Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems using the Event Chaining Approach

Yongle Zhang
University of Toronto
yongle.zhang@mail.utoronto.ca

Serguei Makarov
University of Toronto
serhei.makarov@mail.utoronto.ca

Xiang Ren
University of Toronto
jenny.ren@mail.utoronto.ca

David Lion
University of Toronto
david.lion@mail.utoronto.ca

Ding Yuan
University of Toronto
yuan@ece.utoronto.ca

ABSTRACT

Complex and unforeseen failures in distributed systems must be diagnosed and replicated in a development environment so that developers can understand the underlying problem and verify the resolution. System logs often form the only source of diagnostic information, and developers reconstruct a failure using manual guesswork. This is an unpredictable and time-consuming process which can lead to costly service outages while a failure is repaired.

This paper describes Pensieve, a tool capable of reconstructing near-minimal failure reproduction steps from log files and system bytecode, without human involvement. Unlike existing solutions that use symbolic execution to search for the entire path leading to the failure, Pensieve is based on the *Partial Trace Observation*, which states that programmers do not simulate the entire execution to understand the failure, but follow a combination of control and data dependencies to reconstruct a simplified trace that only contains events that are likely to be relevant to the failure. Pensieve follows a set of carefully designed rules to infer a chain of causally dependent events leading to the failure symptom while aggressively skipping unrelated code paths to avoid the path-explosion overheads of symbolic execution models.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; • **Software and its engineering** → **Software testing and debugging**;

KEYWORDS

Failure reproduction, distributed systems, log, debugging

ACM Reference Format:
Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems using the Event Chaining Approach. In *Proceedings of SOSP '17*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3132747.3132768

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10.
https://doi.org/10.1145/3132747.3132768

1 INTRODUCTION

Production distributed systems inevitably experience failures – whether due to software bugs, human errors (e.g., misconfiguration), or hardware faults. When a failure occurs, it is the vendor's top priority to diagnose it so the system can recover. A 2013 study has shown that half of software development time is spent on debugging [4].

Reproducing the failure is often the most critical step in postmortem debugging, and a prerequisite to thoroughly understanding the failure. Many of the most commonly used debugging techniques, including interactive debuggers, "printf debugging", and delta debugging [29] assume that the failure has already been reproduced. In addition, verifying the resolution (e.g., a patch or configuration workaround) also requires the failure to be reproduced. Large software projects often require developers to provide a test case that reproduces the failure for regression testing. Therefore failure reproduction is typically the first and the last step in postmortem debugging. Indeed, a prior study surveying 466 developers found that developers consider steps to reproduce a failure to be the most useful information in a bug report [3].

Despite its importance, failure reproduction remains the most time-consuming step in postmortem debugging. We thoroughly analyzed the life cycle of 30 randomly sampled bug reports from the production distributed systems HDFS, HBase, and ZooKeeper, and found that developers spend a vast majority of the resolution time (69%) on reproducing the failure. Failures in distributed systems often have a long

and complex manifestation involving multiple causally related user input events [26]. The underlying executions span across multi-threaded components with complex (typically asynchronous) communication mechanisms. Moreover, due to performance and privacy concerns, production distributed systems only record limited information for postmortem analysis, typically in the form of log files.

Unfortunately, existing solutions for speeding up failure reproduction are limited. Intrusive approaches to automatic failure reproduction, such as deterministic replay [1, 2, 7, 9, 10, 15, 17–20, 22], can faithfully replay the failure execution. However, they find limited deployment in production systems as vendors are concerned with the performance overheads and changes introduced to the production environment. ESD [28] and SherLog [27] attempt to reproduce a failure using only coredumps and log files, respectively, as their source of evidence, combined with static analysis on the program's code. Both use a variant of symbolic execution [5] to search for an execution path that leads to the target symptom. ESD guides the search towards intermediate goals inferred through static analysis, while SherLog limits its analysis to a subset of the program. In general, the paths inferred by symbolic execution are precise, and include every instruction along the path. However, this approach does not scale to complete executions of large, complex systems because it requires forking the analysis at all branch instructions regardless of their relevance to the failure execution.

This paper presents Pensieve, a tool for automatically reproducing failures from production distributed systems. Given log files output by the failure execution, the system's bytecode, a list of supported user commands, and a description of the symptoms associated with the failure (typically a user-selected subset of error messages in the log files), Pensieve outputs a sequence of user commands, packaged as a unit test, that can reliably reproduce the failure.

Pensieve's design is based on the *Partial Trace Observation*:

Programmers almost never debug a failure by reconstructing its complete execution path; instead, they skip a vast majority of the code paths by focusing on instructions that are likely to be causally relevant to the failure.

More specifically, programmers typically apply the following inference rules during postmortem debugging:

- A failure event can be explained by searching the codebase for plausible causes. For example, to explain why a variable has a particular value, programmers typically consider where it is defined, which could be in another thread or even component. The programmer 'jumps' directly from the failure event to the cause without reconstructing the intermediate control-flow path.
- Log printing statements provide a useful clue when there are multiple possible explanations of an event. For example, when a variable value can be defined at multiple program locations, programmers often check if a log was printed close to one of these locations.
- For many loops, only a subset of their iterations have effects that are necessary for the failure to occur. Simulating the entire loop (as done by symbolic execution) is typically not relevant to debugging.

Based on these observations, Pensieve implements *event chaining*, a static analysis strategy that substantially differs from prior methods. Pensieve produces a partial trace describing a set of events that occurred in the failure execution. It does so by iteratively analyzing the control and data dependencies that are most likely to be relevant to the failure, until the analysis reaches external API calls (user commands), while discarding likely-irrelevant dependencies and aggressively skipping the rest of the code path. The resulting set of API calls is used to generate a unit test. Pensieve searches the codebase globally for causes, mimicking the 'jumping' strategy used by programmers trying to understand a failure (as discussed in § 3.2). For example, Pensieve finds where a variable of interest is modified, but does not verify the existence of a control flow path connecting the modification and use points. This often allows Pensieve to bypass complex network transfer code and find a root cause that occurs in another thread or component, because coding practices for distributed systems favor using uniform object types at the two ends of network communications.

Inferring a dependency trace that captures dynamic execution information requires Pensieve to distinguish different invocations of the same method and different iterations of a loop. Pensieve solves this by assigning symbolic invocation and iteration IDs to the inferred events. When multiple alternative dependencies exist, Pensieve heavily relies on log printing statements as clues to pick which one is more likely to have occurred. Finally, it classifies the dependencies of each loop on the causal path. For the majority of loops, it only needs to analyze a single iteration.

Pensieve's aggressive skipping of code paths unavoidably sacrifices accuracy, and can lead to some infeasible or inaccurate dependencies. We correct inaccuracies with a dynamic verification phase that provides feedback to refine Pensieve's analysis. The unit test generated by Pensieve is executed in a controlled JVM that observes whether the inferred dependencies and their partial orders are respected. If not, Pensieve identifies the point of divergence, determines the condition to avoid the divergence, and restarts the static analysis phase with this condition as an additional requirement.

Pensieve has the following attributes:

- *Scalable to real distributed systems.* The scalability of Pensieve is not limited by the code size or the number of execution paths, but by the number of causally dependent

events (which is orders of magnitude smaller), allowing it to scale on complex distributed system codebases.
- *Reproduces non-deterministic failures.* Pensieve's analysis captures data dependencies among multiple threads or processes, and the dynamic verification phase enforces the timing constraints to reliably reproduce the failure.
- *Simplifies reproduction steps.* Pensieve's design to selectively infer the most important causal dependencies leads to a near-minimal sequence of reproduction steps as it only seeks to satisfy those most important conditions.

We evaluate Pensieve on 18 randomly sampled real failures reported to HDFS, HBase, Cassandra, and ZooKeeper. Pensieve can automatically reproduce 13 (72%) of them and provides a useful partial diagnosis for two more, giving a helpful result for 83% of the failures overall. The event chain from the failure symptom to the user commands consists of an average of 105 events, which is orders of magnitude smaller compared to the path constraint size for an implementation of SherLog's algorithm (tens of millions of AST nodes that the SMT-solver uses to represent the path conditions) and the complexity of the actual failure execution path (tens of millions of branch instructions).

