Achieving Scalable and Reliable Non-Intrusive Failure Reproduction in Distributed Systems by Enhancing the Event Chaining Approach

by

Xiang Ren

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2018 by Xiang Ren
Abstract

Achieving Scalable and Reliable Non-Intrusive Failure Reproduction in Distributed Systems by Enhancing the Event Chaining Approach

Xiang Ren
Master of Applied Science
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
2018

Complex and unforeseen failures in distributed systems must be diagnosed and replicated so developers can understand the underlying problem and verify the resolution. Unfortunately, failure reproduction is unpredictable and time-consuming, often leading to costly service outages.

Pensieve is a tool that automates failure reproduction by deploying a novel static analysis approach, Event Chaining (EC), which iteratively explains causal dependencies from the failure symptom while avoiding simulating the entire execution by skipping likely irrelevant instructions, which addresses the cause of poor scalability in existing approaches like symbolic execution.

Despite its aggressive design, EC is plagued by combinatorial explosion. This thesis investigates EC’s poor scalability and presents a redesign that enables EC to scale for complex failures. Further, this thesis presents a feedback mechanism that identifies instructions initially skipped by EC but are in fact relevant to the failure. Finally, this thesis presents a design that enables deterministically reproduction of concurrency failures.
Acknowledgements

I would like to thank my supervisor Professor Ding Yuan, who cultivated my interest in research, opened the door to graduate school for me, and continues to inspire me. Ding gave me detailed feedbacks on the thesis despite still being on break. I would also like to thank my committee members Professor Micheal Stumm and Professor Tarek Abdelrahman for giving invaluable feedbacks on the thesis.

I am indebted to my great teammates: Yongle Zhang, Serguei Makarov, and David Lion, from each of whom I discovered different traits of a good researcher. Many of the ideas presented in this thesis were discussed with Yongle. Serguei gave many invaluable feedbacks on writing this thesis.

Shout-out to everyone else from Ding’s team for being so supportive: Adrian Chiu, Kirk Rodrigues, Xu Zhao, and Yu Luo... Kirk helped me compile a detailed list of feedbacks during the defence. Thanks also go out to my awesome office-mates from PT477 for sharing their experiences.

Last but not least, thanks to my friends and family for always being there for me, especially my parents, who patiently listened to many lectures on this project from me. I am deeply grateful for Suya Liu and her family who actively sought advice on thesis requirements for me.
Contents

Acknowledgements iii

Table of Contents iv

List of Tables vi

List of Figures viii

1 Introduction 1
   1.1 Contributions ........................................... 2
   1.2 Organization ........................................ 3

2 Background 5
   2.1 The Role of Reproduction ................................ 5
   2.2 Event Chaining Algorithm ................................ 7
       2.2.1 Problem Definition ................................ 7
       2.2.2 Motivating Example ................................ 8
       2.2.3 Basic Event-Chaining Analysis ...................... 10
       2.2.4 Selective Search for Dependencies .................. 13
       2.2.5 Forking and Scheduling ............................. 13
       2.2.6 Invocation-ID Assignment ............................ 15
       2.2.7 Handling Loops .................................... 16
   2.3 Limitations ............................................. 17

3 Redundancy-Avoidant Event Chaining 19
   3.1 Event Chaining’s Scalability Limits .................... 20
       3.1.1 Event Chaining as a Combinatorial Algorithm .... 21
   3.2 Avoiding Redundant Explanations ....................... 23
       3.2.1 Redundancy-Avoidant Event Chaining ............... 25
List of Tables

