Database-Backed Program Analysis for Finding Cascading Outage Bugs in Distributed Systems

Riza O. Suminto, Shan Lu, Cindy Rubio-González*, and Haryadi S. Gunawi

University of Chicago  *University of California, Davis

Modern distributed systems (“cloud systems”) have emerged as a dominant backbone for many of today’s applications. As these systems collectively become the “cloud operating system”, users expect high dependability including performance stability and availability. Small jitters in system performance or minutes of service downtimes can have a huge impact on company and user satisfaction.

We try to improve cloud system availability by detecting and eliminating cascading outage bugs (CO bugs). CO bug is a bug that can cause simultaneous or cascading of failures to each of the individual nodes in the system, which eventually leads to a major outage. While hardware arguably is no longer a single point of failure, our large-scale studies of cloud bugs and outages reveal that CO bugs have emerged as a new class of outage-causing bugs and single point of failure in the software. We address the CO bug problem with the Cascading Outage Bugs Elimination (COBE) project. In this project, we:

1 study the anatomy of CO bugs, (2) develop CO-bug detection tools to unearth CO bugs.

1 Introduction

Modern distributed systems (“cloud systems”) have emerged as a dominant backbone for many of today’s applications. They come in different forms such as scale-out systems [30, 58], key-value stores [26, 28], computing frameworks [27, 55], synchronization [24, 39] and cluster management services [35, 68]. As these systems collectively become the “cloud operating system”, users expect high dependability including reliability and availability. They have to provision fast and stable response time, which means they need stable performance; and must be accessible anytime and anywhere, an ideal 24/7 service uptime if possible.

Unfortunately, cloud outages keep happening every year [66, 65, 66], and can easily cripple down a large number of other services [2, 15, 16]. Not only do outages hurt customers, they also cause financial and reputation damages. Minutes of service downtimes can create hundreds of thousands of dollar, not multi-million, of loss in revenue [6, 7, 11]. Company’s stock can plummet after an outage [14]. Sometimes, refunds must be given to customers as a form of apology [16]. As rivals always seek to capitalize an outage [1], millions of users can switch to another competitor, a company’s worst nightmare [10].

To improve availability of cloud-scale distributed systems, we focus on preventing downtimes of datacenter and mobile systems caused by cascading outage bugs. “No single point of failure” is the mantra for high availability. Hardware arguably is no longer a single point of failure as the philosophy of redundancies has permeated systems design. On the other hand, software redundancy such as N-version programming is deemed expensive and only adopted in mission-critical software such as in avionics. Thus, in many important systems today, software bugs are single points of failure. Some software bugs are “benign”; they might fail some subcomponents but the whole system can tolerate the partial failure. Some other bugs however can lead to outages such as configuration bugs and state-corrupting concurrency bugs, which have been analyzed extensively in literature. However, our large-scale studies of cloud bugs and outages [33, 34] reveal a new class of outage-causing bugs. In particular, there are bugs that can cause simultaneous or cascades of failures to each of the individual nodes in the system, which eventually leads to a major outage. We name them cascading outage (CO) bugs.

2 Cascading Outage Bugs

“No single point of failure” is the mantra for high availability. Hardware arguably is no longer a single point of failure as the philosophy of redundancies has permeated systems design. On the other hand, software redundancy such as N-version programming is deemed expensive and only adopted in mission-critical software such as in avionics. Thus, in many important systems today, software bugs are single points of failure.

Some software bugs are “benign”; they might fail some subcomponents but the whole system might tolerate the partial failure. Some other bugs, however, can lead to outages, bugs such as state-corrupting configuration and concurrency bugs, which have been analyzed extensively in the literature. However, our large-scale studies of cloud bugs and outages [33, 34] reveal a new class of outage-causing bugs. Specifically, there are bugs that can cause simultaneous or cascades of failures to
each of the individual nodes in the system, which eventually leads to a major outage. We name them cascading outage (CO) bugs.

To tackle this new class of bugs, we did Cascading Outage Bug Elimination (COBE) project. In this COBE project, we study the anatomy of CO bugs and develop tools to detect CO-bug patterns.

2.1 Sample Bugs

We started the COBE project by collecting samples of CO bugs from publicly accessible issue repositories of open-source distributed systems. We initially focused our search in HDFS[8] and HBASE[4] system. Later on, we also added some CO bugs from Yarn[68], Cassandra[3], Kudu[5], and some non-public issues reported by our company partner. One challenge in this bug study is how to categorize the CO bugs that we found. Categorizing CO bugs based on their root cause or symptom is very hard due to their diverse and complex nature. Therefore, we categorize them simply based on the similarity of their outage patterns. Figure 1 shows the list of CO bugs that we found in our study, grouped by their CO pattern. In the following subsections, we will explain each of CO pattern and some issues that fall into that CO patterns category.

2.1.1 Race in Master

The most frequent CO bugs pattern that we found is a race condition that is happening in the master node. We refer to this pattern as race in master. This pattern is especially prevalent in systems with Master-Worker architecture. Although it shares similarities with traditional local concurrency bugs, race in master pattern differs in the subject of the race condition, that is the message delivery timing[43]. Incoming messages to a master node can incur an order violation or atomicity violation.

hb4539 is an example of race in master pattern that caused by order violation of message. HBASE is using ZooKeeper as a synchronization service between HBASE Master (HMaster) and RegionServers, worker nodes in HBASE. When an HBASE region R1 is transitioning from OPENING to OPENED state, HMaster will spawn an OpenedRegionHandler thread to delete the ZooKeeper node (ZkNode) representing R1. But before this thread is executed, the RegionServer RS1 hosting R1 is down, triggering HMaster to reassign R1 to different RegionServer RS2. When this reassignment is finished, a second OpenedRegionHandler thread will then spawn and compete with the first OpenedRegionHandler thread to delete the ZkNode of R1. The losing thread will catch an exception for trying to delete an already deleted ZkNode, and in turn crash HMaster. This bug will not happen if the first handler thread is done executing before reassignment begin.

Another example of race in master pattern is hb4729, caused by atomicity violation of events. In the event of a region splitting, HMaster will first unassign the region by creating an ephemeral ZkNode for that region. In the middle of this splitting process, there is an incoming admin command to alter the region, that is also triggering region unassignment for that alter purpose. The second unassignment then try to re-create an already existing ZkNode and caught a NodeExistsException, which in turn trigger HMaster crash.

Master node crash often leads to a cluster-wide outage. This is why many Master-Worker architecture systems are equipped with high availability features to prevent master node becoming a single point of failures, such as having a backup master node to failover or do a new master election among active worker nodes. But since they share the same master node code, they are prone to hit the same bug again. Worse case, the failover might not happen at all because the original master is hanging instead of failing gracefully (hb12958, hb14536), the backup master also failed because of inconsistent state (hd6289, hb8519), or the race silently cause corruption that leads to outage in future (hd5425, hd5428, hd6908).

2.1.2 Hanging recovery

In this CO pattern, the system is aware of ongoing failure and attempt to do the recovery. However, the recovery procedure is buggy and causes the cluster to hang, unable to service requests. These recovery bugs may happen due to the flaw in the recovery logic itself, or due to another interleaving events that happen concurrently with the recovery process.