This paper makes the following contributions. First, we propose an algorithm, event chaining, that uses the *Partial Trace Observation* to reconstruct a simplified partial trace of a failure execution. Second, using this algorithm, we build the first tool, Pensieve, that is capable of non-intrusively reproducing failures from complex production distributed systems. Finally, we propose a simple dynamic refinement mechanism that verifies and refines the generated reproduction steps.

Pensieve also has the following limitations. First, its analysis is unsound and incomplete; the dynamic refinement phase is designed to correct this inaccuracy. In addition, it must be possible to exercise the system's functionality using simple API calls, and to characterize the failure using a subset of the error log messages or stack traces output by the system. These limitations are discussed in detail in § 6.

The rest of the paper is organized as follows. § 2 studies the importance of failure reproduction in postmortem debugging. § 3 describes the design and implementation of Pensieve. § 4 describes our dynamic refinement mechanism. We present our experimental evaluation in § 5. § 7 surveys related work and § 8 gives concluding remarks.

2 THE ROLE OF REPRODUCTION

To understand the role of reproduction in postmortem diagnosis, we study the following questions: (1) Are most failures reproduced before they are resolved? (2) How much time do developers spend on failure reproduction? (3) Does reproduction improve the understanding of a failure? Table 1 shows the results of our study.

System	Reproduced	Time (%)	Time (absolute)
HDFS	80%	78%	92 days
HBase	55%	71%	93 days
ZooKeeper	85%	57%	53 days
Total	73%	69%	79 days

Table 1: The role of failure reproduction. It shows the percentage of failures that are reproduced, the reproduction time as % of debugging time, and in absolute time.

By studying 60 randomly sampled failures, we found that developers reproduced a majority (73%) of the failures during postmortem debugging so that they can understand the failure and verify resolution. The 60 failures (20 from each system) were sampled from the JIRA issue tracking databases after filtering out bugs with a priority value lower than "Major" and those with the reporter being the same as assignee (which likely indicates failures experienced in testing rather than production systems). Failures were classified in a conservative manner: unless there is clear indication that developers have reproduced the failure, we consider it as not reproduced. Among the 16 failures that we classified as not reproduced, only for 2 did the developers clearly indicate that it was never reproduced.

Deadlocks and resource leaks were two of the common root causes in failures where developers were able to fix the bug without ever reproducing the failures. Nevertheless, we observed the following comments indicating that a reproduction would have been useful even in such cases: *"I do think I can fix the bug but I really really want to find a way to reproduce it consistently as a unit test..."; "The timeout solution is trivial but it's important to try to figure out root cause."*

Table 1 further shows that 69% of the failure resolution time was spent in failure reproduction. Failure resolution time is measured from the time a bug is reported to the time where the first correct resolution is provided, whereas the failure reproduction time is measured from the bug reporting time to the first clear indication that a developer has reproduced the failure. Studying the reproduction time is challenging because not every failure discussion records a clear point in time indicating the failure reproduction. Therefore, for the failure resolution time, we did a separate round of failure selection to select a total of 30 failures (10 from each system) where developers clearly indicated the time of successful failure reproduction. Note that while other factors (such as failure criticality or importance of the customer) could affect the reproduction time, the same factors would also affect the other portions of failure resolution. Thus, their effects should be canceled out when reporting reproduction time as a percentage of resolution time over 30 samples.

Figure 1 shows an example (HDFS-6130) highlighting the importance of failure reproduction. The failure symptoms involved data loss, therefore the bug received the highest

U: Exception caused data loss. Provided error logs.
U: Provided (incomplete) reproduction steps.
D: "I believe that this is a duplicate of..." (Wrong.)
D: "Can't reproduce. Could you try with the latest version?"
U: Proposed a (wrong) patch. Later cancelled.
D: "Apache releases don't have this issue, close as invalid."
U: "Apache release also had this issue."
U: "Add some reproduction steps."
D: "I just tried but cannot reproduce."
D: "Would be very helpful if the fsimage is available."
U: "fsimage uploaded." Also revised reproduction steps.
D: "Still cannot reproduce."
D: "Tried with another version, still cannot reproduce"
D: [After over 5 days, 29 discussions] "Reproduced..."
D: [After another 8 minutes] Posted the working patch.

Figure 1: Discussions between user (U) and developers (D).

priority ("Blocker"). It took three developers over five days to reproduce the failure, resulting in 29 discussions with users. Even after the user provided the file system image that triggered the failure, developers still could not immediately reproduce[1]. After reproducing the failure, it only took the developer 8 minutes to develop a patch that resolved the issue. As we will show in § 5, Pensieve is able to automatically infer the conditions to reproduce this failure.

We found that developers often develop a deeper understanding of the failure after reproduction. In 8 of the 30 failure samples developers adjusted the priority of the failure after reproduction. For example, only after HBase developers reproduced the failure described in HBase-4890 did they realize the gravity of the problem, evidenced by the following comments: *"Upgrade to Blocker... Should hold up (release) 0.92.1 till fixed... This is scary."*

3 EVENT CHAINING ALGORITHM

We discuss the design of the event chaining algorithm in this section. We first define the failure replication problem, its input and output. We then explain the design of Pensieve.

3.1 Problem Definition

The input data to the failure replication problem consists of the following: (1) the system's bytecode; (2) a set of external APIs; (3) a set of log files output by the failed execution; (4) a description of the failure symptoms, represented using a subset of the log messages, a stack trace, or a target program location. (A recent study has shown that a majority, 76%, of the production failures in today's distributed systems output

[1] Note that users can be irritated by such back-and-forth discussions after they already experienced a system failure. For example, in HDFS-7565, the developer could not reproduce the failure, and he kept asking the user for more information. Eventually the user stopped replying.

```
1  void transferBlock(Block b, ..) {
2    if (!isValid(b)) {
3      LOG.info("Can't send invalid block " + b);
4      return;
5    }
6  }
7  boolean isValid(Block b) {
8    ReplicaInfo r = volumeMap.get(b);
9    if (r == null) { throw IOException(..); }
10   return b.generationStamp==r.generationStamp;
11 }
12 void setGenStamp(long stamp) {
13   generationStamp = stamp;
14 }
15 // updatePipeline() executes on client
16 void updatePipeline (Block b) {
17   long newGS = b.generationStamp + 1;
18   b.setGenerationStamp(newGS);
19   LOG.info("updatePipeline(block=" + b + ")");
20 }
21 // This is a thread entry method
22 void DataStreamer.run() {
23   updatePipeline(b);
24 }
25 // appendFile() is an external HDFS API
26 void appendFile(..) {
27   streamer.start(); // -> DataStreamer.run()
28 }
```

Figure 2: Simplified HDFS code from a real-world failure.

error log messages that can be used to characterize the failure [26].) The goal is to produce a sequence of commands, in the form of external API calls with concrete values assigned to each parameter, that when executed causes the system to exhibit the required failure symptoms. External APIs are functions corresponding to the supported user operations of the system. Pensieve identifies these APIs by taking advantage of the fact that today's systems are designed with a strong focus on supporting automated testing. Each system has classes containing API methods corresponding to possible user operations (e.g., DFSClient and DFSTestUtils for HDFS, HBaseAdmin and HBaseTestingUtility for HBase). These systems also allow configuration parameters to be set using external API calls.

3.2 Motivating Example

We first use a real-world failure, HDFS-4022, to illustrate how a human developer makes use of the *Partial Trace Observation* in debugging. The failure had the highest priority (Blocker) in the bug tracker as it can potentially lead to data loss. The user characterized the failure by providing a log file containing the following error log message:

"Can't send invalid block blk_3852_1038"

Developers could not immediately reproduce the failure and had to ask the user to provide detailed reproduction steps.

