IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 7, 2019, accepted June 8, 2019, date of publication June 19, 2019, date of current version August 7, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2923956

Understanding and Statically Detecting Synchronization Performance Bugs in Distributed Cloud Systems

CHEN ZHANG, JIAXIN LI, DONGSHENG LI, AND XICHENG LU
Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha 410073, China

Corresponding author: Chen Zhang (zc1994nudt@163.com)

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB0204300, and in part by the National Natural Science Foundation of China under Grant 61872376.

ABSTRACT In such an information society, the Internet of Things (IoT) plays an increasingly important role in our daily lives. With such a huge number of deployed IoT devices, Cyber-Physical System (CPS) calls for powerful distributed infrastructures to supply big data computing, intelligence, and storage services. With the increasingly complex distributed software infrastructures, new intricate bugs continue to manifest, causing huge economic loss. Synchronization performance problems, which means that improper synchronizations may degrade the performance and even lead to service exception, heavily influence the entire distributed cluster, imperiling the reliability of the system. As one kind of performance problems, the synchronization performance problems are acknowledged as difficult to diagnosis and fix. We collect 26 performance issues in three real-world distributed systems: HDFS, Hadoop MapReduce, and HBase, and do analysis on their root cause, fix strategy, and algorithm complexity in order to understand these synchronization performance bugs better. Then, we implement a static detection tool including critical section identifier, loop identifier, inner loop identifier, expensive loop identifier, and pruning component. After that, we evaluate our detection tool on these three distributed systems with sampled bugs. In the evaluation, our detection tool accurately finds out all the target bugs. Besides, it points out more new potential performance problems than the previous works. With the strict performance overhead, our detection tool is proved to be greatly efficient.

INDEX TERMS Software debugging, software performance, distributed computing, software reliability.

I. INTRODUCTION

In such an information society, Internet of Things (IoT) plays an increasingly important role in our daily lives, which means that computing and communication capabilities can be embedded in diverse objects in the physical environment. With such a huge number of deployed IoT devices, Cyber-Physical System (CPS) calls for powerful distributed infrastructures to supply big data computing, intelligence, and storage services.

Distributed software infrastructures such as large-scale storage systems, scale-out computing frameworks, synchronization services, and cluster management services, have emerged as a dominant backbone for CPS. A mass of applications begin to be developed based on distributed cloud systems, hence, the reliability of these systems is important.

With the increasingly complex distributed software infrastructures, new intricate bugs continue to manifest, causing huge economic loss. Inside, synchronization performance problems make up a large proportion. As one kind of performance problems, they are acknowledged as difficult to diagnosis and fix [1].

Users expect great dependability and efficiency from distributed systems, given the large-scale fundamental infrastructures. With a wide range of applications, performance problems are non-negligible to service quality for information technology enterprise. A few seconds' delay in distributed cloud systems could cause service exceptions.

Unfortunately, it is difficult to guarantee the reliability due to the complexity of software and its non-deterministic runtime. Among all types of performance bugs in distributed

The associate editor coordinating the review of this manuscript and approving it for publication was Zhen Ling.

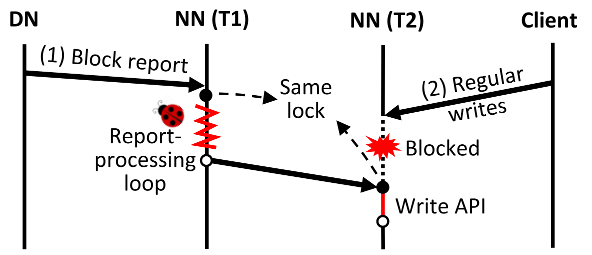
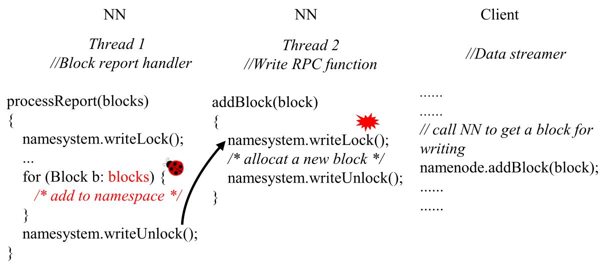


FIGURE 1. A HDFS synchronization performance bug: all writes to NameNode(NN T2) blocked since NameName(NN T1) runs a report processing from DataNode(DN), which contains a time-consuming loop.



```
Thread 1
//Block report handler
processReport(blocks)
{
   namesystem.writeLock();
   ...
   for (Block b: blocks) {
     /* add to namespace */
   }
   namesystem.writeUnlock();
}
```

```
Thread 2
//Write RPC function
addBlock(block)
{
   namesystem.writeLock();
   /* allocat a new block */
   namesystem.writeUnlock();
}
```

```
//Data streamer
......
......
// call NN to get a block for
writing
namenode.addBlock(block);
......
......
```

FIGURE 2. Root cause of the synchronization performance bug shown in Fig. 1.

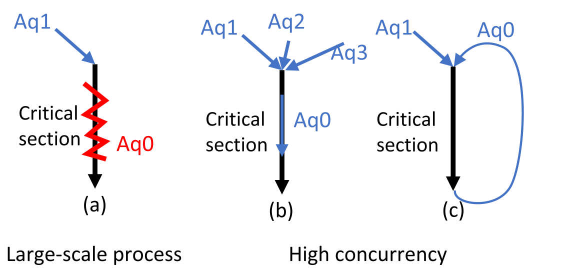


FIGURE 3. The two category of synchronization performance bug. (c) can be a specific instance of (b).

systems, synchronization performance bugs are particularly troublesome. Different from other performance bugs, which generally are caused by unnecessary operations, synchronization performance bugs are more the situations that important tasks are delayed by other workload-sensitive tasks causing performance degradation even service exception.

Figure 1 illustrates a real-world synchronization performance bug from HDFS. It is triggered by an unexpected time-consuming source on NameNode (NN). Specifically, after DataNode (DN) sends a big block report (i.e., Job 1) to NN, NN processes the report through a heavy report-processing loop where the maximum number of iterations is undefined while holds NN's lock. The loop iteration number is related to the number of blocks in DN. However, when NN is still in the process, all regular writes (i.e., Job 2) to HDFS have been blocked and wait for the loop to finish since they need the same lock.

Synchronization bugs or lock-related problems are common in distributed systems. With a strict synchronization mechanism, a thread swell may affect other components, the whole node, even the entire cluster. For instance, a HBase bug says that heavy writes load on a RegionServer (RS). The RS hits its global memstore *upperlimit* and begin to flush data to disk until the global memstore size downs to the *lowerlimit*. It is a time-consuming process which block all writes on the same region. Synchronization problems like this heavily influence performance of the entire distributed cluster, imperiling the reliability of the system.

Thus, we aim to conduct a study around synchronization performance problems in distributed system and address them according to our study. We hope our work can provide certain help for later research work.

Plenty previous studies [2], [3] related to performance bugs have been done. In Jin *et al.*'s paper [1], they extracted efficiency-related rules from performance bug patches and successfully detected new bugs based on these rules. Their work provides us a guidance, but limits exist at the same time. They only studied bugs in single machine, and the efficiency-related rules do not make sense in distributed systems. Following their guidance, our paper discusses the performance issues in different distributed infrastructures.

Besides, SyncPerf [4], which focus on synchronization performance problems, has done a series of studies around lock. Though SyncPerf has made a good discussion on lock-related performance bug, the dynamic tools rarely cover each execution path, meanwhile increase the number of false negatives. So we explore to detect synchronization performance bug with static method which can obtain a better code coverage and find out more potential synchronization performance problems.

Especially, though PCatch [5] also studies performance bugs in distributed systems, the bugs studied in that paper are just a subset of our studies. PCatch looks at the propagation chain and studies the relationship between source and sink. In their work, source is just the preliminary candidates of a bug. And to manifest the bugs, large-scale task is generally needed. PCacth can use relatively small scale tasks to detect source bug. However, there are also false negatives in PCatch and PCatch cannot get a full coverage of the program code or affectable sink, which means the limited input domain my lead to poor coverage. Different from their work, we only focus on the source since it is the key factor that introduce potential synchronization performance problems even though it cannot introduce performance bugs at present.