In hb21344, an HBASE cluster is hanging after RegionServer hosting the META region crashed. HMaster tries to recover by reassigning the META region to other RegionServer. The recovery steps involve marking ZkNode of the META region as OPENING and sending openRegion RPC to newly assigned RegionServer. Unfortunately, the openRegion RPC is also failing. After 10 times retry, the recovery procedure is rolled back, but the ZkNode state is not reverted to OFFLINE. HBASE cluster was hanging with an assumption that the META region is still in the OPENING state, even when HMaster is restarted.

Similarly, hb9721 also involves a META region recovery procedure. In the middle of recovery, the target RegionServer for META reassignment is restarting, causing it to change its name (RegionServer name components contains its start time). Because of this mismatch, the RegionServer is unable to update the ZkNode state of META region from OPENING to OPENED, and the
Table 1: CO bugs patterns. CO bugs patterns found from bug study.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Count</th>
<th>Bug ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Race in Master</td>
<td>23</td>
<td>hb19218, hb16367, hb14536, hb12958, hb9773, hb8519, hb4729, hb4539,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hd7725, hd7707, hd7225, hd6908, hd6289, hd5474, hd5428, hd5425,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hd5283, tb260, tb258, tb246, tb106, tb89, tb58</td>
</tr>
<tr>
<td>Hanging recovery</td>
<td>11</td>
<td>ca13918, hb21344, hb16138, hb14621, hb13802, hb9721, hb5918, hb3664,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hd4816, tb254, tb29</td>
</tr>
<tr>
<td>Repeated buggy recovery</td>
<td>7</td>
<td>hb14598, hb11776, hb9737, hb7515, hd9178, tb259, tb247</td>
</tr>
<tr>
<td>External library exception</td>
<td>7</td>
<td>hb17522, hb15322, hb14247, hd10609, tb301, tb291, tb275</td>
</tr>
<tr>
<td>Silent heartbeat</td>
<td>4</td>
<td>hd9293, hd9107, hd8676, hd6179</td>
</tr>
<tr>
<td>Transient network error</td>
<td>3</td>
<td>hb10272, hd8995, tb181</td>
</tr>
<tr>
<td>Race in Worker</td>
<td>3</td>
<td>hb20403, tb298, tb52</td>
</tr>
<tr>
<td>Authentication bug</td>
<td>3</td>
<td>kd2264, tb245, tb131</td>
</tr>
<tr>
<td>Abnormal message</td>
<td>3</td>
<td>hb10312, hd5483, tb287</td>
</tr>
<tr>
<td>Topology specific bug</td>
<td>2</td>
<td>hb7709, hd10320</td>
</tr>
<tr>
<td>Cascading Worker lockup</td>
<td>2</td>
<td>hb11813, hd7489</td>
</tr>
<tr>
<td>Total</td>
<td>68</td>
<td></td>
</tr>
</tbody>
</table>

2.1.3 Repeated buggy recovery

The recovery procedure in a distributed system typically works by moving the workloads of the failing node to another active node. However, if the workload is the trigger for the node to fail in the first place, the failover node will most likely hit the same failure as well. It may be because of the workload itself that is corrupted, or the same buggy code that is being run. As the same failover logic repeats, all of the nodes then become unavailable. We refer to this kind of CO pattern as repeated buggy recovery.

An example of this CO pattern is hb9737. A corrupt HFile makes its way into a region and the region becomes offline. HMaster notices when the region becomes offline and tries to assign it to different RegionServer. Upon reading the corrupt HFile, an exception is thrown and that RegionServer drops the region again. When this exception occurs, there is a bug that causes the RegionServer not to close the filesystem stream used to read the corrupt HFile. As the failover logic repeats, the region keeps bouncing between multiple RegionServers, accumulates orphaned filesystem stream, and one-by-one crashing with OutOfMemoryError.

In hb14598, a table scan operation against region containing particularly wide rows will cause RegionServer to crash with OutOfMemoryError. This lead to cascading region server death, as the RegionServer hosting the region died, opened on a new server, the client retried the scan, and the new RegionServer died as well.

2.1.4 External library exception

Distributed systems often build in layers of subsystems. For example, HBASE was built on top of HDFS as the storage layer, and incorporate ZooKeeper for internode synchronization. One system needs to interact with other systems in the different layers through the client API library. Consequently, an error or failure that is happening in the subsystem may propagate to the main system and lead to an outage. We refer to this CO pattern as external library exception pattern.

One of CO bug that falls into this pattern is hb14247. Each of HBASE RegionServers is writing their Write Ahead Log (WAL) file into HDFS. After some time, HBASE archived them into a single HDFS directory. In big clusters, because of long time-to-live of WAL or disabled replications, the number of files under WALS archive directory reaches the max-directory-items limit of HDFS (1048576 items), HDFS client library throws an exception and crash the HBASE cluster. A simple solution for this bug is to separate the old WALS into different directories according to the server name of the WAL.

In tb301, a Hadoop cluster with Azure Data Lake Storage (ADLS) backend is experiencing service outages due to an outdated ADLS SDK library being used. The problem only solved after upgrading to a newer version containing updated ADLS SDK library.

2.1.5 Silent heartbeat

In many distributed systems, nodes availability is determined by periodic signaling between nodes, referred to as heartbeat. If a heartbeat signal from an endpoint is not heard after a certain period, that endpoint is deemed unavailable by the system. Silent heartbeat is a CO pattern
where an outage happens because of a missing or delayed heartbeat signal in either sender or receiver, causing the system to falsely interpret the sender node as unavailable.

An example of silent heartbeat caused by the sender node is hd8676. A rolling upgrade of HDFS DataNodes involves cleaning up data blocks in trash directories. This cleanup is done synchronously by the same thread that is doing heartbeat signaling. In a busy cluster where the deletion rate is also high, a lot of data blocks can pile up in the DataNode trash directories. Hence, this cleaning process blocks the heartbeat and causes heartbeat expiration. HDFS NameNode losing hundreds of DataNodes after delayed upgrade finalization. The fix for this bug is to make the deletion of trash directories as asynchronous.

In hd9293, silent heartbeat happens due to delay on the receiver side. In HDFS cluster, DataNodes send heartbeat signals to both Active NameNode and Standby NameNode in a serial manner. An edit log processing by Standby NameNode holds its FSNamesystem lock for too long, causing a delay in processing incoming heartbeat. Active NameNode starts removing stale DataNodes which can not send a heartbeat to Active NameNode because they are stuck waiting for a response from Standby NameNode. A similar CO bug is also happening in hd6179 where Standby NameNode is slow to respond due to long garbage collection.

### 2.1.6 Transient network error

Distributed systems that highly available often come with a protocol to recover from network partition. When the network partition is resolved, nodes that previously unreachable are expected to automatically sync up with the rest of the cluster. However, we found some cases where the system fails to recover from a network partitions, especially if the network partition is happening intermittently. We refer to this kind of CO pattern as transient network error pattern.

Figure 1 illustrates a transient network error pattern in hd8995. When DataNode gets partitioned from NameNode for more than the heartbeat expiration time, DataNode is expected to re-register again with NameNode. Datanodes keep retrying the last RPC call and when it finally gets through, the NameNode will tell it to re-register. The DataNode is supposed to create a registration object which contains address 0.0.0.0, pass it to the NameNode which updates the address and returns it, then the DataNode saves the updated registration object for future calls. The problem is the DataNode saves off the initial registration object containing 0.0.0.0 before it receives the NameNode response. Intermittent network error happens right in this registration process and triggers an exception and left DataNode with invalid registration object containing 0.0.0.0. When the network connectivity restored, the next call to NameNode using this invalid registration object will raise UnregisteredException that in turn will signal DataNode to terminate.