Figure 2 shows simplified code from HDFS. The programmer observes that the log was printed at line 3. Because this log printing statement is guarded by the condition !isValid(b) at line 2, she concludes that the condition must hold in order for the failure to occur. We call this condition a *dominating condition* for the event at line 3. In general, if an event is necessary for the failure to occur, then any dominating condition of the event is also necessary. In practice, we ignore some unlikely dominating conditions, such as *exception-not-thrown conditions*. For example, the condition '!(r == null)' at line 9 dominates line 10, but the alternate path is an exception throw, which we consider unlikely[2].

Next, the programmer must consider why the condition at line 2 held. The return value of isValid() is computed at line 10, and depends on the variables b.generationStamp and r.generationStamp. Instead of analyzing the entire control flow path leading up to isValid(), the programmer would directly search for program locations which write a field '.generationStamp'. This field is written at line 13 in a method setGenStamp(), which has 24 different callsites in the HDFS codebase. The control flow path must have visited one of these callsites in order for the method to be called.

Since considering 24 callsites individually would take too much time, the programmer searches the log file for clues and finds the following log printed from updatePipeline():

"updatePipeline(block=blk_3852_1038)"

Because the block identifier matches the one in the original error log, the programmer focuses her analysis on the call site in updatePipeline(). In general, distributed system coding practices encourage placing logs in a way that reduces ambiguity for programmers debugging a failure [30].

Further exploration will lead the programmer to conclude that updatePipeline() was called because the user performed an appendFile() operation. Exploring the definition sites of r.generationStamp will reveal other commands required to reproduce the failure (not shown in Figure 2).

This style of debugging 'jumps' directly to prior causes instead of following the entire execution path. When the programmer jumps from the use of b.generationStamp to its definition point, she does not check that the block object b in updatePipeline() indeed flows into the object b in isValid(). This massively reduces the complexity of the analysis: in reality, the path from updatePipeline() and isValid() spans from the client to the namenode and finally to the datanode, and the block object is passed multiple times over the network. We summarize our observation as follows:

[2]If an exception-not-thrown condition is necessary, this will be detected by the dynamic verification phase (described in § 4).

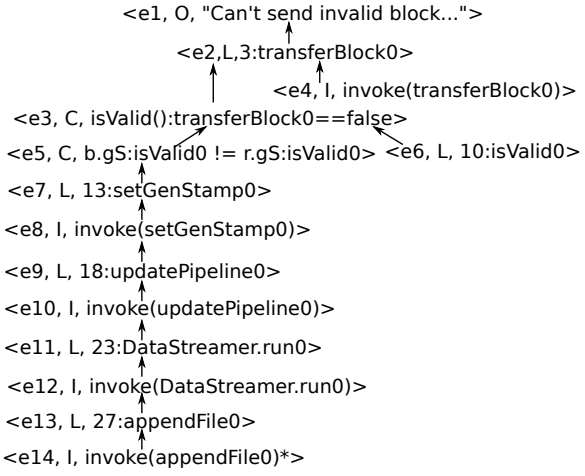

Figure 3: The event chain inferred by Pensieve on the HDFS code snippet shown in Figure 2. Each event is in the format <event-id, type, event> (we do not show task IDs in this figure). Event type "O" represents an output event, "C" represents a condition event, "L" represents a location event, and "I" represents an invocation event. → indicates a happens-before relationship. "*" indicates an external API call.

Jumping directly from an event to its prior causes (without analyzing the intermediate code path) significantly reduces the complexity of debugging.

This 'jumping' strategy is based on the Partial Trace Observation rather than on domain knowledge of the system. Pensieve implements this strategy as an automated analysis, aggressively skipping irrelevant dependencies and code paths, using logs to choose between mutually exclusive causes, and analyzing most loop bodies only once. For HDFS-4022, this produces a simplified trace containing 166 events. By comparison, a symbolic execution (SE) based approach analyzing the complete execution path (containing 72 million branch instructions) easily leads to path explosion. Even SherLog [27], a symbolic execution approach that aggressively applies heuristics to prune explored paths, infers a path constraint with over 100 million Z3 AST nodes (operators and operands) that simply cannot be solved by today's SMT solvers (§ 5.2 contains a more detailed discussion). In the end Pensieve infers a chain of events, shown in Figure 3, that captures all dependencies discussed above.

3.3 Basic Event-Chaining Analysis

Pensieve's static analysis infers chains of events that are causally necessary to reproducing the failure. An *event* identifies a point in time during the system's execution. There are four types of events:

- A *condition event* represents a condition (stored as a symbolic expression) that holds at a program location.

- A *location event* represents reaching a program location.
- An *invocation event* represents a method being called.
- An *output event* represents a log message being printed.

An event consists of three components: a unique logical timestamp, a description of the event, and a task ID identifying the process and thread that the event occurs in. The logical timestamps imply a partial order among the events [13]. When event e_B is generated as a prior cause of event e_A, Pensieve concludes that e_B happens-before e_A, or $e_B \rightarrow e_A$.

Pensieve begins with a set of output events corresponding to the failure symptoms and processes each event by searching the code base for prior causes of the event. These prior causes generate additional events to be processed. We say that an event is *explained* by finding these causally prior events. The goal of the analysis is to generate events corresponding to external API calls to the system.

Pensieve explains each type of event differently. A condition event is explained by using data flow analysis to find program locations that define each variable value used in the condition and generating new location events corresponding to these definition points. The definitions of the values are then substituted into the condition, generating a new condition event in terms of the definition points.

A location event is explained using control flow analysis to find dominating branch conditions guarding the location and generating condition events corresponding to these conditions, as well as an invocation event for the containing method. If there are two or more branch conditions 'a' and 'b' such that at least one must be satisfied to reach the explained location, Pensieve creates a condition event 'a||b'.

An invocation event is explained by generating a location event for the invoked method's callsite. An output event is explained by generating a location event that corresponds to the log printing statement. If any stack trace is printed in the log message, Pensieve analyzes it to direct the search for the location event's callsites.

Consider Figure 3. The analysis begins with a single output event e1. Pensieve explains e1 by finding the log printing statement at line 3 in Figure 2, creating a location event e2.

Each event represents a point in time during the program's execution as opposed to a static program location. Therefore, Pensieve needs to distinguish different invocations of the same method. Locations and variable values in location and condition events are assigned (using a scheme described in § 3.6) an *invocation-ID* consisting of a method name and a numerical ID. For example, "transferBlock0" in Figure 3 is an invocation ID representing one particular invocation of transferBlock(). Similarly, Pensieve uses *iteration-IDs* to distinguish iterations of a loop. Iteration-IDs are only needed when a location occurs inside a loop body. (No events in Figure 3 occur in a loop body, so iteration-IDs are not shown.)

A variable value in a condition event has the form "variable name:program location:invocation-ID:[iteration-IDs]". Pensieve's analysis models the program using Static Single Assignment (SSA) form. Therefore, each method-local variable is defined at one program location. Two variables that have the same name, program location, invocation-ID, and iteration-IDs are guaranteed to have the same value. This important property allows our analysis to detect contradictions among conditions containing the same variables. Similarly, a location event has the form "program location:invocation-ID:[iteration-IDs]", uniquely representing a point in time during the execution.

A set of causally related events forms an *event chain*. Events in the chain form a directed acyclic graph, with each edge representing a happens-before relationship. At any moment during our analysis, Pensieve maintains several chains of events representing alternative possibilities based on multiple mutually exclusive causes (e.g., an object field defined at multiple program locations). Each chain has a *search frontier* of events that are not yet explained. In our example, e1 has been explained by e2, therefore the current search frontier is {e2}. Pensieve's analysis repeatedly explains events that are in the search frontier. An event is removed from the frontier when no reasoning steps can be applied to it, such as when a condition event has resolved to true (e.g. "2==2"). Pensieve also removes a condition event on the return value of a native binary method from the frontier because Pensieve can only analyze Java bytecode. However, Pensieve retains such condition events in a buffer so that during the verification phase (§ 4) it can detect a divergence from the expected execution if the native method returns an unexpected value.

After each reasoning step, Pensieve takes the logical conjunction of all the conditions from condition events within the frontier. It then uses the Z3 SMT-solver [8] to determine whether this conjunction is satisfiable. If it is unsatisfiable, the event chain is removed from the analysis. Z3 is also used to translate constraints from condition events into concrete parameter values for external API calls.