As show in Figure 3, synchronization performance bugs are generally including (1) large-scale process in a critical section (2) high frequency of software contention and resource acquisitions caused by large numbers of callers. It is, however, unrealistic to achieve the scale and detect through dynamic testing. The high contention of software resources has been well studied in previous works [1], [6]–[11], so synchronization performance bugs triggered by large-scale processing are what this paper concerns.

TABLE 1. Application and Bugs used in the study. MapReduce stands for distributed computing framework, HBase stands for distributed key-value stores and HDFS stands for distributed file systems.

Application	#Bugs
Hadoop MapReduce	7
HBase	8
HDFS	11
Total	26

In order to get a comprehensive solution and address synchronization performance problems, guided by the previous work [1], [11], [12], we have made the following contributions.

A. CHARACTERISTICS STUDY

Many empirical studies [1], [2], [13] have been conducted for performance bugs, but rare researches focus on synchronization performance bugs in cloud distributed systems. We collect 26 performance issues in distributed systems, and do analysis on their root cause, fix strategy and time complexity in order to understand these synchronization performance bugs better.

B. STATIC DETECTION

Owing to the poor coverage in previous work with dynamic analysis, we finally choose building static tool to implement detection, which is also our exploring method. We implement the critical section identification and group them by lock objects. Large-scale process identification is also expensive loop identification, analyzing the program to figure out the relationships among nested loops, time-consuming operations and the recursive call.

We evaluated our detection tool on 3 varying real-world distributed systems, HDFS, Hadoop MapReduce, and HBase. We collect 26 performance bugs and make a preliminary study, then pick out 13 synchronization related bugs with clear description in them and conduct further analysis, finally randomly pick out 5 bugs as benchmarks to test our static analysis tool.

II. BUG STUDY

In this section, we conduct a comprehensive performance bug study. The following research questions are studied.

1. How do we collect these bug samples?
2. What are the root causes of them?
3. What can we extract from the study above?
4. What others can we learn?

A. METHODOLOGY

We chose three widely deployed open-source distributed system suites to examine: Hadoop MapReduce, HBase and HDFS. As shown in Table 1, all of them provide a considerable coverage of distributed system types, such as distributed computing framework, distributed key-value stores and distributed file systems. They are all built with Java, which is popular in the development of commercial software.

TABLE 2. The "/" in "HB3483/HB2149" means HB3483 and HB2149 is actually the same bug. These two issues are redundant submits in JIRA or the same problem fixed by different strategies. For category, C1 means large-scale root cause and C2 means high-concurrency root cause. M,N,P,Q,R are used to mark the iteration number of each loop. Specially M stands for outer loop iteration of the critical section and N/P/Q/R stand for the inner loop iteration. "rec" means recursive call exists in the execution path. "✓" in the last two column means we identify that the effective factor contain "I/O", "RPC" or both.

Bug ID	Category	Algorithm Complexity	I/O	RPC
HB3483/HB2149	C1	$O(N) * O(P) * O(Q)$	✓	
HD2379	C1	$O(N) * O(P)rec * O(Q) * O(R)$	✓	
HD3990/HD4702	C1	$O(N) * (O(P))$		✓
HD4183/HD5790	C2	$[O(M)] * O(N) * O(P)$		
HD4186	C1	$O(N) * O(P) * O(Q)$	✓	✓
HD5064	C1	$O(N)rec * O(P)$	✓	
HD5241	C2	$[O(M)] * O(N)$	✓	
MR4576	C1	$O(N) * O(P)$	✓	✓
MR4813	C1	$O(N) * O(P)rec$		✓

From these applications, we can observe similarities and differences among synchronization performance bugs in various distributed systems suites.

All the bugs that we studied come from an open source cloud bug study (CBS) database [14]. This CBS database contains over 3,000 issues which are sampled organized issue repositories (named "JIRA" and maintained by Apache Software Foundation), over a period of three years(1/1/2011-1/1/2014). Each bug issue is labeled with multiple tags and we search synchronization performance bugs based on keywords including "performance" and "lock", "slow"/"delay" etc. From CBS, we firstly filtered out bug nor related to performance, then exclude performance bugs that have not been fixed, and finally randomly picked 26 samples from the remaining detailed bugs.

B. ROOT CAUSE ANALYSIS

As mentioned in Section I, synchronization performance bugs are generally triggered by (1) large-scale processing (2) high concurrency. We mainly focus on addressing the first one, which means to ask what happened in the critical section. To answer this question, we manually recursively analyzed plenty code lines in the problematic critical section.

Workload-sensitive slowdown is generally the key factor behind synchronization performance bugs, which violates the scalability goal of distributed systems. As the root cause of synchronization performance bug, workload-sensitive slowdown plays a vital role in the root cause of performance bug.

The unexpected slowdown of a source job could be triggered by special inputs, configuration or system scale, etc. Time-consuming sources, like block report processing in NameNode of HDFS (shown in Fig. 2), side file localization in TaskTracker of MapReduce, and memstore flushing in HRegionServer of HBase, usually consist of at least two components at the source code level. So the first step of our work is to clarify the algorithm complexity of these critical sections, and part of the results are shown in Table 2.

According to our study, there are mainly three key factors to synchronization performance bugs causing swollen critical section, which are time-consuming operation, nested loop, and recursive call.

1) TIME-CONSUMING OPERATION

In critical section, time-consuming operation is a fundamental factor causing slowdown of jobs, such as I/O-containing operations (e.g., Java libraries operation *java.io.InputStream* :: *read* or Hadoop common libraries operation *fs.rename*), network operations (e.g., Java libraries operation *java.net.InetAddress* :: *getByName* or RPC calls), and some special operations (e.g., *sleep*(100*ms*), *await*(1*s*)).

HD3990 is just a good example. Once the management web page *dfshealth.jsp* of NameNode is accessed, it will place a lock on the namespace and do DNS lookup (i.e., *InetAddress* :: *getByName*) for every DataNode and also do some other node checking referring to DNS lookup, which often results in over 10s load time for the page on a multi-thousand node cluster. In addition, 10 concurrent requests were found to cause more than 7min load times during which time all write operations in the NameNode are blocked.

2) NESTED LOOP

Loop is also an important factor to performance bugs. A slowdown related to unbounded loop is universally linear with the number of iterations of the loop, which generally come from specific configurable options (e.g., the number of local directories used for *dfs.data.dir* in HDFS).

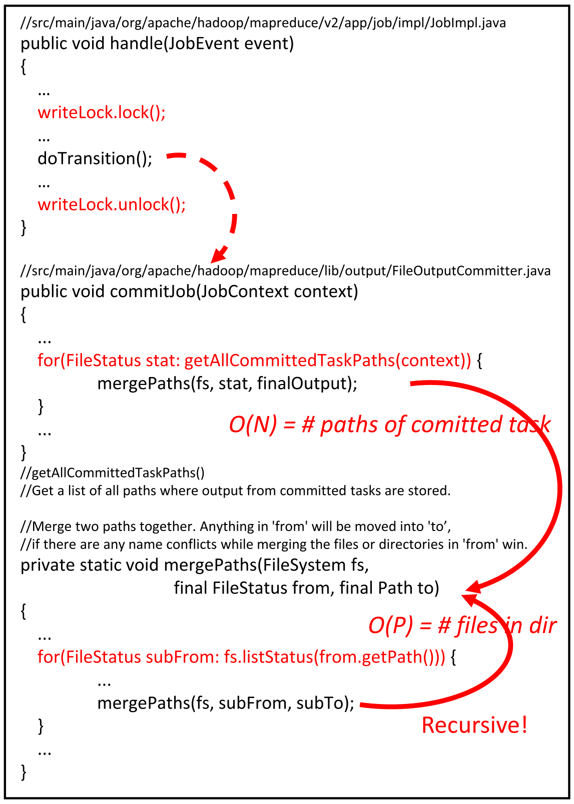
An important scenario is a non-scalable loop wrapping the time-consuming operation, the iteration number of which is usually decided by system-specific scalability parameters (e.g., the number of reported blocks in HDFS, the number of TaskTrackers in MapReduce), input data or commands (e.g., the number of side files for a MapReduce job), or intermediate results of workloads. In other words, the iteration of each loop can be illustrated with specific workload size.