2.1 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. 5

5.1 A sequence of events leading to deadlock. 45

7.1 Pensieve’s result on four real-world systems. “Success” shows whether Pensieve can successfully reproduce the failure (‘P’ indicates a partial reproduction whose event chain is helpful for debugging). “RA-EC” indicates whether RA-EC is required, “Refine” states whether refinement is required, and “Timing” states whether the failure requires enforcing timing dependencies. For cases that Pensieve failed to reproduce, the reason is explained in the last 3 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. 51
7.2 The effects of RA-EC on Pensieve. The failures are limited to those reproducible by Pensieve. “Log Size” shows the size of the failure log files in number of lines. “# Event Chains” shows the estimated number of event chains that would have been produced by the naïve Event Chaining algorithm. “Requires RA-EC?” indicates whether the failure requires RA-EC in order to scale. “# RA-EC Events” shows the number of events produced by RA-EC in the global event chain, “# Feasible Events” shows the number of events remaining in the global event chain after eliminating infeasible events, and “# Reachable Events” shows the number of events reachable from external inputs. “Uses Test Limit” indicates whether the failure requires applying the search restriction to only produce event chains with simplest unit tests. “# Tests” shows the number of successful unit tests against the number of all surviving unit tests that do not contain conflicts between condition events. “# Commands” shows the number of commands in the minimal unit test, where the first number indicates the commands required to reproduce the failure, and the second number indicates the additional commands present in the unit test. “Max Events/Test” shows the maximum average number of events (which may include duplicate equivalent events) in each produced unit test, whereas “Min Events/Test” shows the minimum average number of events, produced by applying the search restriction to favour simpler event chains.
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Discussions between user (U) and developers (D).</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Simplified HDFS code from a real-world failure.</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>The event chain inferred by Pensieve on the HDFS code snippet shown in Figure 2.2. Each event is in the format &lt;event-id, type, event&gt; (task IDs are not shown 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. (\rightarrow) indicates a happens-before relationship. “*” indicates an external API call.</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>The logMap for a log printing statement.</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Example of invocation-ID assignment.</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>Three types of loops that are handled differently.</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>A simplified failure scenario.</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Event chain after applying one step of the Event Chaining analysis. It contains three events: e1, e2, and e3. e1 represents the program location of the failure, which is the failure log printing statement; each of e2 and e3 represents a condition that must hold in order for e1 to occur. Each edge represents a happens-before relationship.</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>The complete result of applying the naïve Event Chaining analysis to the failure example shown in 3.1. Event Chaining inferred 4 event chains, each presenting an execution that will trigger the failure. The unit tests that can trigger the failure are also shown.</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>A global event chain constructed for the failure scenario in Figure 3.1.</td>
<td>24</td>
</tr>
</tbody>
</table>
3.5 An example of two equitable events \(e_5\) and \(e_9\), each belonging to a different event chain in the same analysis. The initial failure event is omitted. The first event chain is shown starting from \(e_4\) and the second from \(e_8\). \(e_5\) and \(e_9\) are equitable and starting from \(e_5\) and \(e_9\), the two event chains become equivalent. Each number beside a method name indicates the invocation-ID. Notice that \(e_5\) and \(e_9\) are assigned different invocation-IDs for the method "bar".

3.6 An example of equitable events not being able to produce equivalent event chains due to existing condition events. The two event chains in the figure are generated from the code snippet shown in Figure 3.7. The initial failure event is omitted. The first event chain is shown starting from \(e_4\) and the second from \(e_9\). Explaining \(e_4\) produces the condition event \(e_5\) and location event \(e_7\). Explaining \(e_9\) produces the condition event \(e_{10}\) and location event \(e_{12}\). The two different condition events, \(e_5\) and \(e_{10}\), causes the two equitable events, \(e_4\) and \(e_9\), to produce non-equivalent event chains.

3.7 An example code snippet.

3.8 An example of equitable events not being able to produce equivalent event chains due to different pre-assigned callers of the method that contain the equitable events. The two event chains in the figure are generated from the code snippet shown in Figure 3.9. The initial failure event is omitted. The first event chain is shown starting from \(e_4\) and the second from \(e_9\). When explaining \(e_4\)'s dominating condition event, \(e_5\), which is located in "bar", the caller of "foo" is pre-assigned to be "bar". On the other hand, when explaining \(e_9\)'s dominating condition event, \(e_{10}\), which is located in "baz", the caller of "foo" is pre-assigned to be "baz".