The analysis continues until a point where the remaining unexplained events correspond to external API calls and their parameters. Usually, several API calls are required to reproduce a failure. Their ordering is inferred from happens-before relationships in the event chain. Z3 is used to assign concrete values to API parameters that satisfy the condition events where they are used. One complication is when the parameter requires an object type. In this case we first check if any previous APIs return an object of the same type. If so, we use that return value. Otherwise we construct an object by invoking its constructor and setting its fields to values that satisfy the condition events. The final output is a sequence of API calls packaged as a unit test and the event chain, a directed acyclic graph like the one shown in Figure 3.

3.4 Selective Search for Dependencies

Next, we describe specific rules Pensieve uses to selectively search for data and control dependencies. When a variable value is a field of an object, say 'obj1.obj2.field', Pensieve uses a search heuristic that we call *fuzzy search*: it searches globally in the code base for any program location defining '.field', say 'X.field', where X has matching type to obj2. The JVM memory model guarantees that all such locations can be determined statically. Pensieve does not analyze the data flow from X to 'obj1.obj2', and it does not verify whether a path exists between the definition and use points without any intervening redefinitions of '.field'. In addition to aggressively skipping code paths, this policy also allows Pensieve's analysis to include dependencies between aliased object references. For local variables Pensieve uses dataflow analysis to find a definition location inside the same method.

Pensieve explains a location event L by searching for branch conditions whose basic blocks dominate L on the control-flow graph. In addition, Pensieve generates an invocation event indicating that the method containing L needs to be invoked. In Figure 3, e4, e8, e10, e12, and e14 are invocation events. When explaining L, Pensieve discards *exception-not-thrown* conditions (i.e., branch conditions where the alternate path leads to an exception throw). Pensieve does this because a large number of statements could potentially throw exceptions, whereas in practice exceptions only occur sparingly in a real execution of the system. However, Pensieve does include *exception-thrown conditions*: if a location event e occurs in a catch block, Pensieve determines the set of instructions which could throw an exception leading to e and generates corresponding location events.

3.5 Forking and Scheduling

When there are several mutually exclusive possibilities for explaining an event, Pensieve forks the analysis so that each possibility is analyzed in a separate event chain. Specifically, Pensieve forks in the following circumstances: (1) multiple program locations defining a variable value; (2) multiple callers of a method; (3) multiple instructions throwing an exception which is caught in a catch block; (4) when a condition event is a logical disjunction. For example, a dominating condition 'a||b' causes the analysis to fork into two chains, one with condition 'a' and another with condition 'b'.

Pensieve uses a simple multilevel feedback queue scheduler to decide which forked event chains to analyze. The idea is to penalize the events that lead to a large number of forks, rewarding likely-taken paths as evidenced by log printing while pruning chains that do not encounter logs. The initial chain receives a priority value of 1000. If at any point Pensieve forks an event chain with priority P into N chains, each child chain receives a priority $P - N$. However, if a child

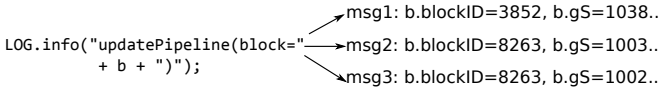

Figure 4: The logMap for a log printing statement.

chain leads to a log printing statement (LPS) which outputs a message found in the log file, Pensieve increases its priority back to 1000 while reducing the priorities of the child chain's immediate siblings to 0, effectively pruning them from the search. After each reasoning step, Pensieve selects the chain with the highest priority to analyze next. Chains of equal priority are analyzed in a round-robin manner.

Pensieve determines whether a forked chain leads to the printing of a log message found in the log file in the following manner. Before the event chaining analysis, Pensieve parses all the log messages from the log file by mapping them to LPSes and extracting variable values (log parsing has been extensively discussed in prior works [25, 27, 31]). Pensieve maintains a logMap data structure that maps each LPS L to the logs output by L. Figure 4 shows the logMap for the LPS at line 19 of Figure 2. For each event chain, Pensieve maintains a set of log messages that are assumed to be printed by the failure execution, starting with user-selected error log messages. Whenever the analysis forks on multiple program locations, Pensieve searches the code around each program location for LPSes that have a non-empty logMap, which indicates that the LPS outputs a message found in the log. The search scope includes the method containing the location as well as its callers.

There can be multiple program locations obtained from a fork with nearby LPSes that output messages found in the log. In this case Pensieve selects the LPS whose logMap contains a log message with the most logged variables whose value overlaps with existing variable values that have been parsed from other logs appearing in the current event chain. Pensieve does not require all of variable values parsed from a log to match existing values, because some values may have been modified between the printing of the two log messages. Similarly, if there are multiple log messages printed by the same LPS (i.e., multiple log messages in the logMap of this LPS), Pensieve selects the log message with the largest number of overlapping variable values. The selected log message is added to the set of logs for the current event chain.

A similar scheduling policy is used for picking which event in a chain's search frontier to analyze next. Each event has a priority value. Events leading to log output gets higher priority, while those produced from forks receive lower priority.

This policy has several advantages. First, it favors minimal failure reproductions, as more complex chains will be deprioritized. In addition, it favors failure reproductions that include more messages from the failure logs, and are therefore more likely to capture the actual root cause. Finally, prioritizing

chains leading to log output effectively reduces the effects of polymorphism on forking. Programmers tend to output different logs in different overriding methods because they themselves need to distinguish polymorphic objects when debugging. When explaining an event whose causes may occur in one of several overridden methods on an object, Pensieve performs a static search for the initialization points of the object to determine a smaller subset of possible types.

3.6 Invocation-ID Assignment

Invocation-IDs distinguish different invocations of the same method. Pensieve assigns an invocation-ID to any newly generated location event that occurs in a different method. If this method has not appeared in other events in the current event chain, Pensieve creates a new invocation-ID consisting of the method name appended with "0".

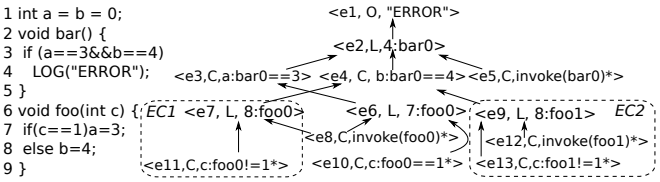

Figure 5: Example of invocation-ID assignment.

If other events already have an invocation-ID for this method, Pensieve must consider two possibilities: reusing an existing method invocation or creating a new one. Pensieve forks the chain so that the event in one child chain will use a new invocation-ID while the other child reuses the existing invocation-ID. Figure 5 shows an example with two event chains, EC1 and EC2, resulting from a fork. The events that are in the framed boxes are unique to each chain, while other events are shared by both chains. EC1 uses the same invocation-ID (foo0) in e6 and e7, which results in a reproduction that attempts to only invoke foo() once, while Pensieve uses two different invocation-IDs (foo0 and foo1) in EC2. Initially, Pensieve decreases EC2's priority to 0 after forking to favor EC1, because EC1 reuses invocation-IDs. However, in this example, Pensieve quickly infers that EC1 leads to an infeasible path due to a contradiction between e10 and e11. At this point, Pensieve resumes the analysis of EC2, which eventually leads to a feasible path, producing the command sequence 'foo(1); foo(0); bar();'.

When multiple invocation-IDs exist for the same method, Pensieve picks the one that appears in the largest number of events and prioritizes it over all other forked chains. A invocation-ID can only be reused if doing so does not introduce a cycle to the event chain.

```
1 for (i=0;i<N;i++)    1 int flag = 0;      1 int sum = 0;
2  a[i] = b[i] * 2;    2 for (i=0;i<N;i++) 2 for (i=0;i<N;i++)
3 if(a[10]==100) FAIL; 3  if (a[i] > T)     3  sum++;
                       4    flag=1;         4 if(sum > 3) FAIL;
                       5 if(flag) FAIL;
```

(A) Map loop *(B) Map loop (with condition)* *(C) Other*

Figure 6: Three types of loops that are handled differently.