Taking MR4813 as one example in Figure 4, it says that if user commits a big job with tons of files to finish the job, and this process holds JobImpl's write lock the entire time during committing, which prevents AppMaster (AM) sending heartbeat to Resource Manager (RM) for a long time, then RM regards AM as unresponsive and attempts to kill the AM, finally job fails.

The first step above is slow owing to moving large numbers of output files, and this process in Table 2 shows a complexity with $O(N) * O(P)rec$, in which N donates the number of directories in the committed job, and P donates the number of files in the current directory, additionally there is a recursive loop in *mergePath*() surrounding P. The details can be seen in Figure 4. So we can illustrate the slowdown with recursively traveling each file in the committed job and the total iteration count is the number of files in a job result.

3) RECURSIVE CALL

Recursive call is also a key factor to the jobs' slowdown. Though compared with general loop, they are both repeating the similar works, the difference is more important. Different from general loop which shows an obvious end condition,



```
//src/main/java/org/apache/hadoop/mapreduce/v2/app/job/impl/JobImpl.java
public void handle(JobEvent event)
{
  ...
  writeLock.lock();
  ...
  doTransition();
  ...
  writeLock.unlock();
}

//src/main/java/org/apache/hadoop/mapreduce/lib/output/FileOutputCommitter.java
public void commitJob(JobContext context)
{
  ...
  for(FileStatus stat: getAllCommittedTaskPaths(context)) {
          mergePaths(fs, stat, finalOutput);
  }
  ...
}
//getAllCommittedTaskPaths()
//Get a list of all paths where output from committed tasks are stored.

//Merge two paths together. Anything in 'from' will be moved into 'to',
//if there are any name conflicts while merging the files or directories in 'from' win.
private static void mergePaths(FileSystem fs,
                    final FileStatus from, final Path to)
{
  ...
  for(FileStatus subFrom: fs.listStatus(from.getPath())) {
          ...
          mergePaths(fs, subFrom, subTo);
  }
  ...
}
```

FIGURE 4. The complexity analysis of MR4813. In the *writeLock* critical section, there are a 2-depth loop and a recursive loop. Sign "$\dashrightarrow$" means going through several calls and sign "$\rightarrow$" means directly calling.

Recursive call is difficult to identify the end of the call chain. According our study, recursive call can be a greatly non-determinate structure which generally causes slowdown, especially for recursive loop. A recursive loop is a loop that recursively calls the function containing itself. Recursive loops could be time-consuming sources. For example in HD2379, DataNodes with multiple millions of blocks on a single machine recursively scan all the data directories to generate a block report, which can take multiple minutes when page cache space is tight, while it holds the FSVolumeSet lock then blocks all the writes and reads to current DataNode. Though recursive call is not a necessary reason responsible for time-consuming source, it indeed increases delay.

4) HIGH FREQUENCY LOCK

As seen in Table 2, the root causes of HD4183/HD5790 and HD5241 belong to high frequency. Looking at their algorithm complexity, they are $[O(M)]*O(P)*O(Q)$ and $[O(M)]*O(P)$. M represents the iteration number of the outer loop, which means the critical section is in a loop field. With such a continuous lock acquisition, it heavily influences other threads' resource contention.

C. FINDINGS

According to the study above, we extracted several findings listed below. Some of these findings are also valuable tips for improving our detection work.

Finding 1: *The paired synchronization primitives, lock and unlock, are generally in the same function.*

This finding is extracted from MR4813, HB3483, HD5153 and etc. It supplies great convenience to static critical section identification, which means we can find out critical section by only analyzing codes of current method without considering codes of the callees. The detail can be seen in Section III-A.

Finding 2: *Bugs related to keyword "synchronized" are generally fixed by skipping the waiting time.*

From MR4576, HD2379 and etc., the keyword "synchronized" means the whole function is in critical section, which is also used in Section III-A. To avoid entering the "synchronized" field, using an asynchronous thread and using *tryLock*() are both the approaches skipping waiting for lock acquisition. This finding may not help to identify synchronization performance problems but can help to analyze the evaluation results that how the bug disappears with the increasing version of packages.

Finding 3: *Nested loop shows a strong workload-sensitive feature.*

Nested loops are common in large-scale process, and they easily show a linearly increasing consuming time due to the unbounded variant in the loops. We regard nested loop as a necessary condition of slowdown.

Finding 4: *Recursive loop is strongly related to workload-sensitive slowdown.*

Learned from HD2379, HD5064 and MR4813, recursive loops are generally merging tree structure or traveling file systems, and they are both workload-sensitive. All the recursive loops we studied are all the root cause of slowdown, so we try to regard recursive loop as a sufficient condition of slowdown.

Finding 5: *Time-consuming operations are mainly the direct cause of expensive loop.*

Almost, each synchronization performance bug in Table 2, the critical sections contain time-consuming operations such as I/0 or RPC. Besides, disk Read/Write and net Read/Write are much slower than memory operation. We regard time-consuming operation as a necessary condition of slowdown.

D. FURTHER DISCUSSION

Synchronization performance bugs are usually fixed by changing the source or the propagation chain. For example, developers sometimes add timeouts to the source loop or move part of the loop out of critical sections or out of the loop thread (i.e., using asynchronous processing). Developers sometimes break the cascading chain (propagate from slowdown to the sink) by skipping waiting for certain contention resources or creating more resources to share.

III. SYNCHRONIZATION PERFORMANCE BUG DETECTION

According to our study in Section II-C, loops are most likely to become performance bottlenecks. Thus, we mainly consider loops as potential large-scale processing problem-related performance bug candidates.

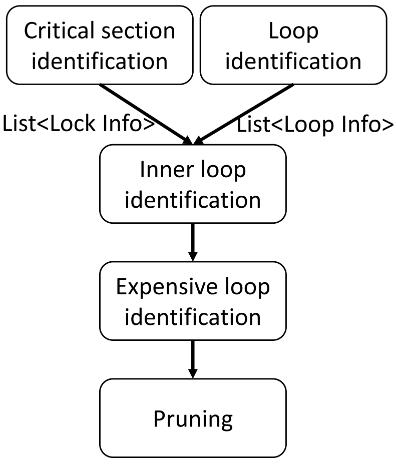

Critical section identification
Loop identification
List<Lock Info>
List<Loop Info>
Inner loop identification
Expensive loop identification
Pruning

FIGURE 5. The logic relationship among detection tool components.

There are four steps for detecting time-consuming sources. (1) Identify all critical sections protected by any locks; (2) Identify all loops and mark the functions containing loops; (3) Identify loops contained inside the critical sections got from Step 1; (4) Further identify vital critical sections that contain time-consuming loops that contain nested loops, I/Os or RPCs, recursive calls. (5) Removing pseudo time-consuming loops out of the loop candidates.

We have implemented the static analysis of synchronization performance bug detection using WALA, a Java byte code analysis infrastructure [15], more details seen in seciton IV.

A. CRITICAL SECTION IDENTIFICATION

It is challenging to identify critical sections because of various types of synchronization operations in distributed systems. Synchronization operations include (1) class-level locks such as *synchronized*(*CLASSNAME*.*class*), etc., and (2) object-level locks such as synchronized methods, *synchronized*(*obj*), *obj*.*lock*()/*unlock*(), or manually defined locks like *obj*.*readLock*()/*readUnlock*(), *obj*.*writeLock*()/*writeUnlock*(), etc.

In addition, it is also challenging to accurately identify the source code region scope between a pair of lock-require and lock-release instructions for a critical section, which is necessary for inner loop identification in Section III-C. In Java bytecode, however, every critical section may correspond to multiple lock-release instructions caused by different execution paths to exit the critical section.

Given a method M and its corresponding control-flow graph (CFG) G, we use WALA byte code analysis infrastructure to identify all the critical sections in M and take the set of contained basic blocks as the code region scope for a critical section.