Another example of transient network error pattern is hb10272. HBase client caches a connection failure to a server and any subsequent attempt to connect to the server throws a FailedServerException. If a node which hosted both of the active HMaster and ROOT/META table goes offline, the newly anointed HMaster’s initial attempt to connect to the dead RegionServer will fail with NoRouteHostException which it handles. But on the second attempt, it crashes with FailedServerException. Each of the backup masters will crash with the same error and restarting them will have the same effect. Once this happens, the cluster will remain non-operational until the node with region server is brought online, or the ZkNode containing the root region server and/or META entry from the ROOT table is deleted.

### 2.1.7 Race in Worker

In contrast with race in master, the race in worker pattern is an outage that is caused by race condition happening in the worker nodes. While we believe that message delivery timings can also contribute to race in worker, all bugs that we found so far usually stem from the use of a non-thread-safe library.

HBASE had this CO pattern in hb20403. HBASE RegionServer sometimes prefetches HFile to improve performance. The prefetching is done by multiple concurrent prefetch threads over a single input stream. Most of the time, the underlying input stream (such as DFSInputStream) is thread-safe, or has a reliable fall back mechanism in case race condition is happening. However, if the file is encrypted, CryptoInputStream will be used instead, and it is not meant to be thread-safe.

In tb52, HBase inside an HDFS Encryption Zone causes Cluster Failure under Load. HBase cannot run safely within HDFS encryption zones because of different concurrency assumptions in the HBase write-ahead log and HDFS encrypting output streams.

### 2.1.8 Authentication bug

An authentication bug pattern is a CO pattern where cluster nodes fail to communicate with each other due to authentication issues between them. In high-security setup, an additional authentication layer usually added into the distributed system. Nodes need to authenticate with each other before start communicating by exchanging identity ticket/certificate, often with the help of a trusted third-party service such as Kerberos [9]. These authentication certificates need to be updated periodically to ensure security. Failure to keep node certificates
up to date will lead to an authentication error, causing the cluster nodes unable to communicate with each other.

One example of service outage due to an authentication bug is kd2264. A bug in Apache KUDU client causes them to never re-read an updated ticket, even if their underlying ticket cache on disk has been updated with a new credential. Other services that query data from KUDU become unable to query after 30 days since the last ticket read.

In tb245, Apache Impala with TLS enabled may fail to start after upgrade. The patch for a security issue included in the new version caused a mismatch in the domain name of the certificate, which expects a fully qualified domain name (FQDN), versus the hostname used to connect.

2.1.9 Abnormal message

Cascading outage can also happen to distributed systems due to the handling of a message that is corrupt or out of order. We refer to this CO pattern as abnormal message pattern.

An example of abnormal message pattern is hd5483. NameNode expects a DataNode to only hold a single copy of any particular data block. However, there was a case where a single DataNode reporting two replicas of the same data block on two different storages. The DataNode has both storages mounted, one storage is mounted as read-write and the other storage is mounted as read-only. Because one DataNode reporting more than one replica of the same block, NameNode failed an assertion and crashed.

In tb287, Apache Sentry may crash Hive Metastore (HMS) due to abnormal notification messages. Sentry expect notifications from HMS to: have no gaps; be monotonically increasing; not have duplicates. If these assumptions are broken Sentry would be very conservative and request several full snapshots around 5 times per day. Full Snapshots are resource intensive and can take 10 or more minutes. Sentry also blocks HMS threads when it performs sync operation. If a full snapshot is being triggered, many of these sync operations timeout and leads to HMS crashing.

2.1.10 Topology specific bug

Topology specific bug pattern is a CO pattern where an outage only happens in a specific network topology. hb7709 and hd10320 both fall into this CO pattern.

In hb7709, two HBASE clusters A and B are set with Master-Master replication. In this mode, replication is sent across in both the directions, for different or same tables, i.e., both of the clusters are acting both as master and slave. A third cluster C is misconfigured to replicate to cluster A. This cause all edits originating from C will be bouncing between A and B forever. In the long run, this infinite ping-pong saturates cluster-wide network traffic.

In hd10320, a CO bug can surface if there are rack failures that end up leaving only one rack available. HDFS default block placement policy seeks to put a block replica on a different rack. But if there is only one rack available, the replication thread can get an InvalidTopologyException, which then propagated up and terminate the NameNode.

2.1.11 Cascading Worker lockup

Cascading worker lockup pattern is a CO pattern where an error or blocking event happening in a single worker node is cascading to another worker nodes, making the cluster to hang indefinitely. Unlike race in master or hanging recovery pattern that usually involve the master node, cascading worker lockup happens entirely between worker nodes.
In this chapter, we will discuss COBE, our program analysis framework to detect CO bugs pattern in distributed systems. COBE is a program analysis stack that combines several program analyses to reveal certain patterns of CO bugs. The goal of COBE analysis stack is to extract system properties such as crash paths, recovery paths, and so on, from the code and correlate between them to reveal CO pattern.

CO bugs have diverse patterns, and each of them has different characteristics. Instead of writing individual program analyses to capture each of the different CO patterns, we design COBE to detect different CO patterns in compositional manners.

Figure 2 shows the idea of compositional program analysis in COBE. At the very bottom layer, is a database of program facts. Gathering program facts can be done by extracting it from the target system binaries using static analysis or through instrumentation and dynamic execution. COBE currently focused on using static analysis to gather its program facts database. These program facts range from a list of classes, methods, class hierarchies, method invocations, and so on. At the top layer is a set of CO pattern queries. CO pattern query is a high-level query to describe a CO pattern. The top and the bottom layer is glued together by the middle layer, which is a layer of program analysis libraries.
These three layers have a resemblance with RDBMS, where the CO pattern query layer is similar to SQL query, the middle library layer is like subqueries, and the bottom program facts layer is like the database. When we execute a CO pattern query, it will lookup for a set of program analyses required to capture that pattern from the program analysis library. Each activated program analysis will then read several program facts from the bottom layer and correlate them to produce results. Similarly, the results from program analysis libraries then returned to the high-level CO pattern query for it to correlate and produce a list of bug reports.

Different CO pattern queries may depend on the same program analysis libraries. Similarly, a program analysis library might depend on or share common analyses with other program analysis libraries. When such common dependencies occur, the analysis result from the shared library is not recomputed but retained for all of its dependents.

For example, to capture CO bug with transient network error pattern, a CO pattern query can be expressed in English as “show a path where an error in network communication to master can lead to all worker nodes to crash”. To execute this query, the program facts layer must include a list of methods, call-paths, methods that represent inter-node communication, exception instructions, and so on. Given these program facts, COBE then runs a set of analyses from the program analysis library layer to reason about the program facts, such as exception propagation analysis to find if certain network error exception can lead to termination instruction (ie., System.exit()) and taint analysis to find how some value definitions propagate between methods. Finally, COBE will correlate the results from each program analysis to answer the CO query.

6 Implementation

Figure 3 shows the architecture of COBE framework implementation. COBE analyzes the target system in two phases: parsing phase and query phase. COBE parser is implemented in 11K LOC of Java, while COBE analysis queries are implemented in 2K LOC of Datalog (which consist of 100 LOC of high-level rules and 1988 LOC of analysis library rules).