3.7 Handling Loops

When Pensieve explains a condition event, the definition location of a variable value v can be within a loop body. If following the ordinary analysis logic, we would be forced to generate events for all previous iterations of the loop.

To avoid analyzing irrelevant loop iterations, when a value's origin is inside a loop, Pensieve analyzes control and data dependencies for the value's definition. We say v is a *map* value with respect to a loop if its definition point has no loop-carried dependencies outside of container indices. More specifically, Pensieve performs an intra-procedural analysis to compute control and data dependencies for v and identifies whether it is subject to a cyclic dependency. If v is a function call return, Pensieve assumes that v has data dependencies on the function call's parameters. When explaining a condition on a map value, Pensieve discards any location events that would come from previous loop iterations, unless a different event leads the analysis there. For the loop in Figure 6(A), Pensieve finds a[i] to be a map value and only analyzes the loop body once, explaining the condition a[10] == 100 by generating b[10]*2==100.

When a map value is guarded by a condition (other than the loop guard condition), the same reasoning applies if the condition is itself a map value. For the loop in Figure 6(B), Pensieve explains the condition flag!=0 by locating its definition point at line 4, and infers the dominating condition a[i]>T to be a map value. Pensieve will then search elsewhere in the program for definitions of a[i] that satisfy a[i]>T.

For other kinds of loops, such as the one in Figure 6(C), Pensieve models multiple iterations of the loop, distinguishing events in different iterations using iteration-IDs, and forking separate chains for different numbers of iterations. To minimize the number of iterations, iteration-IDs are reused using a similar policy to the one for invocation-IDs. For the example in Figure 6(C), Pensieve explains sum>3 by forking on the two definition points (lines 1 and 3), eliminating the chain where the condition became 0>3, and repeating this reasoning three more times to infer the condition N>3.

Our intuition is the proportion of loops in Java software computing map values is sufficiently large for the "map value" heuristic to be frequently applicable. Our observations on the studied software seem to match this intuition. We randomly sampled 95 loops in HDFS and classified computed

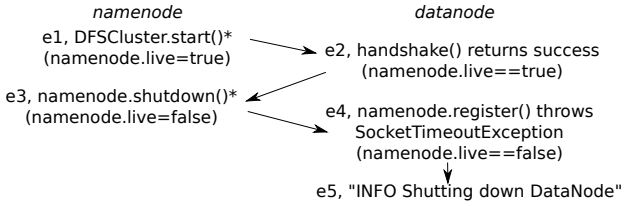

Figure 7: Event chain for an HDFS failure that is non-deterministic. Events with * are external APIs. The virtual variable Pensieve uses to model the liveness of namenode is shown in parentheses.

values which escape the loop. The majority of the loops (77%) computed a map value, 18% computed a reduction (of the form v = v op w), and only 24% of loops computed any values or side effects that did not fit a map/reduce pattern.

3.8 Task-ID Inference

For each event, Pensieve infers a task-ID in the form "process-ID:thread-ID" for use by Pensieve's dynamic verification phase: when causally dependent events have different task-IDs, there is a timing dependency between parallel tasks which must be enforced. Because Pensieve explains the invocation event of a method by locating its callsites, the analysis eventually reaches a process entry method (main()), or a thread entry method (run()) if the event belongs to a thread other than the Java main thread. The process-ID is represented as "process-entry-method:invocation-ID" and the thread-ID is represented as "thread-entry-method:invocation-ID". For example, e7-e14 in Figure 3 have task-ID "DFSClient.main0:DataStreamer.run0", while e1-e6 have "DataNode.main0:BPServiceActor.run0". Two events belong to the same thread only when they have the same task-ID. The invocation-ID in a task-ID is used to distinguish dynamic thread instances executing the same thread entry method. A dynamic thread D is mapped to the static thread "m:n" if: (1) "m:n" has not been mapped to any dynamic thread, (2) D has the entry method "m", and (3) for any unmapped static thread "m:k", $n <= k$.

Many of the non-deterministic failures in distributed systems involve the sudden change of liveness of a particular node (e.g., a node becomes unreachable at a particular time point). Figure 7 shows a simplified event chain Pensieve inferred for a real, non-deterministic failure in HDFS, where a network glitch triggered a bug that led to the shutdown of all datanodes in a cluster (HDFS-1540). The command sequence generated by Pensieve consists of two events: DFSCluster.start() that starts the namenode and the datanode, and namenode.shutdown(). However, the manifestation of the failure requires additional time constraints among internal execution states. During startup, the datanode connects to the namenode twice, first sending a handshake request to check whether the version number matches, and later registering itself with the namenode. The timing dependency is that the shutdown of the namenode has to occur between the two requests, in order to cause namenode.register() to throw a SocketTimeoutException, leading to the failure.

Pensieve models the liveness of a node by creating a variable shared by all communicating nodes (namenode.live in Figure 7). Node startup and shutdown set this variable to true and false respectively. A connection to the node is modeled by reading the variable. When Pensieve's analysis reaches a handler for IOExceptions thrown during a network connection, it infers that a condition event live==false needs to be satisfied for the destination node. Figure 7 shows the event chain inferred by Pensieve, capturing the required timing dependency. It also reveals how Pensieve captures the dependency for data race conditions (such as read-after-write) on shared variables.

3.9 Implementation Details

We implement Pensieve's analysis using the Chord static analysis framework [6] on Java bytecode.

Reusing analysis across forked chains. When Pensieve forks, each chain contains a copy of all the events in the parent chain. Pensieve avoids repeatedly analyzing the same event across multiple forked chains. After forking, Pensieve will only explain one copy of the event while marking all other copies of the same event as "blocked". After Pensieve explains all non-blocked events of a chain and the remaining events in the chain's frontier correspond to external APIs, it selects a blocked event and checks its status in the chain where it was not blocked. It has one of three states: (1) it is on a path leading to external API calls; (2) its analysis is incomplete; or (3) it leads to unexplainable events (e.g., native library calls). In the first two cases, Pensieve resumes analyzing the blocked event following the path it inferred on the non-blocked copy. Pensieve must reanalyze the event instead of reusing an explanation from another chain because invocation-, loop-, and task-IDs can be assigned differently in different event chains.

Modeling the environment. We model the semantics of the system's configuration, the external environment (e.g. file system operations) and generic libraries (e.g. Java standard library containers) by implementing additional reasoning steps ("annotations") to directly generate explanations for these operations. Pensieve used a total of six types of annotations for the evaluated cases. While the use of annotations is not ideal, we found that our event chaining model simplifies the annotation task, as all we need is to provide an explanation in terms of prior events for any generic library operation that Pensieve cannot automatically explain.

4 VERIFICATION AND REFINEMENT

Pensieve's selective analysis of dependencies leads to two major sources of inaccuracy. First, given $e_1 \rightarrow e_2$, if e_1 defines a variable v and e_2 uses v, then it is possible for v to be redefined on the path from e_1 to e_2. Second, given $e_1 \rightarrow e_2$, an instruction along the path may throw an exception, causing the execution to diverge from the path $e_1 \rightarrow e_2$.

Pensieve relies on a dynamic verification phase, Pensieve-D, to refine the event chain analysis and correct inaccuracies. Pensieve-D has three goals: (1) verify that the generated sequence of API calls can indeed reliably reproduce the failure; (2) if not, identify the point of divergence from the expected execution and correct the analysis based on feedback from the dynamic trace; (3) replicate non-deterministic failures by enforcing happens-before relationships that have a timing dependency, i.e., when two events have different task-IDs.

4.1 Divergence Detection and Refinement

Pensieve-D first executes the unit test containing the reproduction steps using the testing framework provided by each system. Mature distributed systems typically provide testing frameworks that use different threads to simulate different nodes and can process input events the same way as they are processed in a real cluster. The unit test is executed three times. If the expected failure symptoms are reproduced every time, Pensieve outputs the unit test as a successful reproduction. Otherwise it considers the execution to have diverged and searches for the point of divergence.