Algorithm 1 shows an outline of static critical section analysis. First, if M is synchronized, we will consider the whole method as a critical section and put its basic block (BB) set BB_G into the output set $\mathbb{C}$. Second, we traverse all instructions of M (i.e., the *InstructionsOf*(M) function) to find

Algorithm 1 Critical Section Analysis Algorithm

```
Input: A method $M$ and its control-flow graph $G$
Output: A set of critical sections $\mathbb{C}$={$C$|$C$ is a critical section
        in $M$ }
Set<Critical section> $\mathbb{C} \leftarrow$ Null;
if $M$ is a synchronized method then
    new $C$
    $C \leftarrow BB_G$
    $\mathbb{C}.add(C)$
for $i$ in InstructionsOf(M) do
    if $i$ is a Lock-Require instruction then
        new $C$
        $BB_C \leftarrow$ Null
        for $j$ in Lock-ReleaseInstructionsOf(i) do
            $BB_C \leftarrow BB_C \cup (SuccBBs(BB_i) \cap PrecBBs(BB_j))$
        end
        $C \leftarrow BB_C$
        $\mathbb{C}.add(C)$
end
return $\mathbb{C}$;
```

Algorithm 2 Loop Field Analysis Algorithm

```
Input: A method $M$ and its control-flow graph $G$
Output: A set of loop field $\mathbb{L}$={$L$|$L$ is a loop in $M$ }
Set<Loop Info> $\mathbb{L} \leftarrow$ Null;
for $i$ in InstructionsOf(M) do
    if IndexOf(SuccInstruction(i)) < IndexOf(i) then
        new $E$
        $E_{end} \leftarrow SuccInstruction(i)$
        $E_{start} \leftarrow i$
        if $E_{end}$ equal FirstInst(any $L_k$ in $\mathbb{L}$) then
            continue
        new $L$
        $BB_L \leftarrow (SuccBBs(E_{end}) \cap PrecBBs(E_{start}))$
        $L \leftarrow BB_L$
        $\mathbb{L}.add(L)$
end
return $\mathbb{L}$;
```

out lock-require instructions and the correspondingly multiple lock-release instructions of each lock-require instruction (i.e., the $Lock - ReleaseInstructionsOf$(i) function) based on the class name (when it is a class-level lock) or the variable name (when it is a object-level lock). Then, we compute the code region scope for each pair of lock-require and lock-release instructions respectively by the intersection between the successor BB set of the lock-require instruction (i.e, the $SuccBBs(BB_i)$ function) and the precursor BB set of the lock-release instruction (i.e, the $PrecBBs(BB_j)$ function) in G. BB_i and BB_j means the BB containing the instruction i or j respectively. The successor BB set of an instruction means the set of all reachable BBs from the instruction in CFG, and the precursor BB set of an instruction means the set of all BBs that can reach the instruction in CFG. Finally, we can infer a critical section's code region scope BB_C by the union of the code region scope of all pairs of lock-require and lock-release instructions.

B. LOOP IDENTIFICATION

As shown in algorithm 2, we first find out and tag all loops Ls for a target system through control-flow analysis. Specifically, we statically analyze the program to find out all loops by identifying back edges (Es) in control-flow graph. For each loop, we also figure out the set of basic blocks (i.e., BB_L) corresponding to the loop's real code region through a way as similar as the previous critical section analysis. That is, the basic block set of a loop could be computed as the intersection of the successor set of the back edge's end basic block and the precursor set of the back edge's start basic block (i.e., $BB_L \leftarrow SuccBBs(E_{end}) \cap PrecBBs(E_{start})$).

C. INNER LOOP IDENTIFICATION

We find all inner loops for each critical section, where the basic blocks of the loops are completely contained in the code region scope of the critical section. (e.g., $BB_L \subset BB_C$). As loop analysis has been done in Section III-B, The performance features of critical section with inner loop can be collected to get further analysis.

D. EXPENSIVE LOOP IDENTIFICATION

There are a wide variety of time-consuming loops in system. According to the findings in Section II-C, we aim to identify loops that satisfy the following three conditions. First, each iteration of the loop contains time-consuming operation. Second, the depth of loop is more than one (i.e., nested loop). Finally, recursion exists in the call path to time-consuming operations.

1) TIME-CONSUMING OPERATION ANALYSIS

Our goal here is to judge whether a loop contains time-consuming operations in its every iteration.

Since time-consuming operations might be conditional in a loop, we aim to identify out a loop containing non-conditional time-consuming operations or containing time-consuming operations at all the same-level of conditional branches. Note that we only consider the first-level conditional branches but not nested conditional branches in our implementation for reducing the time complexity of program analysis.

From our findings in Section II-C, we mainly focus on time-consuming operations such as I/O, RPC, network operations and explicit sleep. We search every loop's body and all its callees recursively for time-consuming operations. In the implementation of detection tool, we identify I/O, network-related and sleep operations by checking relevant function calls in Java libraries. For RPCs, we automatically tag all of RPC call functions for the target system based on its design of RPC communication component in advance. Also, users also can specify system-specific time-consuming functions as time-consuming operations for reducing the time complexity of program analysis or increasing the accuracy of time-consuming operations analysis.

2) NESTED LOOP ANALYSIS

To identify the nested loops, finding out all the inner loop is necessary. So, we look for all inner loops for each loop field in the same function, where the basic blocks of the loops are completely contained in the code region scope of the outer loop. (e.g., $BB_{L\_inner} \subset BB_{L\_outer}$). The process of inner-loop identification involves both intra- and inter- procedural analysis. Specifically, we do intra-procedural analysis by searching in the basic block set of the outer loop field for finding tagged loops. Also, for inter-procedural analysis, we analyze all instructions of the outer loop field to get function calls and recursively find tagged loops inside the callees through the depth-first search.

3) RECURSIVE CALL IDENTIFICATION

A recursive loop is a loop that recursively calls the function containing itself. From our findings in Section II-C, recursive loop could make the loop become a time-consuming source. No extra component is built to analyze recursive loop, instead, we record all the intermediate functions when we recursively search time-consuming operations and once find circle exists in the call path we mark the loop as a recursive loop.

E. FURTHER FALSE POSITIVE PRUNING

After identifying expensive loops, we consider them as performance bug candidates. These expensive loops, however, might be not really time-consuming sources to cause performance bugs. That is, an expensive loop wrapped by a critical section might release the corresponding lock of the critical section inside its loop body so that it would not hold the lock for a long time. For example, the code snippet "`obj.lock(); for(..){ ..; obj.unlock(); sleep(100ms); obj.lock(); ..} obj.unlock();`" releases the object lock *obj* through *obj.unlock*(); during the loop iteration to avoid holding the object lock all along. Note that releasing lock during loop iteration is in fact one kind of fixing methods to real-world synchronization performance bugs.

Thus, we also identify loops containing lock releases in critical sections by static analysis and exclude them out of the bug candidates for false positive pruning.

IV. IMPLEMENTATION

The goal of our detection tool is to find out potential synchronization performance bugs as many as possible. Due to the limits that exist in dynamic detection with large-scale processing related performance bugs, aiming at high code coverage, this static detection tool is a beneficial try.

We implement our detection tool using WALA [15] static analysis frameworks. WALA is a Java bytecode analysis infrastructure maintained by IBM, which is a mature tool and provides strong support for Java program static analysis in previous work [16].

To begin with our detection tool, several works have been done with WALA including (1) making class hierarchy analysis, (2) building the control flow graph of the whole package. Each node in the graph stands for a method, and our analysis is established on parsing every static single assignment(SSA) in each method.

Especially for identifying the time-consuming operation, we use specific character string to match the prefix of SSA with WALA, such as using "java.io." and "java.net." to separately check file-related and network-related Java library APIs, using "org.apache.hadoop.fs." to check Hadoop common library file-related APIs and etc. We consider all the I/O operations and sleep-related functions as expensive operations. And as mentioned in Section III-D, our analysis checks whether any such time-consuming API is called in the loop field, including the callee functions of a loop.

TABLE 3. Benchmark: target bugs and their reported application vertion. Especially, Hadoop MapReduce v1.0.0 is integrated into package `hadoop-core-1.0.0.jar`, and HDFS v0.20.205.0 is integrated into package `hadoop-core-0.20.205.0.jar`, so we use the integreted jar package to make evaluation.