In the parsing phase, COBE will parse the code and extract comprehensive facts about target systems program structures and store them into facts database. These facts ranging from the system’s class hierarchy, function call graph, control flow, basic blocks, instruction list, mappings between variable definitions and uses, and so on. This phase also includes built-in pointer analysis that required to compute the function call graph.

The second step in COBE is the query phase. In this phase, we query for a particular CO bug pattern by correlating program facts gathered from the previous phase using Datalog as our query language. All program analyses in the library layer and high-level CO pattern queries are expressed as a set of connected Datalog rules. Currently, we have implemented the analysis of race in master and transient network error CO pattern using Datalog language. We will review more about Datalog in Section 6.2.

6.1 Program Facts Extraction

In the parsing phase, COBE extracts structural information of the target system such as class hierarchies, function call graphs, and control-flow graphs. COBE leverage WALA [13] to do this initial program parsing. WALA is a mature industrial-level program analysis tool that already has versatile tools inside it such as pointer analysis, call graph generation, and control/data flow analysis that we can use out of the box. COBE then
Table 2: Program facts domain. This table lists the domain of individual attribute in program facts relations extracted by COBE parser. A domain can have type number (integer) or symbol (string). For clarity, we define Num and Sym as domains for generic numbers and symbols data.

<table>
<thead>
<tr>
<th>Domains</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_m$</td>
<td>symbol</td>
<td>Methods reference</td>
</tr>
<tr>
<td>$C_b$</td>
<td>number</td>
<td>Basic block number</td>
</tr>
<tr>
<td>$C_{eb}$</td>
<td>number</td>
<td>Exploded basic block number</td>
</tr>
<tr>
<td>$C_t$</td>
<td>symbol</td>
<td>Type/class reference</td>
</tr>
<tr>
<td>$C_f$</td>
<td>symbol</td>
<td>Field reference</td>
</tr>
<tr>
<td>$C_{ii}$</td>
<td>number</td>
<td>Instruction index</td>
</tr>
<tr>
<td>$C_v$</td>
<td>number</td>
<td>Value number</td>
</tr>
<tr>
<td>$C_s$</td>
<td>symbol</td>
<td>Method selector</td>
</tr>
<tr>
<td>$C_c$</td>
<td>symbol</td>
<td>WALA CGNode context</td>
</tr>
<tr>
<td>Num</td>
<td>number</td>
<td>Generic number</td>
</tr>
<tr>
<td>Sym</td>
<td>symbol</td>
<td>Generic symbol</td>
</tr>
</tbody>
</table>

reads the generated call graph and control/data flow information from WALA and parse it into a set of program facts relation.

Specifically, each relation is stored as tab-separated values (tsv). Each attribute (column) of the file belongs to one of the domains listed in Figure 2. These files later then feed into Datalog solver as the database for the query phase. Table 3 lists some of the resulting relations from this parsing phase. In total, COBE can extract 55 program facts relations from system binaries.

### 6.1.1 Selective Program Analysis

Full program analysis can be costly and produce too many program facts that may be unrelated to CO bugs patterns. Since we are targeting CO bugs that involve the global system states, we are not doing a whole program analysis in this phase. Instead, we only focus on analyzing namespaces that contain most of the global system states modification and coordination, namely the server-level codes. For example, in HBASE system, this server-level code resides in namespace org.apache.hadoop.hbase.master for code related to HMaster and org.apache.hadoop.hbase.regionserver for RegionServer. While in the HDFS system, we can focus on namespace org.apache.hadoop.hdfs.server for both NameNode and DataNode codes. We also add namespace that defines RPC protocols into our analysis scope, so we can capture the inter-node communication.

By default, WALA will include all main methods found in the target system as default entry points to start building the call graph. However, building a call graph just from these default entry points is not enough, especially if the target system is a distributed system. In distributed systems, many parts of the codes are not necessarily reachable directly through the main methods, such as RPC server methods and periodic task threads. RPC server methods are usually only called by RPC client or the worker nodes and not called internally by the server itself. Similarly, from main method entry points, WALA will only see the instantiation and start invocation of the thread, but not the runnable body of the thread itself. COBE provides two options to broaden WALA’s visibility over the target system code. First, COBE provides a configuration file where users can specify additional entry points by either specifying important types or method names that should be added as additional entry points. Such type can be an RPC protocol interface or abstract type of event handler threads. After WALA’s class hierarchy analysis and before call graph analysis, COBE will search for names specified in the user configuration file and inject them as additional entry points if they are found. The second option is by supplying harness code to COBE. Harness code is an additional main method code to help guide WALA to find the important entry points that are not directly reachable through existing system main methods. This harness code should contain explicit invocations to methods that should be included as entry points. It needs to be compilable, but does not need to be executable. Harness code can be written separately out of the target system code, therefore it is not intrusive. Figure 4 shows an example of a harness code that we use to analyze hb16367, containing
an explicit allocation of HMaster object.

### 6.1.2 Context Sensitivity

Another aspect that plays a role in scalability and precision of COBE parser is context-sensitivity selection. There are two popular kinds of context sensitivity that usually employed in object-oriented languages: call-site-sensitivity (k-cfa) [60] or object-sensitivity (k-obj) [54]. K-cfa uses the last $k$ call site into the method call as context elements, while k-obj uses the last $k$ allocation site of the receiver object as context elements. Both k-cfa and k-obj offer high precision in analysis. However, our experience in using any of them for distributed system yields in either exponential fact results, long parsing time, or loss in function calls. The exponential result and long parsing time are due to the combination of the target system volume and selection of number $k$. High precision context also causes WALA call graph builder to stop the call path exploration if within a procedure it can not determine the concrete type of a call site target or receiver object, causing some losses in the resulting call graph.

To get better scalability and more complete call graph, we choose a more relax type-sensitivity [63]. Type-sensitivity is almost similar to object-sensitivity, but where an object-sensitive analysis will keep an allocation site as a context element, a type-sensitive analysis will keep the type instead. COBE achieves this type-sensitivity by using WALA built-in ZeroCFA pointer analysis policy and ReceiverTypeContextSelector, a context selector that takes the receiver object type as the context element. Type-sensitivity is especially helpful in creating a complete function call graph. When WALA finds out that there is more than one possible concrete type of receiver object for a call site, it will not stop the path exploration. Instead, it will explore all the possibilities of receiver object types where that call site might go. For COBE, it is more important to receive s function call graph that reveals wider call paths but contain some ambiguity, rather than losing those call paths at all.

### 6.1.3 Pre-computed analysis

While the majority of COBE analysis resides later in the query phase, COBE also does some pre-computation analyses in the parsing phase such as dominance analysis and exception propagation. These pre-computation analyses are usually a type of intraprocedural analysis that can be done on-the-fly while reading program structure, such as dominance analysis, exception propagation analysis, IPC call analysis, and so on.