3.9 An example code snippet.

3.10 Two infeasible events are eliminated due to an unsatisfied condition. This global event chain is constructed for the failure in Figure 3.1 with the modification that "bar()" now assigns "1" to "a1" and "baz()" assigns "0" to "a2".

3.11 Two events are eliminated because they are unable to reach external inputs. This global event chain is constructed for the failure in Figure 3.1 with the modification that "bar()" now assigns "random()" to "a1".

3.12 Out of the two outcomes of \(e_2\), the outcome that produces a simpler unit test with fewer external inputs is preferred. This global event chain is constructed for the failure in Figure 3.1 with the modification that "bar()" now assigns the sum of "f1" and "f2" to "a1". "f1" is a field that is set by "api1()" and "f2" is a field that is set by "api2()".
4.1 Refinement of Event Chaining based on feedback from Pensieve-D ........................................... 39

5.1 Timing dependencies in HDFS-1540 ............................................................................................. 41

5.2 A scenario involving an interfering write: in this case, Pensieve-D infers that the read r1 should consume the output from the write w1. However, an interfering write w2 precedes r1 in the same thread. ......................................................... 42

5.3 Timing dependencies in HDFS-1540 are enforced as two read-after-write dependencies and one write-after-read dependency. ................................................................. 43

5.4 Two potential scenarios for deadlock. For case (a), Pensieve-D infers that both r1 and r2 depends on w1. Pensieve-D requires that w1 wait until the breakpoints at both r1 and r2 are reached. However, the breakpoint at the r2 cannot be reached until the r1 finishes executing, but the r1, in turn, is waiting for the w1 to execute first. A circular dependency is created in this case and the unit test execution will deadlock. For case (b), Pensieve-D infers that r1 depends on w1, but r1 itself is also a definition: w3, which Pensieve-D infers should be read by r2. After w1 finishes executing, but before r1/w3 gets a chance to execute, an interfering write w2 happens in Thread 1, and Pensieve pauses Thread 1. However, according to Pensieve’s design, w3 needs to wait until the breakpoint at r2 is hit. As a result, w3 in Thread 2 ends up waiting for r2 in Thread 1, but Thread 1 is paused at w2 and waiting for r1 to execute first. Deadlock is again created in this case. ............................................................. 44

5.5 A potential scenario for deadlock: there are two pairs of interacting threads: Thread 1 and 2, and Thread 3 and 4. For each pair, there is another interfering write from a third thread: w5 and w6. ........................................................................ 45

5.6 A set of impossible timing dependencies. In this case, r1 reads from w1 and r2 reads from w3. w2 redefines the output from w1, so w2 should happen after r1. However, w4 redefines the output from w3 and should happen before w3. It is impossible to meet both requirements. .................................................. 46
Chapter 1

Introduction

Failures in distributed systems are ubiquitous and extremely costly. A 2013 study found that half of the software development time is spent on debugging [1].

Reproducing the failure is often a prerequisite to developing a thorough understanding of the failure. A prior study surveyed 466 developers and found that the developers consider steps to reproduce a failure the most useful information in a bug report, but which is also the most difficult to provide [2]. In fact, most of the common debugging approaches, including interactive debuggers, “printf debugging”, and delta debugging [3], often require that the failure can already be reproduced. Further, reproducing the failure is also important for verifying the bug resolution: large software projects frequently require developers to provide a test case to reproduce the failure for regression testing.

In addition to being a crucial step in postmortem debugging, failure reproduction also tends to be the most time-consuming step, contributing to the difficulty of timely resolution of failures. A study [4] found that reproduction takes up 69% of the failure resolution time for 30 randomly sampled bug reports from production distributed systems HDFS, HBase, and Zookeeper. Various factors complicate failure reproductions in distributed systems: failures in distributed systems normally manifest themselves through a complex series of causal links caused by multiple user inputs [5]; communication mechanisms between threads and processes complicate the tracing of the execution flow; and finally, records of the execution are often limited to printed logs of sparse verbosity, due to performance and privacy concerns.