Bug ID	Application	Reported Version
HB3483	HBase	v0.90.0
MR4576	Hadoop MapReduce	v1.0.0
HD2379	HDFS	v0.20.205.0
HD3990/HD4702	HDFS	v0.23.0
HD4186	HDFS	v0.23.4

V. EVALUATION

This section will illustrate the following questions:

Basic Preparation: What is the experimental environment? How can we use the application to perform the evaluation? (Section V-A)

Detection Result: What is the effect of our static detection tool? How can we understand the outputs of the detection tool? (Section V-B)

Further Measurement: What can we learn from each step of our detection? (Section V-C)

Performance Overhead: What is the performance overhead of our static detection tool? How can we improve it? (Section V-D)

A. BENCHMARK

1) EXPERIMENTAL SETUP

We performed experiments on a PC with Intel(R) Core(TM) CPU i5-4590 processor and 32GB of memory. The experiments are performed on the unchanged Ubuntu 18.04 operating system. We used Java-1.8.0_191 with argument -Xmx16G to run the bug detection tool.

2) EVALUATED APPLICATION

Our evaluation uses 3 real-world synchronization performance bug which separately comes from HDFS, Hadoop MapReduce and HBase as our Benchmarks. This group of test samples stand for distributed systems for different purposes, as seen in Table 3. And the application versions all come from

TABLE 4. Bug detection result. The foot number of "✓" means the quantity of related bugs in candidates and '-' stands for unchecked.

Application	Target version	#Candidates	Bug Check List					
			HB3483	MR4576	HD2379	HD3990	HD4186	New Bug?
HBase	0.90.0	58	✓$_2$	-	-	-	-	✓
MapReduce	1.0.0	154	-	✓	-	-	-	✓
HDFS	0.20.205.0	115	-	-	✓	-	✓	✓
	0.23.1	72	-	-	-	✓	✓$_3$	✓
	0.23.4	73	-	-	-	✓	✓$_3$	✓
	0.23.7	66	-	-	-	-	✓	✓

bug repository, in which each bug issue is committed with the corresponding application version.

We first check whether our tool can detect the original bugs above with input the corresponding application `jar` package and observe if new bugs are detected. We also record the intermediate result, to further analyze the effectiveness. Then we record the time consumption of each step to evaluate the performance of our detection tool.

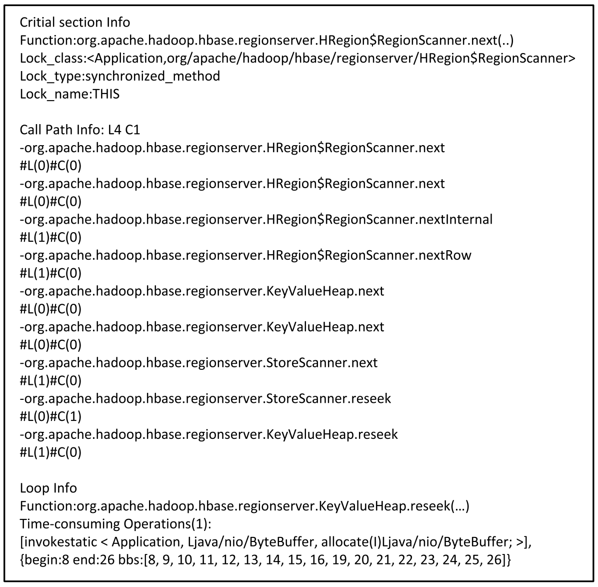
B. RESULTS

With many workload-sensitive call paths, we collect all the time-consuming loop containing critical sections and regard these critical sections as potential synchronization performance problems (i.e., the candidates). And we set the problematic critical sections in bugs as our actual targets. Overall, all the benchmarks are successfully detected by our detection tool as shown by the ✓ in Table 4. Especially, the foot numbers, right of ✓, illustrate that the numbers of candidates targeting the same bug (i.e., multiple problematic critical sections are all the root causes of the bug).

In addition, with the increasing version of HDFS, the new bug is introduced, like HD3990 in `HDFS-0.23.1.jar` and the current bugs are disappearing, like HD2379 in `HDFS-0.23.1.jar` and HD3990 in `HDFS-0.23.7.jar`. Analyzing these two bugs' fix strategies, HD2379 is fixed by using an asynchronous thread to do large process to achieve a short-time lock and HD3990 is fixed using a better algorithm to replace the time-consuming operation for speeding. Considering that once the bug is fixed (e.g, HD2379 was fixed in v0.21.0 and HD3990 was fixed in v0.23.6), it cannot be checked (i.e., our rules or findings make sense).

We randomly select several candidates to make further manual analysis, and the results show that they all has the complete factors that we think a potential synchronization performance problem matches (i.e., critical section, nested loop, time-consuming operation). Then through our manual work with the other output of the detection tool, all the candidates except benchmarks are considered as new bugs. Due to our goal is to find out all the potential synchronization performance bugs, we consider all the candidates that obey our rules or our findings as the effective bug, even though they are not found making actual damages currently. So there is no absolute error.

Figure 6 shows a part of the output, which illustrates the detail of one candidate for HBase synchronization performance bugs. As we can see, the critical section is the


```
Critial section Info
Function:org.apache.hadoop.hbase.regionserver.HRegion$RegionScanner.next(..)
Lock_class:<Application,org/apache/hadoop/hbase/regionserver/HRegion$RegionScanner>
Lock_type:synchronized_method
Lock_name:THIS

Call Path Info: L4 C1
-org.apache.hadoop.hbase.regionserver.HRegion$RegionScanner.next
#L(0)#C(0)
-org.apache.hadoop.hbase.regionserver.HRegion$RegionScanner.next
#L(0)#C(0)
-org.apache.hadoop.hbase.regionserver.HRegion$RegionScanner.nextInternal
#L(1)#C(0)
-org.apache.hadoop.hbase.regionserver.HRegion$RegionScanner.nextRow
#L(1)#C(0)
-org.apache.hadoop.hbase.regionserver.KeyValueHeap.next
#L(0)#C(0)
-org.apache.hadoop.hbase.regionserver.KeyValueHeap.next
#L(0)#C(0)
-org.apache.hadoop.hbase.regionserver.StoreScanner.next
#L(1)#C(0)
-org.apache.hadoop.hbase.regionserver.StoreScanner.reseek
#L(0)#C(1)
-org.apache.hadoop.hbase.regionserver.KeyValueHeap.reseek
#L(1)#C(0)

Loop Info
Function:org.apache.hadoop.hbase.regionserver.KeyValueHeap.reseek(...)
Time-consuming Operations(1):
[invokestatic < Application, Ljava/nio/ByteBuffer, allocate(I)Ljava/nio/ByteBuffer; >],
{begin:8 end:26 bbs:[8, 9, 10, 11, 12, 13, 14, 15, 16, 19, 20, 21, 22, 23, 24, 25, 26]}
```

FIGURE 6. One sample of the candidate HBase bugs from the outputs. The number in bracket following L means the nested loop number in each call step and the number following C means recursive call quantity.

whole method, which is attached with *synchronized* attribute. And a 4-depth nested loop exists in the critical section. The loops are *while*(*true*) end with signal value *stopRow*, *while*(*Bytes*.*equals*(*currentRow*, *peekRow*())), $while((kv = this.heap.peek())! = null)$ and $while((scanner = this.heap.poll())! = null)$. Easy to draw a conclusion that this critical section is unscalable. Besides, the innermost loop field contains one time-consuming operation (i.e. *allocate*()). This instance matches our conditions that judge the potential performance problems, and we think it has the opportunity to hold a long lock on *HRegion*$*RegionScanner* causing other operations on this component blocked. So we consider it as a potential synchronization performance bug.

C. FURTHER EVALUATION

We observe the result after each component's process from three different perspectives (i.e. # of function, critical section and loop). As shown in Table 5, 6 and 7, compared with the initial quantities, the numbers of final candidates show a reduction by 4-12 times, which greatly do good to developer to find out the effective performance problem. Besides, the final loop count is smaller than that of critical

TABLE 5. The # of candidate functions in each condition. F,C,L,Tc,R separately means function, critical section, loop, time-consuming operation and recursive call. Especially, "# CF" means the number of F containing C without recursive analysis. "# LF" means the number of F containing L without recursive analysis. "# LCF" means the number of CF containing L. "# TcLCF" means the number of LCF containing Tc. "# TcRLCF" means the number of TcLCF with recursive call in the call path.

