following the analysis flow illustrated in Figure 2.

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 [21]. We analyze potential causality and software-resource contentions based on our cascading model, which will be presented in Section 4. 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 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 explained in Section 6. After this step, all the remaining candidates are reported by PCatch as PCbugs.

4 CASCADING ANALYSIS

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**[2] 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.

Challenges. There are three key challenges here.

- 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.

[2]Hardware resource contention can also cause lack of performance isolation [52], but is out of the scope of PCbugs.

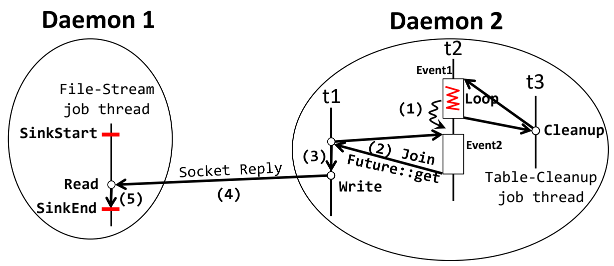

Figure 3: A real PCbug in Cassandra, where a slow clean-up loop in one job could slow down another job on a different node due to competition on the event-handling thread t2. $\rightarrow$ represents must-HB and $\rightsquigarrow$ represents may-HB.

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.

4.1 PCatch Performance Cascading Model

We will first discuss causality relationships and resource- contention relationships separately below in Section 4.1.1 and Section 4.1.2, followed by a discussion in Section 4.1.3 about how to compose them together to reason about performance cascading relationships in distributed systems.

4.1.1 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. All the causal relationships modeled in PCatch have been discussed in Section 2.

4.1.2 Resource contention (may-HB) relationships. Under resource contention, one resource-acquisition operation could nondeterministically 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, vice versa. That is, 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 $\texttt{lock}_1\texttt{(l)} \mathbin{//} \texttt{lock}_2\texttt{(l)}$, we can infer that $\texttt{o}_2 \rightsquigarrow \texttt{lock}_1\texttt{(l)}$ and $\texttt{o}_1 \rightsquigarrow \texttt{lock}_2\texttt{(l)}$, where $\texttt{o}_1$ and $\texttt{o}_2$ 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, as demonstrated in Figure 1. 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 could be unexpectedly delayed by the execution of the other event/RPC handler, as shown by thread t_2 in the Daemon-2 node in Figure 3. 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 $\texttt{o}_1 \rightsquigarrow \texttt{start}_{h_2}$ and $\texttt{o}_2 \rightsquigarrow \texttt{start}_{h_1}$, where $\texttt{o}_1$ and $\texttt{o}_2$ are any operations inside handler functions h_1 and h_2, respectively.

The above may-HB relationship is transitive — if $A \rightsquigarrow B \rightsquigarrow C$, we know that $A \rightsquigarrow C$ as long as $A//C$.[3] 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 $\rightsquigarrow$ `lock(`l_1`)` in $\texttt{t}_2 \rightsquigarrow$ `lock(`l_2`)` in $\texttt{t}_3 \rightsquigarrow$ `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.

4.1.3 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}

[3]$A//B$ 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

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 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.

4.2 PCatch cascading analysis

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 will discuss the tracer implementation in Section 7.2. 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 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.

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

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

Figure 4: Data-dependence may not affect loop bound (Line 1 affects the content but not the size of `vector`, which defines the loop count).

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.

6 LOOP SCALABILITY ANALYSIS

Goals. As discussed in Section 3, 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 is straightforward and we present the implementation details in Section 7.1. Identifying loops that satisfy the second condition is more challenging and is the focus of this section.

Challenges. Identifying non-scalable loops is related to but different from loop complexity analysis [15, 16] and program slicing [42], 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 4). 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 4. 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 6.2 and Figure 5 (in Figure 5, 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).

Our Solutions. Since accurately computing the loop count and identifying all non-scalable loops is impractical [15, 16], we give up on soundness and completeness, and instead aim to identify

```
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>
```

Figure 5: A simplified loop example from HDFS

```
while (rjob.localing) {
    rjob.wait();
}
```

Figure 6: Synchronization loop in MapReduce (`rjob.localing` is a loop invariant if not consider other threads)

```
while (bytesRead >= 0) {
    ...
    bytesRead = in.read(buf);
}
```

Figure 7: MapReduce Loop with workload-sensitive index

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 4) and the loop index bound (e.g., `vector.size()` in Figure 4), 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 6).
(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 7).
(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 5, 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 6.1) 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 6.2). 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.

6.1 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 [19] to identify all the loop exit instructions and the branch conditions 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 6, 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 8, 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 8, as an approximation for the loop-count’s upper bound. The global analysis in Section 6.2 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 7. 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 9. 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.