<table>
<thead>
<tr>
<th>Program facts relation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bbDef(Cm, Ch, Cv, Num)</code></td>
<td>Method Cm blockCb define value Cv</td>
</tr>
<tr>
<td><code>bbEbbTupleInt(Num, Ch, Ceb)</code></td>
<td>In method Num, blockCb contains exploded block Ceb</td>
</tr>
<tr>
<td><code>bbExit(Ch, Cm)</code></td>
<td>Cb is exit block of method Cm</td>
</tr>
<tr>
<td><code>bbUse(Cm, Ch, Cv, Num)</code></td>
<td>Method Cm blockCb use value Cv</td>
</tr>
<tr>
<td><code>bbInvoke(Cm, Ch, Cii)</code></td>
<td>Method Cm blockCb has instruction with index Cii</td>
</tr>
<tr>
<td><code>callWithContext(Cm1, Cc1, Ch1, Cm2, Cc2)</code></td>
<td>Method Cm1 context Cc1 call method Cm2 context Cc2 through block Ch1</td>
</tr>
<tr>
<td><code>classDeclareMethod(Cs, Cs)</code></td>
<td>Ct declare method Cm</td>
</tr>
<tr>
<td><code>dominateInt(Num, Chb, Cb)</code></td>
<td>Method Cm has id Num</td>
</tr>
<tr>
<td><code>ifaceSelector(Ct, Cb)</code></td>
<td>BlockCb dominateCb in method id Num</td>
</tr>
<tr>
<td><code>immediateSubclass(Ct1, Ct2)</code></td>
<td>Interface Ct define method with selector Cs</td>
</tr>
<tr>
<td><code>implement(Ct1, Ct2)</code></td>
<td>Ct2 is immediate subclass of Ct1</td>
</tr>
<tr>
<td><code>instInvoke(Cm1, Chb, Cb)</code></td>
<td>Ct2 implement interface Ct1</td>
</tr>
<tr>
<td><code>instNew(Cm, Ch, Ct)</code></td>
<td>Method Cm1 blockCb has invocation to Cm2</td>
</tr>
<tr>
<td><code>interFace(Ct)</code></td>
<td>Method Cm blockCb allocate object of type Ct</td>
</tr>
<tr>
<td><code>killerExHandling(Cm, Ceb1, Ceb2, Ceb3, Num)</code></td>
<td>Ct is an interface</td>
</tr>
<tr>
<td><code>methodSelector(Cm, Cs)</code></td>
<td>Exception thrown in block Ceb1 is handled in Ceb2 and</td>
</tr>
<tr>
<td><code>mtUnhandledInvoke(Cm1, Cc1, Cc2)</code></td>
<td>trigger abort in Ceb3</td>
</tr>
<tr>
<td><code>rpcInterFace(Ct)</code></td>
<td>Method Cm has selector Cs</td>
</tr>
<tr>
<td><code>rpcMethod(Cm)</code></td>
<td>Method Cm2 invoked in method Cm3 blockCb may</td>
</tr>
<tr>
<td><code>call site into the method call as context elements, while an object-sensitive analysis will keep an allocation site as a context element, a type-sensitive analysis uses the last K-cfa number signify attributes that correlated with each other. For example, (Cm1,Cb1) together represent a basic block number Cb within method Cm.</code></td>
<td>throw exception and not handled locally</td>
</tr>
<tr>
<td><code>Ct</code></td>
<td>Ct is an RPC interface</td>
</tr>
<tr>
<td><code>Cb</code></td>
<td>Cm is an RPC method</td>
</tr>
</tbody>
</table>

Table 3: Examples of extracted program facts. Each attribute has their own domain (see Table 2). Subscript number signify attributes that correlated with each other. For example, (Cm1,Cb1) together represent a basic block number Cb within method Cm.
public class Cobeharness extends Thread {

    @Override
    public void run() {
        try {
            ExecutorService executor = Executors.newFixedThreadPool(5);
            Configuration conf = new Configuration();
            CoordinatedStateManager cp =
                CoordinatedStateManagerFactory.getCoordinatedStateManager(conf);
            HMaster master = new HMaster(conf, cp);
            executor.execute(master);
        } catch (Exception ex) {
        }
    }

    public static void main(String[] args) throws Exception {
        Cobeharness ch = new Cobeharness();
        ch.start();
    }
}

Figure 4: Example of Harness Code. The listing above show example of COBE harness used to analyze hb16367.

6.2 Datalog-based Analysis

Given the program facts we gain from the parsing phase, we can proceed to correlate these facts and search the existence of CO bug patterns in the system. Initially, we implement this phase as a Java program, along with the parsing code. However, the sheer complexity of rules to describe a CO bug pattern makes it difficult to express the analysis query algorithm in a Java program.

In recent years, there has been a resurgence of interest in Datalog as a query language for a wide range of new applications. This include data integration [29, 32, 44], declarative networking [49, 48, 47], program analysis [23], information extraction [31, 61], network monitoring [17], security [40, 52], and cloud computing [18]. Furthermore, prior success in adopting Datalog for program analysis motivates us to use Datalog query language to express our COBE analysis.

Datalog is a declarative logic programming language that syntactically is a subset of Prolog, where a predicate is defined as a conjunction of other predicates. For example, the Datalog rule

\[ A(w, z) : \neg B(w, x), C(x, y), D(y, z). \]

says that “\( A(w, z) \) is true if \( B(w, x), C(x, y), D(y, z) \) are all true”. Variables in the predicates can be replaced with constants, which are surrounded by double quotes, or don’t-cares, which are signified by underscores. Predicates on the right side of the rules can be inverted.

Datalog is more powerful than SQL because Datalog predicates can be naturally declared as a recursive definition[67]. Take an example of this Datalog program to query all class hierarchies.

\[
\text{extends}(\text{super}, \text{sub}) : - \\
\text{immediateSubclass}(\text{super}, \text{sub}). \\
\text{extends}(\text{super}, \text{subsub}) : - \\
\text{extends}(\text{super}, \text{sub}), \\
\text{immediateSubclass}(\text{sub}, \text{subsub}).
\]

Program analyses are highly recursive in nature, making Datalog a natural fit for it.

Predicates in Datalog are divided into two classes: extensional database (EDB), the relation that is stored in the database; and intentional database (IDB), that is all relation defined by one or more rules. All program facts extracted by COBE and domain knowledge relations defined by users are EDBs, while program analysis libraries and high-level CO pattern rules are IDBs. A collection of datalog rules is also called a Datalog program.

In the current iteration of COBE, we have implemented high-level rules to detect race in master and transient network error that can lead to node crash or hang. Crash here means the node process exits due to unhandled exception or explicit call to abort instruction (ie., System.exit()). While hang means node operation or sub-component is failed and not recovered/retried, making the system unavailable (not progressing). We implement these high-level rules of CO patterns in Souffle[12], a variant of Datalog language. Figure 5 illustrates activated Datalog rules and their dependencies across COBE analyses stack for these two high-level CO pattern rules. We will describe these high-level rules to more detailed in Section 6.3 and 6.4 respectively.
6.3 Race in Master Analysis

For race in master CO pattern, the high-level idea is a race condition that happens in the master node will cause it to crash or hang. The race may be triggered by inter-node communication or interaction between the master node’s internal periodic threads. To capture this CO pattern, our Datalog analyses will build a static happen-before (HB) model of the system and find concurrent and conflicting memory access in the HB model that can lead to system crash. Two events are concurrent if there are no happen-before causality relationships between them, while conflicting means multiple access are touching the same memory location with at least one write access [45].

HB model has been thoroughly studied in the literature [37, 38, 45, 51, 57, 59]. To build our HB model, we use HB rules from DCatch[45], specifically the synchronous RPC (Rule-Mrpc), custom push-based synchronization (Rule-Mpush), synchronous multi-threaded concurrency (Rule-Tfork), and sequential program ordering (Rule-Pseq). However, DCatch HB rules cannot be directly applied in COBE static analysis settings. There are two problems that we need to address.