Application package	# F	# CF	# LF	# LCF	# TcLCF	# TcRLCF
hbase-0.90.0.jar	8518	347	769	171	58	34
hadoop-core-1.0.0.jar	13942	1369	1512	684	381	215
hadoop-core-0.20.205.0.jar	13817	1354	1508	679	384	276
hadoop-hdfs-0.23.1.jar	4776	420	473	212	72	0
hadoop-hdfs-0.23.4.jar	4829	425	476	216	73	0
hadoop-hdfs-0.23.7.jar	4989	427	491	219	66	0

TABLE 6. The # of critical section in each condition. C,G,L,Tc separately means critical section, lock group, loop and time-consuming operation. Especially, "# LC" means the number of C containing L. "# TcLC" means the number of C containing L-wrapping Tc.

Application package	# C	# G	# LC	# TcLC
hbase-0.90.0.jar	382	106	179	58
hadoop-core-1.0.0.jar	1473	353	723	393
hadoop-core-0.20.205.0.jar	1458	348	718	396
hadoop-hdfs-0.23.1.jar	450	78	229	75
hadoop-hdfs-0.23.4.jar	454	78	232	76
hadoop-hdfs-0.23.7.jar	452	76	233	69

TABLE 7. The # of distinct loops from candidates in each condition. L,C,Tc separately means loop, critical section, time-consuming operation. Especially, "# CL" means the number of L in C. "# TcL" means the number of L containing Tc. "# CTcL" means the number of Tc-containing L in C.

Application package	# L	# CL	# TcL	# CTcL
hbase-0.90.0.jar	1039	272	283	70
hadoop-core-1.0.0.jar	2168	993	1178	481
hadoop-core-0.20.205.0.jar	2160	1036	900	360
hadoop-hdfs-0.23.1.jar	607	276	195	48
hadoop-hdfs-0.23.4.jar	612	276	195	48
hadoop-hdfs-0.23.7.jar	641	301	194	45

section or synchronized function. Now that they come from the same time-consuming call paths, the loop field code is reused by different critical sections, which means loop is the key factor in the large-scale process. In the future, we will study and learn more loops, which are more related to scalability. Deserve to mentioned, loop with recursive call make huge promotion to the complexity.

D. PERFORMANCE OVERHEAD

As analysis time of each stage in the whole process shown in Table 8, synchronization performance bug is reasonable for in-house analysis. It cost 9s - 120s to analysis the `jar` packages. The consuming time is more sensitive to the number of function in the program instead of the package size. Especially, we cannot find an independent package of HDFS-v0.20.205.0 in Maven repository, so we use `hadoop-core-0.20.205.0.jar` instead. Due to the hadoop-core package integrate HDFS and many other components, it can be seen that the package size of HDFS-v0.20.205.0 in Table 8 is twice bigger than that of HDFS package in other versions. Moreover, we found that almost 90% the static analysis time is actually spent for WALA to build the program dependency graph. To improve the performance, we can pre-compute the program dependency graph and store it to disk, which can greatly accelerate the second analysis with loading the existing program dependency graph.

VI. RELATED WORK

A. BUG STUDY

1) PERFORMANCE BUG STUDY

Several performance related empirical studies have been conducted in recent ten years. They proposed different methods to study the reported bugs. Some [1] of them focus on the lifecyle of a performance bug, like what is the root cause, how they are introduced, how they are exposed, and how they are fixed, then find that performance problems take long time to get diagnosed and the help from profilers is very limited; some [12], [13] of them look at how performance are noticed and reported by end users; some [2] of them compare qualitative difference between performance bugs and non-performance bugs across impact, fix and fix validation.

Besides, some researches have been done on a specified code structure, like loop. LDoctor [3] can not only use statistical analysis to identify which loop is more correlated with a performance symptom, but also point out whether or what type of inefficiency this loop contains by providing detailed root cause information and fix strategy suggestions.

And Liu *et al.* works [17] focuses on the performance problem in a specific application scenarios (i.e. smart-phone applications). Also comprehensive studies [18], [19] have been conducted for local concurrency bugs.

2) DISTRIBUTED SYSTEM BUG STUDY

Compared with studies on local machine, bug studies [14], [20] on distributed systems take up a small number. Some distributed program related works [21] only discussed few bugs. Among the studies above, no one specifically discuss the performance bug in distributed systems. So we hope our work can make some contributions to performance bug study in distributed systems and provide certain help for later research work.

Similar to our work, SyncPerf [4] analyze a few synchronization performance bugs from several distributed infrastructures, but the dynamic tool based on its findings limits the bug detection coverage. PCatch [5] make some causality anal-

TABLE 8. Time consumption detail of detection tool.

Application	Version	Package size(byte)	#Function	Time consumption in each stage (s)						Total time consumption (s)
				Wala	Lock	Loop	Inner loop	Expensive loop	Further pruning	
HBase	0.90.0	2,231,837	8518	62.312	0.968	0.105	0.037	0.468	0.411	64.301
MapReduce	1.0.0	3,740,200	13942	89.757	0.647	0.176	0.103	1.327	7.352	99.362
HDFS	0.20.205.0	3,701,573	13817	98.107	0.695	0.185	0.12	3.497	8.535	111.139
	0.23.1	1,767,238	4776	8.466	0.8	0.067	0.069	0.444	0.416	10.262
	0.23.4	1,789,089	4829	7.844	0.366	0.066	0.063	0.357	0.357	9.251
	0.23.7	1,816,199	4989	8.551	0.827	0.07	0.071	0.48	0.491	10.49

ysis with a double-digit number of distributed performance bugs and its work focus on the cascading relationship between root cause and sink. But they do not characterize the sinks, which goes against auto detection.

B. BUG DETECTION

1) PERFORMANCE BUG DETECTION

Extensive efforts have been done to detect the potential performance problems. A large number of tools have been proposed to make contributed to performance bug detection. Xu *et al.* [22] look at low-utility data structure with unbalanced costs and benefits. Jin *et al.* [1] come up with rule-based methods based on their characteristic study that is extracting rules from performance-bug patch and applying them to bug detection. Cachetor [23] uses a combination of dynamic dependence profiling and value profiling to identify and report operations that keep generating identical data values. Predator [24] targets false sharing for multithreaded applications. WAIT [25] focus on those bugs blocking the application from making progress.

There are also tools [3], [16], [26]–[28] that analyze inefficient nested loop, workload-dependent loops, unnecessary iterations and their improvement and expansion. Additionally, our work also analyzes loops, but focuses on not efficiency of loop but the workload-sensitive information (e.g. the complexity of a group of nested loop).

2) LCbug DETECTION

Specifically, LCbug detection is always hot spot in the recent decades. Identify data races [29]–[32], atomicity violations [33], [34] and order violations [35]–[37] are the main methods. Besides, a large amount of lock contention detectors and critical thread detectors have been developed but they all focus on their specific goal.

3) DCbug DETECTION

For DCbug detection, DCatch [38] learns from LCbug detection approaches and shares the same theoretical foundation with them.

ECRacer [39] and DCatch both focus on concurrency bugs in distributed systems, but their target is different. ECRacer looks at how application use underlying distributed eventual-consistency data stores, while DCatch devotes to general distributed systems, particularly basic infrastructures.

As shown in Section II, syncPerf [4] and PCatch [5] also have done much work to detect performance bug in distributed systems. But due to the dynamic method syncPerf and the artificially designated sink in PCatch, their detection coverage is greatly restricted.

C. OTHERS

There are also many other effective technologies to analyze performance problems and concurrent bugs.

1) PERFORMANCE PROBLEM DIAGNOSIS

Many diagnosis tools have been built to not only identify root causes but also give suggestions about fixing strategies when some problems are reported. These proposed tools also can be specified to diagnose certain type of performance problems. X-ray [40] aims to identify inputs or configuration entries, and nearly all of which are caused by end users. StackMine [41] focus on the callstack and use automated tools to identify certain records correlated with event handlers that cause performance loss. Yu *et al.* [42] can help the software developer understand how performance problem propagate by using processes detailed system traces to clarify the performance causality relationship between different components of the system. Coz [43] is an all-right profiling tool for multi-threaded programs, which works on unmodified Linux binaries and finds out potential optimization opportunity.

Also, many performance diagnosis techniques have been proposed to solve problems in distributed systems. They focus on different parts of the distributed systems. Some [44]–[47] look at identifying the faulty components or aim to study faulty interactions that lead to performance anomaly. Some [44], [47]–[49], [49]–[51] make full use of a quantity of run-time traces to clarify the performance dependency relationship and casual relationship then reason about performance problems and give a diagnosis report.