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

Figure 8: A loop with constant stride in HDFS

```
for (Block block: toAdd) {
    addStoredBlock(block, ...);
    ...
}
```

Figure 9: Collection-related loop in HDFS

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

Algorithm 2: Global loop count analysis algorithm

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 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.

6.2 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 [1, 53]. 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 5, 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 5).

We need to pay special attention to augmentation operations like `c += exp` or `Collection::add(..)`, such as line 18 in Figure 5. 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 the 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 5), 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 5), 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 5). 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.

7 IMPLEMENTATION

We have implemented the PCatch static analysis using the WALA Java byte code analysis infrastructure [19], and the PCatch dynamic tracing using Javassist, a dynamic Java bytecode transformation framework [20]. 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.

7.1 I/O operations and PCbug source candidates

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 common library APIs `fs.rename`, network-related Java APIs like `java.net.InetAddress::getByName`, and RPC calls. As mentioned in Section 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.

7.2 PCatch tracing

The PCatch cascading analysis (Section 4) 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 discussed in Section 2.

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 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.

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 1: Evaluation benchmarks. (Different HD and MR bugs affect different versions.)

8 EVALUATION

8.1 Methodology

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.

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 1, where severe performance slowdowns or node failures were observed by users under similar types of workloads at *large* scales.

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

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 8.5.

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.

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 1. 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`.

Experiment settings. We run each node of a distributed system in a virtual machine, and run all VMs in two physical machines that use Ubuntu 14.04, JVM v1.7, 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 6.

8.2 Bug detection results

Overall, PCatch successfully detects PCbugs for all benchmarks while monitoring *correct* execution of these applications, as shown by the ✓ in Table 2. 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.

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 8.5.

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 3. We have triggered all of them successfully, and then carefully checked 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.[4]

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 8.5), 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 another through resource contention and causal operations. More bug-triggering details will be presented in Section 8.5.

Table 3 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.

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.

8.3 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 4.1.2), 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 5), we tried simply skip this check; for non-scalable loop

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

BugID	What does the source loop do?	Sink	Chains	Loop Job	Sink Job	Loop Type
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 3: 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.

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

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

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 4, 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 2. Clearly, all three components of PCatch are important in PCbug detection.

8.4 Performance results

As shown in Table 5, 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 5 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 – 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 5.

8.5 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

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

when contention does not occur;[5] 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 3), with the details of the 8 benchmark bugs' triggering listed in Table 6.

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 6. In these 4 PCbugs, the source can affect the 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 3, 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 6. 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 [25, 28]; 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.

8.6 Discussion

PCatch is neither sound nor complete. 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 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, as discussed in Section 8.2.

9 RELATED WORK

Performance bug detection. Many tools have been built to detect performance problems common in single-machine systems, such as performance-sensitive API misuses [21], inefficient and redundant loop computation [31–33], object bloat [12, 45], low-utility data structures [46], cache-line false sharing [26, 29], etc. In general, these tools have a different focus and hence a completely different design from PCatch. Particularly, previous loop-inefficiency detectors [31–33] 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. PCatch also analyzes loops, but focuses on not efficiency but scalability (i.e., how the execution time of a loop scales with the workload size).

MacePC [23] detects non-deterministic performance bugs in distributed systems developed upon MACE [22], 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

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

completeness guarantees if finished, but suffers from state explosion problems and may only work for systems built upon a special framework like MACE.

Causality analysis in distributed systems. As mentioned in Section 2, much research has been done to model, trace, and analyze causal relationships in distributed systems [7, 24, 25, 27], and leverage the knowledge of causal relationships to detect functional bugs [25].

PCatch used the previous DCatch model and techniques [25] in its causal relationship analysis. However, as discussed in Section 4.1.1, 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.

Lock contention and impact analysis. Many tools have been proposed to profile lock contention [10, 34, 50]. Recently proposed SyncPerf [3] profiles run-time 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 [9] tries inserting delays at various places in software to measure the performance impact at different program locations. SyncProf [48] and work by Yu et al. [49] tries to identify performance bottlenecks by analyzing traces of many runs. They both consider waits incurred by lock contention that *have already* been recorded in the trace, and build graphs to represent such wait relationship. By analyzing the graph and other information, they figure out which code regions or critical sections have the biggest impact on a past run.

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 [3].

Scalability problems in software systems. Profiling techniques have been proposed to help developers discover code regions that do not scale well with inputs [8, 14, 44, 51]. Previous work has proposed techniques to better evaluate scalability of distributed systems by co-locating many distributed nodes in a single machine [17, 41]. These profiling and testing techniques are orthogonal to the performance-problem prediction conducted by PCatch.