We detect the point of divergence by checking if any of the happens-before relationships in the event chain are violated during the execution. Pensieve-D uses the JVM Tool Interface [12] to set breakpoints at bytecode locations corresponding to each event. For a condition event, the breakpoint location is at the branch instruction where the condition was inferred. Initially, Pensieve-D only sets breakpoints at the root events in the event chain, i.e., those nodes that do not have predecessors. These include external API calls and events which Pensieve was not able to explain (e.g. return values of native methods). Once a breakpoint at node e is hit, Pensieve-D sets breakpoints for the successor nodes of e and removes the breakpoint for e. If the execution finishes (or hangs, i.e., an expected breakpoint is not hit within a 1 minute threshold) and there are still outstanding breakpoints that were not visited, the corresponding events are diverging points. A diverging point is an event that did not occur, but all of its predecessor events in the event chain have occurred.

The program location corresponding to a single location event could be executed multiple times, because the event chain only contains a partial execution trace. For example, for the majority of the loops we only analyze the loop body

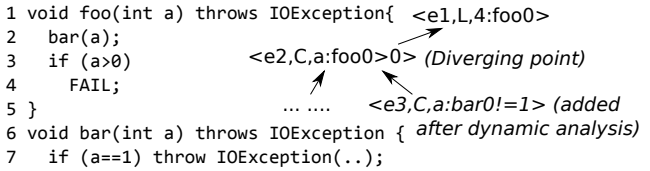

Figure 8: Refinement of Pensieve's analysis based on feedback from dynamic verification.

once and generate one event, whereas the dynamic execution executes multiple loop iterations. Our divergence detection algorithm tolerates this, as it does not require an exact mapping from the statically inferred event to the dynamic instances. As long as the program location at e_A is visited once, e_A is considered to have occurred, and we expect its successor e_B to occur without considering how many more times the program location at e_A is exercised.

A divergence can be caused by the two sources of inaccuracy in Pensieve's design: a variable value being redefined or an exception thrown by an instruction. When a breakpoint at e_A is hit during dynamic verification, where e_A defines variable value v, Pensieve-D further sets a watchpoint at v. If e_B never occured and v has been modified, then the divergence was caused by this modification. Similarly, Pensieve-D also records any thrown exceptions. Note that Pensieve-D does not stop the execution when a watchpoint is hit or an exception is thrown; as long as the next expected event e_B occurs, the variable redefinitions or exceptions are ignored as they did not cause the execution to diverge. Only when the execution diverges will Pensieve-D analyze the redefinition or exception.

When a divergence is detected, Pensieve-D *negates* the branch conditions that caused the execution to diverge and restarts the Pensieve analysis to find an explanation for the diverging branch. Figure 8 shows an example. Initially, Pensieve infers an event chain without e3, as it does not consider the exception-not-thrown condition. This results in a unit test 'foo(1);'. After executing this test dynamically, Pensieve-D observes that the expected failure symptom did not occur, and the diverging point was at *e*2 because bar() threw an IOException. Pensieve-D locates the exception throw instruction (line 7), and creates a condition event (e3) by negating the dominating branch condition 'a==1'. E3 is added as a parent node for the diverging event node. Pensieve-D then restarts the event chaining analysis with the parent node of the diverging event in the search frontier. This time, Pensieve's analysis will generate a unit test case 'foo(2);'. Similarly, if the divergence is caused by a variable value being unexpectedly redefined at location l, Pensieve-D refines the event chaining analysis by creating an event with negated dominating branch conditions of l. If there are multiple dominating branch conditions, Pensieve forks the analysis with

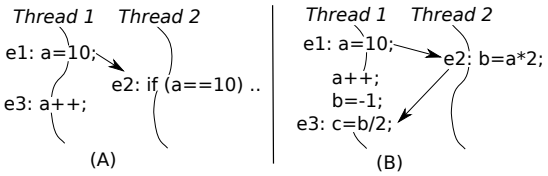

Figure 9: Sequence of statements executed in two variable redefinition cases caused by a data race. → indicates a happens-before relationship.

one chain for each, and prioritizes the chain that contains the negated condition closest to the point of divergence.

4.2 Enforcing Timing Dependencies

According to a prior study [26], 26% of production failures in distributed systems are non-deterministic. These failures are not guaranteed to manifest under the sequence of required commands, and require enforcing additional timing dependencies among internal events.

During dynamic verification, for each edge $e_A \rightarrow e_B$ in the event chain where the two events have different task-IDs, Pensieve-D enforces their execution order using breakpoints. When Pensieve-D sets a breakpoint for e_A and there is an event e_B such that e_B is from another task and $e_A \rightarrow e_B$, a breakpoint is also set for e_B. If the breakpoint on e_B triggers before the one on e_A, Pensieve-D delays the execution of the thread containing e_B until e_A arrives at its breakpoint, thus enforcing the partial order. For the failure example shown in Figure 7, Pensieve-D enforces the timing dependencies $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4$ between the two nodes. If any of the events are in a critical section (Java code section decorated by the "synchronized" keyword), Pensieve extends the data dependency of shared variables from within the critical section to the start of the critical section.

A timing dependency could be violated due to a data race redefining a shared variable. Figure 9 shows two such examples. In Figure 9(A), $e1 \rightarrow e2$ can be violated by an event $e3$ that redefines a. Although Pensieve does not infer this in its original event chain, Pensieve-D's divergence detection will identify the dependency $e2 \rightarrow e3$. Figure 9(B) shows a case where the event chaining analysis infers an infeasible path: there is no schedule that can satisfy both → relationships.

5 EXPERIMENTAL EVALUATION

We answer three questions in our experimental evaluation of Pensieve. (1) How many real-world production distributed systems failures can Pensieve reproduce? (2) How does it compare with symbolic execution approaches? (3) Why does Pensieve fail to reproduce some of the failures?

Instead of hand-picking cases on which Pensieve works best, we evaluate Pensieve on 18 *randomly sampled* failures on four widely used distributed systems: HDFS, HBase, ZooKeeper, and Cassandra. We filtered out failures whose reported symptoms did not include any log messages. Each failure was manually reproduced in order to collect a set of logs. The log files for all sampled failures included only logs at default (INFO) verbosity.

We reproduced HDFS-3436 and three Zookeeper cases on 10 nodes in order to evaluate Pensieve's performance on log files from a realistic workload. We ran a client generating a random read/write workload for an extended period of time and triggered the failure in the middle of the workload. The resulting log files were used as input to Pensieve and contained at least 10,000 messages for each failure. (Other failures were reproduced without additional workload.)

5.1 Overall Results

Table 2 shows Pensieve's effectiveness on the sampled failures. Overall, Pensieve can successfully reproduce 13 (72%) of the sampled failures within ten minutes of analysis time. These failures are complex, as evidenced by most of them requiring multiple commands to be reproduced, and some have taken the authors of this paper days to reproduce. For all of the failures, Pensieve infers the same number of commands as in the manual reproduction, suggesting that Pensieve's scheduling favors near-minimal reproduction steps.

The last column in Table 2 shows the number of unit tests generated by Pensieve. Not all of the tests generated by Pensieve's event chaining algorithm can reproduce the failure. Some failed because of missing APIs or constraints. Pensieve generated more than one failure-reproducing test for some cases either because there are indeed more than one scenario to trigger the failure (HDFS-3436) or some tests contain irrelevant APIs. We notice that when there are fewer logs to constrain the event chain exploration redundancy tend to arise, leading to the increase of the number of tests.

The number of events in the event chain leading to a successful reproduction was consistently under 300, orders of magnitude less than the number of instructions in the execution, demonstrating the power of Pensieve's design to skip irrelevant code paths. The analysis finished within 10 minutes for all cases. We believe the event chains produced by Pensieve are simple enough that programmers can manually examine them to gain further understanding of a failure. Interestingly, we found Pensieve can infer simpler explanations to the symptoms than we expected. For example, in HDFS-4022 (the failure discussed in § 3), Pensieve found a much simpler path leading to the appendFile() API compared to the reasoning in our manual analysis.

In 7 of the failures Pensieve-D's dynamic verification was required to refine the analysis. Pensieve-D enforced happens-before relationships for 2 non-deterministic failures.