All tools above depend on diverse run-time traces, which is time-consuming and deviates from the our goal that the efficiency of problem diagnosis.

2) MODEL CHECKING

In distributed systems, verification and model checking are widely used.

The verification method [52], [53] usually uses some frameworks verify the distributed system implementations. Also, there is some exception, like building distributed systems with verifiable language(e.g. P# [54]). Though the result accuracy of verification can be perfect (no false positive and no negative), a long proof tremendously limits the application and propagation. MACE [55] is such a language with a suite of tools for building and model checking distributed systems. And MacePC [56] developed on MACE can detect non-deterministic performance bugs.

Diverse distributed system model checkers have been proposed, such as Demeter [57], MaceMC [58], SAMC [59], Dbug [60], and MoDist [61]. They usually intercept non-deterministic distributed system events and permute their ordering. Comparing to verification, these methods can get a better speed and some additional mistakes.

Besides, application of the Happens-Before (HB) model [38] and performance cascading (may-HB) model [5] can draw a clear map of multiple threads' relationships.

VII. CONCLUSION

Synchronization performance bugs widely exist in diverse distributed systems. As performance problems, they are imperceptible, which means they need a large workload to manifest, and generally introduced by resource contention. To understand this kind of problems better, we conduct a bug study, extract some findings. Based on our findings, we build a static synchronization performance bug detection tool. The evaluation of our detection tool verified our findings. Especially, to get a better execution and code coverage is the main reason why we choose static instead of dynamic. The work in this paper is the start that we fight against synchronization performance bugs and we hope it also can provide certain help for later research work.

REFERENCES

[1] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Beijing, China, 2012, pp. 77–88. doi: 10.1145/2254064.2254075.

[2] S. Zaman, B. Adams, and A. E. Hassan, "A qualitative study on performance bugs," in *Proc. 9th IEEE Working Conf. Mining Softw. Repositories (MSR)*. Zürich, Switzerland: IEEE Press, Jun. 2012, pp. 199–208. [Online]. Available: http://dl.acm.org/citation.cfm?id=2664446.2664477

[3] L. Song and S. Lu, "Performance diagnosis for inefficient loops," in *Proc. 39th Int. Conf. Softw. Eng. (ICSE)*. Buenos Aires, Argentina: IEEE Press, 2017, pp. 370–380. doi: 10.1109/ICSE.2017.41.

[4] M. M. Ul Alam, T. Liu, G. Zeng, and A. Muzahid, "SyncPerf: Categorizing, detecting, and diagnosing synchronization performance bugs," in *Proc. 12th Eur. Conf. Comput. Syst. (EuroSys)*, 2017, pp. 298–313. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3064176.3064186

[5] J. Li, Y. Chen, H. Liu, S. Lu, Y. Zhang, H. S. Gunawi, X. Gu, X. Lu, and D. Li, "Pcatch: Automatically detecting performance cascading bugs in cloud systems," in *Proc. 13th EuroSys Conf. (EuroSys)*, 2018, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3190508.3190552

[6] G. Chen and P. Stenstrom, "Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Salt Lake City, UT, USA, Nov. 2012, pp. 71-1–71-11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389093

[7] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu, "What change history tells us about thread synchronization," in *Proc. ACM 10th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, Bergamo, Italy, 2015, pp. 426–438. doi: 10.1145/2786805.2786815.

[8] Y. Huang, Z. Cui, L. Chen, W. Zhang, Y. Bao, and M. Chen, "HaLock: Hardware-assisted lock contention detection in multithreaded applications," in *Proc. ACM 21st Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Minneapolis, MN, USA, 2012, pp. 253–262. doi: 10.1145/2370816.2370854.

[9] X. Liu, J. Mellor-Crummey, and M. Fagan, "A new approach for performance analysis of openMP programs," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomput. (ICS)*, Eugene, OR, USA, 2013, pp. 69–80. doi: 10.1145/2464996.2465433.

[10] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, "Analyzing lock contention in multithreaded applications," in *Proc. 15th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, 2010, pp. 269–280, Bangalore, India. doi: 10.1145/1693453.1693489.

[11] T. Yu and M. Pradel, "SyncProf: Detecting, localizing, and optimizing synchronization bottlenecks," in *Proc. ACM 25th Int. Symp. Softw. Test. Anal. (ISSTA)*, Saarbrücken, Germany, 2016, pp. 389–400. doi: 10.1145/2931037.2931070.

[12] L. Song and S. Lu, "Statistical debugging for real-world performance problems," in *Proc. 2014 ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, 2014, pp. 561–578. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2660193.2660234

[13] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," in *Proc. 10th Work. Conf. Mining Softw. Repositories (MSR)*. San Francisco, CA, USA: IEEE Press, 2013, pp. 237–246. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487134

[14] H. S. Gunawi, V. Martin, A. D. Satria, M. Hao, T. Leesatapornwongsa, T. Patana-Anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, and J. F. Lukman, "What bugs live in the cloud? A study of 3000+ issues in cloud systems," in *Proc. ACM Symp. Cloud Comput. (SOCC)*, 2014, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2670979.2670986

[15] *Main Page—Walawiki*. [Online]. Available: http://wala.sourceforge.net/wiki/index.php/Main_Page

[16] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "CARAMEL: Detecting and fixing performance problems that have non-intrusive fixes," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng. (ICSE)*, May 2015, pp. 902–912. [Online]. Available: http://ieeexplore.ieee.org/document/7194636/

[17] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proc. ACM 36th Int. Conf. Softw. Eng. (ICSE)*, Hyderabad, India, 2014, pp. 1013–1024. doi: 10.1145/2568225.2568229.

[18] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun./Jul. 2010, pp. 221–230. [Online]. Available: http://ieeexplore.ieee.org/document/5544315/

[19] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proc. ACM 13th Int. Conf. Architectural Support Program. Lang. Oper. Syst. (ASPLOS)*, Seattle, WA, USA, 2008, pp. 329–339. doi: 10.1145/1346281.1346323.

[20] S. Li, H. Zhou, H. Lin, T. Xiao, H. Lin, W. Lin, and T. Xie, "A characteristic study on failures of production distributed data-parallel programs," in *Proc. Int. Conf. Softw. Eng. (ICSE)*. San Francisco, CA, USA: IEEE Press, May 2013, pp. 963–972. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486921

[21] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. Zhou, "Nondeterminism in MapReduce considered harmful? An empirical study on non-commutative aggregators in MapReduce programs," in *Proc. Companion ACM 36th Int. Conf. Softw. Eng. (ICSE)*, Hyderabad, India, 2014, pp. 44–53. doi: 10.1145/2591062.2591177.

[22] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky, "Finding low-utility data structures," in *Proc. 31st ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Toronto, ON, Canada, 2010, pp. 174–186. doi: 10.1145/1806596.1806617.

[23] K. Nguyen and G. Xu, "Cachetor: Detecting cacheable data to remove bloat," in *Proc. ACM 9th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, Saint Petersburg, Russia, 2013, pp. 268–278. doi: 10.1145/2491411.2491416.

[24] T. Liu, C. Tian, Z. Hu, and E. D. Berger, "PREDATOR: Predictive false sharing detection," in *Proc. 19th ACM SIGPLAN Symp. Princ. Practice Parallel Program. (PPoPP)*, Orlando, FL, USA, 2014, pp. 3–14. doi: 10.1145/2555243.2555244.

[25] E. Altman, M. Arnold, S. Fink, and N. Mitchell, "Performance analysis of idle programs," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, Reno/Tahoe, NV, USA, 2010, pp. 739–753. doi: 10.1145/1869459.1869519.

[26] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *Proc. IEEE Int. Conf. Softw. Eng. (ICSE)*, San Francisco, CA, USA, May 2013, pp. 562–571. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486862

[27] X. Xiao, S. Han, D. Zhang, and T. Xie, "Context-sensitive delta inference for identifying workload-dependent performance bottlenecks," in *Proc. ACM Int. Symp. Softw. Test. Anal. (ISSTA)*, 2013, p. 90. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2483760.2483784

[28] O. Olivo, I. Dillig, and C. Lin, "Static detection of asymptotic performance bugs in collection traversals," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2015, pp. 369–378. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2737924.2737966

[29] B. Kasikci, C. Zamfir, and G. Candea, "Data races vs. data race bugs: Telling the difference with portend," in *Proc. 17th Int. Conf. Architectural Support Program. Lang. Oper. Syst. (ASPLOS)*, London, U.K., 2012, pp. 185–198. doi: 10.1145/2150976.2150997.