The first problem is how to select HB nodes and edges in static analysis settings. DCatch uses real program execution trace as the basic building block to build their HB graph. COBE, on the other hand, builds its static HB model based on program facts retrieved from the parsing phase. It will use the HB rules to guide which functions should be added to the HB graph. The HB edges then applied between them based on the logical relationship defined by the HB rules, the call graph, and the class hierarchy relationship. The result from this process is a
static HB graph that almost looks like a call graph, but with most of the function call unrelated to the HB rules left out. Each vertex in our HB graph is a program point represented as a tuple \((m, b, h)\), where \(m\) is a \textit{call-graph node} from WALA (which is pair of a method name and its type-context), \(b\) is a basic block number, and \(h\) is the type of HB node. The edges are the HB relationship between two program point, saying that program point \((m_1, b_1, h_1)\) happen before program point \((m_2, b_2, h_2)\).

Memory accesses are also represented as HB nodes. However, this information needs to be specified by COBE user. User must specify which field member need to be checked for write and read access or, in case of global data structure, which method is used to write and read.

Given the HB rules, we first search for all related methods than can be applied to the rules. For example, given a \textit{Synchronous RPC (Rule-\textit{M}^{\text{rpc}})} HB rule, we will search all method implementation of an RPC interface and all methods that contain call-site to that RPC interface. The basic block containing that RPC call-site, along with both entry and exit blocks of the RPC implementation, then taken as HB node. We then use the definition of the HB rule to properly add HB edges between them. For synchronization rules that do not share an interface such as the case in \textit{Custom push-based synchronization protocol (Rule-\textit{M}^{\text{push}})}, COBE allows users to manually specify the pair of correlated methods that represent \textit{Update} and \textit{Pushed}.

However, this resulting static HB graph is not yet enough to reflect concurrencies that happen in the target system. Because of our selection of \textit{type-sensitivity} in the parsing phase, different call paths into a function are not distinguished and will be collapsed into the same node. Different chains of events should be represented in different HB subgraphs, and not collapsed into single subgraph. To solve this second problem, COBE will do another pass over the static HB graph to insert a second call-site sensitive context into HB nodes that we refer to as \textit{path-context}. Path-context of an HB node is implemented as a list of HB nodes from previous HB rules that lead to it. When two HB nodes are logically connected by an HB rule, the path-context of the predecessor HB node is copied as path-context of the successor HB node. Additionally, if the HB rule is either of \textit{Create}(r, n_1) \overset{M^{\text{create}}}{\longrightarrow} \text{Begin}(r, n_2), \text{Create}(t) \overset{\text{fork}}{\longrightarrow} \text{Begin}(t), \text{or} \text{Update}(s, n_1) \overset{M^{\text{push}}}{\longrightarrow} \text{Pushed}(s, n_2)\), the id of successor HB node is prepended into path-context of the successor HB node. If two HB nodes with different path-contexts have happen-before relationships to the same successor HB node, the successor HB node will be duplicated for that two different path. The addition of path-context also helps remove cycles (ie., recursive call) from the HB graph. When inserting the path-context to the HB graph, we check if the destination node already has a path-context inserted. If it does, and it shares the same path-context with the origin node, then we will not connect them with an HB edge, as it will cause a cycle. After this path-context insertion, the HB nodes will be a tuple of \(((m, b, h), px)\), with the addition of \(px\) as the path-context.

From the final static HB graph, we search for all pairs of memory access nodes \((q, r)\) that do not have happen-before relationships between them (concurrent) and at least one of them is write access (conflicting). To further prune benign pairs from the harmful pairs, COBE will do another filtering to only report pairs where either \(q\) or \(r\) may reach \text{failure instructions}, such as an invocation of abort or exit function (e.g., System.exit), through exception throwing or implicit flow.

Note that these \((q, r)\) pairs may be duplicated with each other. Two different pairs \((q_1, r_1)\) and \((q_2, r_2)\) might be differentiated by their different path-context, but \(q_1\) and \(q_2\) might represent the same program point, as well as both \(r_1\) and \(r_2\). To remove this duplication, COBE will do further reduction by stripping the context out of \((q, r)\) pairs into a smaller set of unique program point pairs \((s, t)\). This list of \((s, t)\) pairs is what COBE report as bugs for \textit{race in master} CO pattern. Figure 6 shows the high-level rule to detect this pattern.

### 6.4 Transient Network Error Analysis

For \textit{transient network error} CO patterns, we take two high-level insights. First, \textit{an invalid value is obtained during a network communication error}. Second, the \textit{invalid value is propagated further to the next operation that will, in turn, trigger a system crash or hang}. Values can be local variables or class field members. For this pattern analysis, we will focus on class field members.

For the first insight, our first intuition is to do inter-procedural, inter-node, taint analysis to ask what is the possible value definition assigned to a class field member in a certain basic block of a method when there is no conditional path taken. Specifically, we do a traversing over the control-flow graph of a method. If in the control-flow graph a class field is assigned twice in succession, then querying a possible value of that field at any basic block after the second assignment will return only value definition from the second assignment. But if there are two possible assignments in two different conditional branches, querying a possible value at the end of the two branch will return both value definition as possible values at the queried block.

If a value definition was obtained from a method invocation, we continue our taint analysis to the originating method. If a value definition is passed to the next method invocation, we also continue our taint analysis to that
next method invocation. For inter-node message communication, we continue the taint analysis by analyzing RPC interface calls and permuting the possible destination RPC implementations.

Along with this inter-procedural, inter-node, taint analysis, we also permute one RPC call error. Our analysis notes if there is any possible field value that retained or overridden when a particular RPC method call caught a network error. We do this taint analysis for each different RPC method, where we assume a single network error for RPC call to that method.

Now, for the second insight, how do we define a value as invalid? And how do we know if that invalid value will trigger a system crash? This assertion can be different between the systems and protocols being tested. For some systems, it might be enough to verify whether the value definition is obtained from the local node or remote node [62]. For other systems or protocols, a more precise assertion might be needed.

In our initial implementation, we create an assertion to target the DataNode registration protocol in HDFS. HDFS NameNode maintains several global data structures. One of them is DataNode mapping. A valid DataNode registration object must be updated by NameNode, added to this DataNode mapping, and then returned to DataNode for further communication with NameNode. If NameNode can not find a registration object in this data structure, it will raise a remote exception back to DataNode as a response to abort the DataNode. We write Datalog rules to express this DataNode map, its put method and get method. We extend our assertion to verify that a registration object is invalid if it does not reach the put method in NameNode but later on reach the get method through the next RPC call.

COBE will report the invalid value definition, the location where it is first defined, and the failed RPC method that triggers this invalid value definition as one bug report. Figure 7 shows the high-level rule for this pattern.

7 Evaluation

7.1 Methodology

7.1.1 Benchmarks

We evaluated COBE on five problems reported by users in HBASE and HDFS system. HBASE is a distributed NoSQL database, while HDFS is a distributed file system. The description of these five bugs can be seen in

Figure 6: High level rules for race in master CO pattern. The bug report (s,t) pair maps into ((t1,m1,b1),(t2,m2,b2)) from relation raceInMaster.
Table 4.
We obtained these bug benchmarks from our CO bug study listed in Section 2.1. Since COBE starts with implementations of race in master and transient network error high-level rules, we took five representative bugs from these two categories. Four of the bugs were taken from race in master category, two of which have order violation (OV) root cause, while the other two have atomicity violation (AV) root cause. For the last bug, we took an HDFS bug that falls into the transient network error category.