There were 2 failures where Pensieve was only able to infer a partial reproduction. In both cases, the failures required

	Failure	Description	Suc.	Cmd.	Refine	Timing	Evt.	Forks	Frontier	Tests
HBase	2312	Newly written data is permanently lost	N	(failure is not fully specified by logs)						
	3403	A region cannot be accessed after region split fails	Y	4+0	N	Y	167	769	994	1/4
	3627	Region server crashes during region open operation	Y	2+0	Y	N	131	1848	2652	8/9
	4078	A column family is lost due to HDFS error	Y	3+0	N	N	177	1772	2525	1/6
	5003	Master hangs on startup due to invalid rootdir	Y	1+0	N	N	50	1813	2691	3/6
	7433	Fails when client and server use different versions	P	2	N	N	24	2	3	N/A
HDFS	1540	Temporary namenode outage brings down all DNs	Y	2+0	Y	Y	49	5	6	1/1
	3415	Namenode cannot start with modified version file	Y	2+0	Y	N	90	1811	2512	1/9
	3436	File append fails due to datanode shutdown	Y	4+0	Y	N	157	1629	2918	3/8
	3875	Corrupted block on one DN disables other DNs	N	(failure is not fully specified by logs)						
	4022	Block always remains under-replicated	Y	3+0	N	N	166	1303	1617	1/6
	4205	FSCK fails after creating a symbolic link	Y	4+0	Y	N	100	977	1385	2/6
	4558	Load balancer fails to start	Y	1+0	N	N	8	1	1	1/1
	6130	Dataloss due to an invalid fsimage	P	3	Y	N	96	2618	3075	N/A
ZooK.	1434	Checking status of a non-existent znode fails	Y	1+0	N	N	18	5	6	1/1
	1851	Client gets disconnected sending create request	Y	2+0	N	N	109	2583	2881	5/5
	1900	Trying to truncate a deleted log file fails	Y	3+0	Y	N	236	3372	4719	2/5
Cas.1299		Garbage data is written to user table	N	(requires simulating table with >200,000 columns)						
Averages			72%	2.5+0	46%	13%	105.2	1367.2	1870.5	

Table 2: Pensieve's result on four real world systems. "Suc." shows whether Pensieve can successfully reproduce the failure ('P' indicates a partial reproduction whose event chain is helpful for debugging). "Cmd." shows the number of commands inferred by Pensieve, where the first number is the minimal number of commands required to reproduce the failure, and the second number counts the additional commands inferred by Pensieve. For partially reproduced failures, "Cmd." shows how many commands were inferred in the partial reproduction. "Refine" states whether refinement was required, and "Timing" whether the failure requires enforcing a timing dependency. "Evt." is the number of events in the chain leading to a successful reproduction. "Forks" shows the number of forked event chains and "Frontier" the number of events (shared across forked chains) at the end of the analysis. "Tests" shows the number of working unit tests out of the number of total generated unit tests. For cases that Pensieve failed to reproduce we explain the reason in the last 6 columns. "Not fully specified by logs" means that the failure had additional symptoms (e.g. data loss) that were not signaled by any log message and therefore could not be specified in an input to Pensieve.

an invalid input generated by a different software version. Pensieve was able to infer required commands on the new version, as well as exact constraints on the data inputs that cause the failure, but it cannot analyze the prior version or generate a complete test case. (Pensieve was also able to infer all other commands that are required to reproduce the failures.) The partial reproductions inferred by Pensieve capture the key error conditions; we believe that they provide sufficient information for developers to quickly finish reproducing the failure.

There were several reasons for why Pensieve could not reproduce a failure. HBase-2312's symptoms consist of a data loss and a generic error message reporting data inconsistency. On its own, this error message was not sufficient to characterize the failure and guide Pensieve's search. Cassandra-1299 occurs on tables with more than 200,000 columns, requiring an event chain that is too large for Pensieve to complete.

5.2 Comparison with Symbolic Execution

We compare Pensieve with a backwards symbolic execution design based on SherLog [27]. Compared to other symbolic execution implementations, SherLog uses more aggressive heuristics to avoid path explosion. Starting from a target program location (the symptom), SherLog only symbolically executes methods that are predecessors to the symptom-containing-method in the call graph and those whose return values are used in path conditions, skipping all other functions. Because SherLog worked on C programs, we reimplemented its design on Java bytecode using Chord [6].

		SherLog		Pensieve	
Failure	Branches	Expr Size	Instr	Evt.	Forks
3436	115,741,257	72,882,516	412	157	1629
4022	72,943,652	109,018,324	693	166	1303

Table 3: Comparison of Pensieve and SherLog [27] on two HDFS failures. "Branches" is the number of branch instructions on the failure execution path. "Instr" is the number of instructions analyzed by SherLog, while "Expr Size" is the number of operators and operands (AST nodes) in the resulting path constraint.

Table 3 compares the behaviour of SherLog and Pensieve on two of the studied failures. After fewer than a thousand instructions, SherLog's path constraint is too large for Z3 to handle: simplification and other operations become extremely slow and the analysis makes no further progress.

In general, symbolic execution is poorly suited for failure reproduction in a distributed system because (1) the system's codebase is complex, with any execution path including many network operations and loops operating over data that is causally unrelated to the failure and (2) more fundamentally, the *Partial Trace Observation* further indicates that simulating this entire execution path is unnecessary. In addition, symbolically executing parallel software is difficult, and requires either including a thread scheduler model as in ESD [28] (which forks possibilities at every preemption point, further increasing the complexity of the search) or accepting some imprecision in the analysis.

By comparison, Pensieve infers a partial trace which ignores a large portion of the code on the failure execution path. Pensieve even allows its analysis to skip hypothetically unimportant parts of the data flow while jumping to important events which are surrounded by logs. Potential mistakes caused by ignoring an "unimportant" event can be subsequently corrected by dynamic refinement. Pensieve is also able to naturally model events in parallel threads by only imposing a partial order on its event chains and subsequently enforcing that partial order using Pensieve-D.

5.3 Case Study

Case 1: HDFS-4022 We continue our discussion of the analysis on HDFS-4022, described in § 3. Pensieve infers the following reproduction steps:

```
1 MiniDFSCluster.numDataNodes(1).start();
2 createFile("/random", 2);
3 appendFile("/random", "random data");
4 startDataNode(1);
```

The first command starts a cluster with only 1 datanode. The second command creates a file with replication factor 2. We have already outlined how Pensieve inferred the command appendFile(). Pensieve infers the condition 'replica.liveReplicas < requiredReplication' after searching for the callsite of transferBlock(). Pensieve further infers that requiredReplication is initialized in the second parameter of the API call createFile(String, short), and that liveReplicas comes from the parameter of MiniDFSCluster.numDataNodes(int). Therefore Pensieve infers these two external APIs must also be invoked and uses the SMT-solver to assign parameter values. Finally, Pensieve infers that, in order for transferBlock() to be invoked, at least one data node has to be started. Thus, it includes startDataNode(1) . The order of these reproduction commands is inferred from the partial order of the events.

The failure occurred because the HDFS namenode was missing code to update the generation stamp. When a file is created, the namenode puts the file's under-replicated data blocks into a replication queue. When the user appends to the file, the generation stamp of the block is updated on the datanode (from 1038 to 1039 in our example log). However, the namenode does not update generation stamps of blocks in the replication queue. When a new datanode is added, the namenode requests that block blk_3852_1038 be replicated, which the existing datanode refuses to do since it has a block with generation stamp 1039.

Case 2: HDFS-6130 This failure occured during an system upgrade. It is reproduced by starting a cluster in 'UPGRADE' mode, providing a filesystem image (FSImage) from the appropriate prior version of HDFS. Then the cluster must be shut down and restarted again. The shutdown corrupts the FSImage, causing the restarted namenode to crash with a null pointer exception.