[30] R. H. B. Netzer and B. P. Miller, "Improving the accuracy of data race detection," in *Proc. 3rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPOPP)*, Williamsburg, VA, USA, 1991, pp. 133–144. doi: 10.1145/109625.109640.

[31] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multi-threaded programs," in *Proc. 16th ACM Symp. Oper. Syst. Princ. (SOSP)*, Saint-Malo, France, 2015, pp. 27–37. doi: 10.1145/268998.266641.

[32] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy, "Detecting and surviving data races using complementary schedules," in *Proc. 23rd ACM Symp. Oper. Syst. Principles (SOSP)*, Cascais, Portugal, 2011, pp. 369–384. doi: 10.1145/2043556.2043590.

[33] C. Flanagan, C. Flanagan, and S. N. Freund, "Atomizer: A dynamic atomicity checker for multithreaded programs," in *Proc. 31st ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, Venice, Italy, 2004, pp. 256–267. doi: 10.1145/964001.964023.

[34] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting atomicity violations via access interleaving invariants," in *Proc. ACM 12th Int. Conf. Architectural Support Program. Lang. Oper. Syst. (ASPLOS)*, San Jose, CA, USA, 2006, pp. 37–48. doi: 10.1145/1168857.1168864.

[35] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin, "2ndStrike: Toward manifesting hidden concurrency typestate bugs," in *Proc. ACM 16th Int. Conf. Architectural Support Program. Lang. Oper. Syst. (ASPLOS)*, Newport Beach, CA, USA, 2011, pp. 239–250. doi: 10.1145/1950365.1950394.

[36] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng, "Do i use the wrong definition?: DeFuse: Definition-use invariants for detecting concurrency and sequential bugs," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, Reno/Tahoe, NV, USA, 2010, pp. 160–174. doi: 10.1145/1869459.1869474.

[37] W. Zhang, C. Sun, and S. Lu, "ConMem: Detecting severe concurrency bugs through an effect-oriented approach," in *Proc. ACM 15th Ed. ASPLOS Architectural Support Program. Lang. Oper. Syst. (ASPLOS)*, Pittsburgh, PA, USA, 2010, pp. 179–192. doi: 10.1145/1736020.1736041.

[38] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, "DCatch: Automatically detecting distributed concurrency bugs in cloud systems," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Oper. Syst. (ASPLOS)*, 2017, pp. 677–691. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3037697.3037735

[39] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev, "Serializability for eventual consistency: Criterion, analysis, and applications," in *Proc. 44th ACM SIGPLAN Symp. Princ. Program. Lang. (POPL)*, Paris, France, 2017, pp. 458–472. doi: 10.1145/3009837.3009895.

[40] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Proc. 10th USENIX Conf. Oper. Syst. Design Implement. (OSDI)*. Hollywood, CA, USA: USENIX Association, 2012, pp. 307–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387910

[41] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Zürich, Switzerland, 2012, pp. 145–155. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337241

[42] X. Yu, S. Han, D. Zhang, and T. Xie, "Comprehending performance from real-world execution traces: A device-driver case," in *Proc. ACM 19th Int. Conf. Architectural Support Program. Lang. Oper. Syst. (ASPLOS)*, Salt Lake City, UT, USA, 2014, pp. 193–206. doi: 10.1145/2541940.2541968.

[43] C. Curtsinger and E. D. Berger, "Coz: Finding code that counts with causal profiling," in *Proc. 25th Symp. Oper. Syst. Principles (SOSP)*, Monterey, CA, USA, 2015, pp. 184–197. doi: 10.1145/2815400.2815409.

[44] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proc. 19th ACM Symp. Oper. Syst. Principles (SOSP)*, Bolton Landing, NY, USA, 2003, pp. 74–89. doi: 10.1145/945445.945454.

[45] R. Fonseca, M. J. Freedman, and G. Porter, "Experiences with tracing causality in networked services," in *Internet Netw. Manage. Conf. Res. Enterprise Netw. (INM/WREN)*, vol. 10, 2010, p. 7.

[46] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan, "Black-box problem diagnosis in parallel file systems," in *Proc. 8th USENIX Conf. File Storage Technol. (FAST)*, San Jose, CA, USA, 2010, p. 4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855511.1855515

[47] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Principles (SOSP)*, Big Sky, MT, USA, 2009, pp. 117–132. doi: 10.1145/1629575.1629587.

[48] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale Internet services," in *Proc. 11th USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, Broomfield, CO, USA, 2014, pp. 217–231. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685066

[49] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, "Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle," in *Proc. 12th USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, Savannah, GA, USA, 2016, pp. 603–618. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026924

[50] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proc. ACM 25th Symp. Oper. Syst. Principles (SOSP)*, Monterey, CA, USA, 2018, pp. 378–393. doi: 10.1145/2815400.2815415.

[51] J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Visual, log-based causal tracing for performance debugging of MapReduce systems," in *Proc. IEEE 30th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2010, pp. 795–80. doi: 10.1109/ICDCS.2010.63.

[52] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "IronFleet: Proving practical distributed systems correct," in *Proc. ACM 25th Symp. Oper. Syst. Principles (SOSP)*, Monterey, CA, USA, 2015, pp. 1–17. doi: 10.1145/2815400.2815428.

[53] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Portland, OR, USA, 2015, pp. 357–368. doi: 10.1145/2737924.2737958.

[54] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, "P: Safe asynchronous event-driven programming," in *Proc. 34th Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Seattle, WA, USA, Jun. 2013, pp. 321–332. doi: 10.1145/2491956.2462184.

[55] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace: Language support for building distributed systems," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, San Diego, CA, USA, Jun. 2007, pp. 179–188. doi: 10.1145/1250734.1250755.

[56] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala, "Finding latent performance bugs in systems implementations," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, Santa Fe, NM, USA, Nov. 2010, pp. 17–26. doi: 10.1145/1882291.1882297.

[57] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang, "Practical software model checking via dynamic interface reduction," in *Proc. 23rd ACM Symp. Operating Syst. Princ. (SOSP)*, Cascais, Portugal, Oct. 2011, pp. 265–278. doi: 10.1145/2043556.2043582.

[58] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: Finding liveness bugs in systems code," in *Proc. 4th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*, Cambridge, MA, USA, Apr. 2007, p. 18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1973430.1973448

[59] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems," in *Proc. 11th USENIX Conf. Operating Syst. Design Implement.*, Broomfield, CO, USA, Oct. 2014, pp. 399–414. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685080

[60] J. Simsa, R. Bryant, and G. Gibson, "dBug: Systematic evaluation of distributed systems," in *Proc. 5th Int. Conf. Syst. Softw. Verification (SSV)*, Voncouver, BC, Canada, Oct. 2010, p. 3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929004.1929007

[61] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "MODIST: Transparent model checking of unmodified distributed systems," in *Proc. 6th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Boston, MA, USA, Apr. 2009, pp. 213–228. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1558992

CHEN ZHANG is currently pursuing the bachelor's degree with the College of Computer, National University of Defense Technology (NUDT). His research interests include distributed systems and computer networks.

JIAXIN LI received the Ph.D. degree in computer science and technology from the National University of Defense Technology (NUDT), in 2018, where he is currently an Associate Professor with the College of Computer. His research interests include distributed systems, and computer networks and communications.

DONGSHENG LI received the Ph.D. degree in computer science and technology from the National University of Defense Technology (NUDT), in 2005, where he is currently a Professor and a Ph.D. Supervisor with the College of Computer. His research interests include distributed systems, cloud computing, and big data processing. He received the Chinese National Excellent Doctoral Dissertation, in 2008.

XICHENG LU is currently a Professor and a Ph.D. Supervisor with the College of Computer, National University of Defense Technology (NUDT). He has been a member of the Chinese Academy of Engineering, since 1999. His research interests include parallel and distributed processing, and computer networks.