### 7.1.2 Evaluation metric
We will evaluate COBE by the number of reported bugs and its ability to find the true positive case that was reported by the original issues. We will also review the size of extracted program facts, time to run parsing and query. Both parsing and query run time are obtained by averaging measurement from 5 runs. For race in master analysis, we will also discuss the correlation between the reported bug and the resulting static HB graph.

### 7.1.3 Experiment settings
For all bug benchmarks, we configure COBE to focus its analysis only at server level codes. For HBASE, we focus the analysis on package 
\[
\text{o.a.h.hbase.master, o.a.h.hbase.regionserver, o.a.h.hbase.zookeeper, o.a.h.hbase.ipc, and o.a.h.hbase.executor,}
\]
while in HDFS, we focus our analysis on package 
\[
\text{o.a.h.hdfs.server, o.a.h.hdfs.protocolPB (o.a.h stands for org.apache.hadoop).}
\]
In each bug analysis, all main methods found in the focused packages are added as entry points in the program parsing phase. For HBASE bugs other than hb16367, we also add all methods that implement RPC interface between HMaster to RegionServer and HBASE Client to HMaster (\text{o.a.h.ipc.HRegionInterface} and \text{o.a.h.ipc.HMasterInterface} respectively), ZooKeeper listener interface (\text{o.a.h.ZooKeeperListener}), and methods extending EventHandler abstract class \text{o.a.h.executor.EventHandler}. For hb16367, since the bug only involves concurrency between HMaster internal thread and does not involve inter-node messaging, we do not add those communication endpoints as additional entry points. Instead, we add harness code from Figure 4 as our additional entry point. For hb14536, both harness code and additional inter-node messaging entry points. Similarly, for HDFS system (hd8995), we add RPC interface between DataNode to NameNode (\text{o.a.h.hdfs.server.protocol.DatanodeProtocol}) as additional entry points.

Besides specifying the analysis scope and important communication protocols, COBE also requires users to specify which global states to focus on the analyses. For example, HBASE global states are stored in ZooKeeper, so we tag methods that do creation, deletion, or update of the ZooKeeper node. COBE users currently need to do this step manually by listing methods that represent global state modification in a user-defined program facts table. This step can be automated in the future as shown by some recent work[50].

For hb4539, hb4729, and hb16367 we use the same assertion to capture the race in master pattern as we explain in Section 6.3. However, for hb14536, since it is
Table 4: Benchmark bugs.

<table>
<thead>
<tr>
<th>Bug ID</th>
<th>Description</th>
<th>CO Pattern</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>hb4539</td>
<td>event handler race to delete ZK node</td>
<td>Race in Master (OV)</td>
<td>Master crash</td>
</tr>
<tr>
<td>hb4729</td>
<td>event handler race to create ZK node</td>
<td>Race in Master (AV)</td>
<td>Master crash</td>
</tr>
<tr>
<td>hb16367</td>
<td>read-write race in HMaster init</td>
<td>Race in Master (OV)</td>
<td>Master crash</td>
</tr>
<tr>
<td>hb14536</td>
<td>read-write race in META recovery</td>
<td>Race in Master (AV)</td>
<td>Master hang</td>
</tr>
<tr>
<td>hd8995</td>
<td>invalid ID reported to NameNode</td>
<td>Transient network error</td>
<td>Worker crash</td>
</tr>
</tbody>
</table>

7.2 Bug detection result

Table 5 shows the result of our experiment. In all bug benchmarks, COBE was able to capture the true positive case that was reported in the original issue. Analysis for transient network error pattern in hd8995 can show that in method BPServiceActor:register(), an invalid local variable definition was saved off to field BPOfferService bpRegistration when a network error should be occurred in RPC call DatanodeProtocol:registerDatanode().

Analysis for hb4539, hb4729, and hb14536 for race in master pattern reveals more than one bug report (pairs of (s,t)). For bugs other than what was reported in the original issues, we have not yet been able to claim whether all of them are false positive or some of them are indeed a true positive that was unknown before.

Ideally, we should able to verify these bug reports by observing the resulting static HB graph, because the HB graph can show us the chain of events that need to happen to lead to the bug. However, the large size of our resulting static HB graph hinders us from doing so. The last two rows of Table 5 shows the number of (q,r) pairs before reduced into bug candidate (r,s), and the size of resulting static HB graph in terms of the number of nodes and edges. A large number in these two measures compared with the number of bug candidates indicate that our static HB graph tends to repeat some of the HB subgraphs several times.

There are number of reasons why this repetition happens in our static HB graph. The first reason is because of our path-context addition to the HB graph. Our main goal in adding this second context is to differentiate happen-before relation to the same method but coming from different call path origins. In the conventional k-cfa sensitivity, only the last k call-site is saved for distinction comparison. But in our case, we don’t limit the k yet. Our call-site context is able to achieve maximum distinction, but with the cost of growing the HB graph deep. The second reason is we have not added logic in our HB graph building to verify whether a certain path is reachable control-flow wise, considering the path taken in the previous steps. This causes our HB graph to grow wide, as it considers all path from one point of HB node to be a valid-possible call path. Employing some path-refutation algorithm [21] might help us reduce the width of the HB graph.

In our current iteration, COBE’s static HB graph is more observable and helpful when we exclude some noise constructs from analyses. Figure 8 is a minimal static graph to reveal hb4539 from a rerun of the same analysis, but excluding event handler threads and RPC protocols that do not directly involve with RegionServer failover. The left subgraph is the chain of events when the RegionServer signal region has opened, and the right subgraph is the chain of event triggered by that RegionServer terminating shortly after the signaling region opened. The two yellow nodes with a bold red border are the pair of conflicting memory access that can crash the master node. If the bold node in the right subgraph happens before the left bold node, the HMaster will crash.

7.3 Performance result

Table 5 also shows COBE performance for parsing and query time. In terms of parsing time, the speed of parsing is highly dependent on the volume of the program being parsed. For HBASE system, the parsing phase for 5 packages is quite fast, around 30 seconds. For HDFS, the size of programs under our 2 package selection is quite high. Therefore, the parsing time and the extracted program facts relations are higher than HBASE.

In terms of query time, the performance is highly dependent on how efficient we write the Datalog program and whether we run it in interpreter or compiler mode. Souffle support running queries in either interpreter or compiler mode. In interpreter mode, Souffle translates the Datalog program to a RAM program and executes the RAM program on-the-fly. While in compiler mode, Souffle will compile the Datalog program into C++. For computationally intensive Datalog programs, the inter-
Figure 8: **Minimal static HB graph to reveal hb4539.** Both yellow and cyan node is a memory access to the same location. The yellow nodes can reach failure instructions while the cyan node is not. The two yellow nodes with bold red border is the true positive case reported by the original issue.
pretation is slower than the compilation into C++. However, the interpreter has no costs for compiling a RAM program to C++ and invoking the C++ compiler, which is expensive for large Datalog programs.

For race in master analysis, we are getting relatively fast query performance partly due to the static HB graph abstraction that reduces numbers of program facts that we need to be correlated. On the opposite, for transient network error analysis, our queries are not optimal. We are doing taint analysis for all field members, traversing all basic blocks in every method along the entire call graph. Running transient network error analysis in Souffle interactive mode does not finish after more than 30 minutes. The larger amount of program facts to analyze also makes the speed worse. Therefore we switch to compiler mode for hd8995 and gain a much faster performance compared to interactive mode. On the contrary, we are not able to run transient network error analysis in compiled mode due to compilation error by Souffle in our experiment environment.