Performance anomaly diagnosis. Many tools have been built to diagnose system performance anomalies [4, 6, 11, 37–39, 43], including many built for diagnosing and reasoning about performance problems in distributed systems [2, 7, 27, 40, 47, 54]. Some of them [7, 54] can approximate some performance dependency relationships by analyzing a huge number of traces together. 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.

10 CONCLUSIONS

Performance cascading bugs (PCbugs) could severely affect the performance of distributed systems violating scalability and performance-isolation properties, and are difficult to catch during in-house testing. Our PCatch tool, including cascading analysis, job-origin analysis and loop scalability analysis, 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 cascading problems in distributed systems. Future work can extend PCatch to not only detect but also fix PCbugs leveraging PCatch bug reports and analysis techniques.

ACKNOWLEDGMENTS

We would like to thank Julia Lawall, our shepherd, and the anonymous reviewers for their insightful feedback and comments. This research is partly supported by the National Key Research and Development Program of China (2016YFB1000101), the National Natural Science Foundation of China (61222205, 61772541), NSF (CNS-1563956, IIS-1546543, CNS-1514256, CCF-1514189, CCF-1439091) and generous supports from CERES Center for Unstoppable Computing.

REFERENCES

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, 1990.
[2] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, pages 74–89, 2003.
[3] Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *EuroSys*, pages 298–313, 2017.
[4] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *OOPSLA*, pages 739–753, 2010.
[5] Apache. Mapreduce-4576. https://issues.apache.org/jira/browse/MAPREDUCE-4576.
[6] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, pages 307–320, 2012.
[7] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *OSDI*, pages 217–231, 2014.
[8] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *PLDI*, pages 89–98, 2012.
[9] Charlie Curtsinger and Emery D Berger. C oz: finding code that counts with causal profiling. In *SOSP*, pages 184–197, 2015.
[10] Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Continuously measuring critical section pressure with the free-lunch profiler. In *OOPSLA*, pages 291–307, 2014.
[11] Daniel Joseph Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In *SOCC*, pages 1 – 13, 2014.

[12] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *FSE*, pages 59–70, 2008.

[13] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[14] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. Measuring empirical computational complexity. In *ESEC-FSE*, pages 395–404, 2007.

[15] Sumit Gulwani. Speed: Symbolic complexity bound analysis. In *CAV*, pages 51–62, 2009.

[16] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304, 2010.

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

[18] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *ATC*, pages 11–25, 2010.

[19] IBM. Main page - walawiki. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[20] jboss javassist. Javassist. http://jboss-javassist.github.io/javassist/.

[21] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77 – 88, 2012.

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

[23] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *FSE*, pages 17 – 26, 2010.

[24] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[25] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. DCatch: Automatically detecting distributed concurrency bugs in cloud systems. In *ASPLOS*, pages 677–691, 2017.

[26] Tongping Liu, Chen Tian, Ziang Hu, and Emery D. Berger. Predator: Predictive false sharing detection. In *PPoPP*, pages 3–14, 2014.

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

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

[29] Mihir Nanavati, Mark Spear, Nathan Taylor, Shriram Rajagopalan, Dutch T. Meyer, William Aiello, and Andrew Warfield. Whose cache line is it anyway?: Operating system support for live detection and repair of false sharing. In *EuroSys*, pages 141–154, 2013.

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

[31] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. CARAMEL: detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*, pages 902–912, 2015.

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

[33] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, pages 369–378, 2015.

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

[35] Veselin Raychev, Martin T. Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *OOPSLA*, pages 151–166, 2013.

[36] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[37] Kai Shen, Ming Zhong, and Chuanpeng Li. I/O system performance debugging using model-driven anomaly characterization. In *FAST*, pages 309–3222, 2005.

[38] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *OOPSLA*, pages 561–578, 2014.

[39] Christopher Stewart, Ming Zhong, Kai Shen, and Thomas O'Neill. Comprehensive depiction of configuration-dependent performance anomalies in distributed server systems. In *HOTDEP*, 2006.

[40] Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Visual, log-based causal tracing for performance debugging of mapreduce systems. In *ICDCS*, pages 795–806, 2010.

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

[42] Mark Weiser. Program slicing. In *ICSE*, pages 439–449, 1981.

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

[44] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ISSTA*, pages 90–100, 2013.

[45] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *PLDI*, pages 419–430, 2009.

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

[47] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, pages 117–132, 2009.

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

[49] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: A device-driver case. In *ASPLOS*, pages 193–206, 2014.

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

[51] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *PLDI*, pages 67–76, 2012.

[52] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. Heading off correlated failures through independence-as-a-service. In *OSDI*, pages 317–334, 2014.

[53] Xiangyu Zhang, Sriraman Tallam, and Rajiv Gupta. Dynamic slicing long running programs through execution fast forwarding. In *FSE*, pages 81–91, 2006.

[54] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *OSDI*, pages 603–618, 2016.