Pensieve's static analysis infers that the failure requires cluster initialization and restart commands. On their own, these commands are insufficient to reproduce the failure, but dynamic verification detects that the execution diverged from the failure path when a variable of type INode was initialized. The initialization is treated as a redefinition (from the original null value), and Pensieve prevents it by negating the following condition in FSImageFormat$Loader.load():

```
if (NameNodeLayoutVersion.supports(
    LayoutVersion.Feature.
    FSIMAGE_NAME_OPTIMIZATION, imgVersion))
```

An FSImage satisfying this condition must come from a prior version of HDFS, which Pensieve does not model. Out of thousands of forked event chains, only 22 unique test cases were produced for dynamic verification, and only 8 have meaningful constraints on input data. This is a sufficiently small number for the developer to study the Pensieve output, recognize the event chain which constrains the FSImage version, and test the inferred commands on a suitable FSImage.

6 LIMITATIONS

The design of Pensieve makes a number of assumptions about the structure of the system and the way in which failures can be characterized. First, the system must be controlled by external API calls. Many systems take complex data structures as input. Replicating a failure may require synthesizing an instance of the data structure subject to arbitrarily complex constraints. For example, Pensieve would not work as-is for a system like MapReduce whose functionality is exercised by writing a program. However, a partial event chain inferred by Pensieve would likely still be helpful.

The failure must be characterized by a set of external outputs such as log messages or a stack trace. We assume that reproducing the log messages selected by the programmer is equivalent to reproducing the failure. In practice, failures may have additional requirements not reflected in the logs. If the selected failure logs insufficiently constrain the system's behaviour, Pensieve may produce an execution unrelated to

the underlying root cause. In such cases the programmer can verify Pensieve's reproduction to be incorrect and restart the analysis with a more complete set of failure logs.

In general, formally describing the external symptoms of a failure is an open problem. Certain types of failures are straightforward to characterize: for example, a program crash with core dump can be characterized by selecting a subset of the data structures in the core dump. Infinite loops are more difficult to characterize, since there is not a single state at which the system can be said to fail. More subtle failures in other types of software (e.g. incorrect code generated by a compiler) may only be possible to describe with reference to the expected semantics of the system.

7 RELATED WORK

Static program slicing, originally formulated by Weiser [23] in 1981 and later extensively refined [21, 24], extracts a subset of the program that is relevant to a given program state via control and data flow analysis. While Pensieve also analyzes control and data dependencies, the key difference is that Pensieve aims to infer a *dynamic* trace that is likely executed by the failure execution, instead of a subset of static program statements. The use of invocation-IDs, loop iteration-IDs, task-IDs, as well as forking and log-guided scheduling are all unique to our goal. In addition, program slicing combines mutually exclusive causes (e.g., multiple definitions of a variable) into the same slice, which can result in uninformative slices that contain most of a program [24]. (Dynamic slicing [24] can reduce the size of a slice, but requires a failure to already have been reproduced.) Pensieve separates mutually exclusive causes into different chains and uses carefully designed search heuristics to aggressively reduce the code paths being considered. Despite these differences, both Pensieve and Weiser's work [23] are inspired by human debugging principles; Pensieve pushes this idea to further extremes.

Symbolic execution, originally proposed for detecting bugs by exploring all possible execution paths [5], has recently been used to reproduce failures by searching for an execution trace containing the desired symptoms. Given a target symptom represented by a coredump, ESD [28] extracts a subprogram using static program slicing ("static phase"), then uses symbolic execution to search for paths that exercise the entirety of this subprogram and reach the symptom ("dynamic phase"). Pensieve's approach is different but complementary. Symbolic execution infers a more precise trace than Pensieve, as it analyzes the complete path. The difference between the event chaining analysis and ESD's static phase is that Pensieve aims to infer a partial trace which skips a large part of the code on the execution path. Since the event chain already captures a dynamic trace, our dynamic verification phase simply verifies this trace by executing the commands concretely instead of symbolically, avoiding path explosion. In addition, ESD requires a coredump, which typically contains much more information than the failure log and is often not available for non-crashing failures.

BugRedux instruments software to record a partial trace that reduces the search space of symbolic execution [11]. SherLog [27] is discussed in § 5.2.

Log analysis tools are used to detect performance anomalies instead of failure reproduction. *lprof* [31] assigns each log output from a distributed system to a request, generating per-request profiling information including the request latency and the nodes that were traversed. It also uses static analysis, but only analyzes the call-graph of request processing code to infer which log printing statements correspond to a request, as its goal is log grouping rather than causal diagnosis. Stitch [30] uses pattern matching on logs to reconstruct the hierarchical relationship of objects in a system. Other log analyses [16, 25] detect anomalies using machine learning techniques.

ReproLite [14] provides a Domain Specific Language to describe a series of events comprising a failure scenario, along with an engine that executes the scenario while enforcing event ordering. Its log analyzer generates an initial failure scenario based on a set of user-selected log messages. ReproLite does not aim to automatically recreate a complete failure scenario without human involvement, instead relying on the developer to manually refine the scenario.

8 CONCLUDING REMARKS

This paper describes and evaluates Pensieve, a tool for automatically reproducing failures in complex distributed systems. Pensieve's event chaining analysis is based on the *Partial Trace Observation*, which gives rise to a debugging strategy that aggressively skips code paths to produce a trace containing only relevant prior causes. We demonstrate the feasibility of using event chaining for automatic failure reproduction, using feedback from dynamic verification to refine the analysis in cases where event chaining is unsound or incomplete, enforcing timing dependencies, and making extensive use of the system's log output to guide the search.

ACKNOWLEDGEMENTS

We greatly appreciate the insightful feedback from the anonymous reviewers and our shepherd, Andreas Haeberlen. We thank Yizhan Jiang, Zheping Jiang, and Wen Bo Li for their analysis of HDFS bugs and their contributions to the initial prototype of the static analysis. We thank Xu Zhao, Michael Stumm, and Ashvin Goel for useful discussions. This research is supported by an NSERC Discovery grant, a NetApp Faculty Fellowship, a MITACS grant, and a Huawei grant.

REFERENCES

[1] Gautam Altekar and Ion Stoica. 2009. ODR: Output-deterministic Replay for Multicore Debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. 193–206.

[2] Joy Arulraj, Guoliang Jin, and Shan Lu. 2014. Leveraging the Short-term Memory of Hardware to Diagnose Production-run Software Failures. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. 207–222.

[3] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. 308–318.

[4] T Britton, L Jeng, G Carver, and P Cheak. 2013. Reversible debugging software. *Judge Business School* (2013).

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. 209–224.

[6] chord 2015. Chord: Java Bytecode Analysis. https://code.google.com/p/jchord/. (2015).

[7] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. 2013. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*. 388–405.

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. 337–340.

[9] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 525–540.

[10] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay. In *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI'02)*. 211–224.

[11] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing field failures for in-house debugging. In *34th International Conference on Software Engineering (ICSE'12)*. 474–484.

[12] jvmti 2017. JVM TI: Java Virtual Machine Tool Interface. http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/. (2017).

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

[14] Kaituo Li, Pallavi Joshi, Aarti Gupta, and Malay K. Ganai. 2014. ReproLite: A Lightweight Tool to Quickly Reproduce Hard System Bugs. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'14)*. 25:1–25:13.

[15] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. DTHREADS: Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*. 327–336.

[16] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. 26–26.

[17] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture (ISCA'05)*. 284–295.

[18] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. 2009. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP'09)*. 177–192.

[19] Dinesh Subhraveti and Jason Nieh. 2011. Record and Transplay: Partial Checkpointing for Replay Debugging Across Heterogeneous Systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'11)*. 109–120.

[20] H. Thane and H. Hansson. 2000. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS'00)*. 265–272.

[21] Frank Tip. 1995. A survey of program slicing techniques. *Journal of Programming Languages* 3, 3 (1995), 121–189.

[22] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. 15–26.

[23] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE'81)*. 439–449.

[24] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* 30, 2 (2005), 1–36.

[25] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. 117–132.

[26] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 249–265.

[27] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. 143–154.

[28] Cristian Zamfir and George Candea. 2010. Execution synthesis: a technique for automated software debugging. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems (EuroSys'10)*. 321–334.

[29] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)*. 253–267.

[30] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. 603–618.

[31] Xu Zhao, Yongle Zhang, David Lion, Muhammad FaizanUllah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'14)*. 629–644.