In terms of extracted program facts, we get tens to hundreds of MB of data. The large size of data most likely happen due to duplicate strings. For example, \( Cm \) domains are encoded as a full method reference string. For relation that contain an attribute with domain \( Cm \), many of these method reference strings will be repeated. One technique that we do to reduce these repeated strings by making a dictionary relation \( dictCm \) that maps integer id with a unique method reference, as shown in Table 3. Other relation having lots of rows such as \( bbEbbTupleInt \) and \( dominateInt \) may refer to method reference through their id. However, we let other relations to keep the \( Cm \) domain attributes for ease of debugging.

### Table 5: Evaluation result. The table show result statistics for each bug benchmark. The number of bugs reported does not distinguish between the true positive and false positive cases. However, all true positive cases reported by the original issues were successfully found by COBE. For \( hd8995 \) query time, the Datalog program was run in compiled mode, while the others was run in interpreter mode.

<table>
<thead>
<tr>
<th></th>
<th>hb4539</th>
<th>hb4729</th>
<th>hb16367</th>
<th>hb14536</th>
<th>hd8995</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugs reported</td>
<td>15</td>
<td>7</td>
<td>1</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>True positive case found</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Program facts size (MB)</td>
<td>106</td>
<td>99</td>
<td>67</td>
<td>67</td>
<td>313</td>
</tr>
<tr>
<td>Parsing time (s)</td>
<td>24.4</td>
<td>25.8</td>
<td>30.2</td>
<td>36</td>
<td>41.4</td>
</tr>
<tr>
<td>Query time (s)</td>
<td>18.9</td>
<td>24.4</td>
<td>10.6</td>
<td>68.1</td>
<td>148.8*</td>
</tr>
<tr>
<td>(#(q,r)) pairs</td>
<td>3818</td>
<td>90</td>
<td>3</td>
<td>2520</td>
<td>67</td>
</tr>
<tr>
<td>HB graph size (nodes/edges)</td>
<td>894/1181</td>
<td>367/438</td>
<td>57/75</td>
<td>652/1280</td>
<td>-</td>
</tr>
</tbody>
</table>

82 Related Work

We now discuss some works related to COBE. We contrast them between specific analysis framework (dynamic and static), specific Datalog framework, and general Datalog framework. We also contrast their target system between distributed systems and other specific systems. Table 6 show this comparison between COBE and other related works.

Specific Framework: FindBugs[36] is one of the popular static analysis tools to capture bugs in Java. It has hundreds of checks to capture many bug patterns including multithreaded correctness but mostly limited for intraprocedural, single-machine applications. Both

9 Discussions

Scalable program analysis (e.g., [70, 69, 41, 42]) has been an important area of research in the past few years. The size and complexity of software applications continues to grow and computer resources often become a bottleneck when analyzing large applications such as distributed systems.

The goal behind scalable program analysis is to enable software developers to run program analysis in commodity hardware by making program analysis more efficient in terms of runtime and/or memory usage. If a program analysis does not require large amounts of memory and time to run, then developers are more likely to be able to run these analyses in their work machines, including small laptops. Such analysis scalability would also allow developers to run analyses more often, which would be highly beneficial for bug finding given the fast-paced changing nature of software.

As with past work, the use of a database-backed program analysis in COBE allows to reduce memory requirements. Furthermore, COBE’s compositional architecture enables the reuse of program facts and analyses when searching for different bug patterns. We believe this will be a key to achieve scalability when expanding our approach to support all the patterns described in Section 2. In the future, it will be interesting to explore incremental analysis for COBE so that an application does not have to be analyzed from scratch in the presence of small code changes.
Table 6: Related Work (COBE). The table categorizes works that relate to failure analysis, race analysis, and Datalog in the space of program analysis grouped by either targeting specific system or distributed system.

<table>
<thead>
<tr>
<th>Specific System</th>
<th>Specific Framework</th>
<th>Specific Datalog</th>
<th>General Datalog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sierra[38]</td>
<td>RacerD[22]</td>
<td>NDlog[47]</td>
<td>EC-Diff[64]</td>
</tr>
<tr>
<td>FindBugs[36]</td>
<td></td>
<td>PQL[53]</td>
<td>Chord[47]</td>
</tr>
<tr>
<td>RacerD[22]</td>
<td></td>
<td>NDlog[47]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PQL[53]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed System</td>
<td>DCatch[45], FCatch[46]</td>
<td>Bloom[19], Dedalus[20]</td>
<td>COBE</td>
</tr>
</tbody>
</table>

RacerD[22] and Sierra[38] are static analysis framework targeting race bugs in the Android system. Sierra uses a static HB graph, similar to what COBE use for race in master analysis. While Sierra’s static HB graph model is based on Android message passing and event handling, COBE model its static HB graph based on inter-node communication such as RPC. RacerD is based on Infer[25], thus inherits Infer’s compositional analysis. RacerD compositional granularity is procedure summaries, while COBE composition is program facts and Datalog rules. DCatch[45] and FCatch[46] are both dynamic analysis framework for distributed systems. DCatch targets distributed concurrency bugs, while FCatch target time of fault (TOF) bugs. Both DCatch and FCatch works by instrumenting the target system to produce traces and analyze that trace to find bugs. COBE uses some ideas from DCatch to build static HB graph, but does not relies on real execution traces.

**Specific Datalog:** Datalog has been adopted as the foundation for applied, domain-specific languages in a wide variety of areas. *Network Datalog (NDlog)[47]* is a language for declarative network specifications. It enables declarative specification and deployment of distributed protocols and algorithms via distributed recursive queries over network graphs. *PQL[53]* is a query language that translates into Datalog, aimed to capture errors and security flaws such as SQL injection vulnerabilities. *Dedalus[20]* is a declarative language that enables a specification of rich distributed system concepts. Dedalus reduces to a subset of Datalog with negation, aggregate functions, successor and choice, and adds an explicit notion of logical time to the language. *Bloom[19]* is a declarative language to build a program that can runs naturally on distributed machines. Bloom programs are bundles of declarative statements about collections of facts, similar to Datalog.

**General Datalog:** EC-Diff[64] is a static analysis for computing synchronization differences of two programs. It use Datalog to compute differentiating data-flow edges in large multithreaded C programs. *Chord[56]* is a static race detection analysis tool for multithreaded Java programs. Chord detects race in four stages where all four of them are expressed in Datalog language based on bddbddb[71]. Unlike COBE, both EC-Diff and Chord are targeting single-machine applications.

## 10 Conclusion

We revealed a new class of outage-causing bugs in distributed systems that we refer to as cascading outage (CO) bugs. Specifically, CO bugs are bugs that can cause simultaneous or cascades of failures to each of the individual nodes in the system, which eventually leads to a major outage. We do CO bugs study by collecting CO bugs reported in publicly accessible issue repositories of open-source distributed systems and group them by their CO pattern. We presented COBE, static program analysis framework to detect CO bugs pattern in distributed systems. We have implemented COBE prototype to detect race in master and transient network error CO pattern.
References

[10] Messaging app Telegram added 5m new users the day after WhatsApp outage (link), February 24, 2014.