Unfortunately, existing solutions have limited effectiveness in speeding up the failure reproduction process. Deterministic replay techniques [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] are able to faithfully replay the failure, but come at the cost of heavy performance penalties and as a result, see limited deployment in production systems. Symbolic execution based approaches, including ESD [17] and Sherlog [18], search for paths leading to the
failure by applying program analysis on the program source code, given the coredump and printed logs: ESD uses intermediate goals inferred from static analysis to guide its search, while SherLog limits its analysis to a subset of the program. Generally, symbolic execution infers precise paths by analyzing every instructions on the path. However, because of its design to maintain preciseness, symbolic execution has difficulty scaling on large and complex software systems, caused by its requirement to fork for every branch instruction along the path even though many are irrelevant to the failure execution.

To address the scalability issues of existing failure reproduction approaches, Pensieve [4] is proposed for automatically reproducing failures from production distributed systems. Inspired by the observation that instead of reconstructing complete failure execution paths, developers tend to skip a vast majority of the code paths by focusing on instructions that are likely to be causally relevant to the failure, Pensieve presents the design of Event Chaining, a static analysis strategy that mimics the ‘jumping’ strategy used by developers to try to understand a failure (as §2.2.2 will discuss). Event Chaining produces a partial trace describing a set of events causally relevant to the failure symptom, where each event identifies a point in time during the failure execution; Event Chaining iteratively analyzes 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 set of API calls reached by Event Chaining is then used to generate a unit test, executing which is expected to reproduce the failure symptom.

However, by simply deploying Event Chaining, Pensieve is unable to achieve its goal of automatically reproducing failures. First, the naïve Event Chaining algorithm is unfortunately plagued by combinatorial explosion and cannot scale on any failures with non-trivial execution. Second, Because of Event Chaining’s design, which trades precision for scalability by aggressively skipping likely irrelevant instructions, Event Chaining may miss required dependencies or infer invalid dependencies and cannot guarantee to produce correct reproductions. Finally, Event Chaining cannot ensure deterministic reproductions for concurrency failures: it does not infer desirable interleavings of threads and processes and lacks the ability to enforce them at run time.

1.1 Contributions

Contributions of this thesis include:

* Scaling Event Chaining to real-world distributed systems.
  * This thesis identifies the root cause of Event Chaining’s combinatorial explosion prob-
problem and proposes Redundancy-Avoidant Event Chaining (RA-EC), a modification to Event Chaining that enables Event Chaining to scale for complex failure executions.

- In addition to RA-EC, this work implements an effective mechanism that enables Pensieve to produce simpler unit tests when a failure can be reproduced through multiple unit tests. Simpler unit tests containing fewer reproduction steps are preferred by developers for failure diagnosis.

**Enabling reliable and deterministic reproductions.** This thesis presents the design of a second dynamic verification phase called Pensieve-D. Pensieve-D is deployed while running the unit test produced by Event Chaining.

- To verify the dependencies inferred by Event Chaining, Pensieve-D observes the execution of the generated unit tests. If a unit test fails to reproduce the failure symptom, Pensieve-D identifies inaccuracies in the dependencies that caused the reproduction to fail and generates feedback for Event Chaining to refine the initial set of dependencies.

- To ensure Pensieve can deterministically reproduce concurrency failures, Pensieve-D infers timing dependencies between threads and processes from data dependencies captured by Event Chaining and enforces such timing dependencies during the execution of the generated unit tests.

In this thesis, “Event Chaining” or “EC” refers to the naive Event Chaining analysis and “RA-EC” refers to Redundancy-Avoidant Event Chaining. Pensieve refers to the complete tool which consists of RA-EC and Pensieve-D.

### 1.2 Organization

The rest of the thesis is organized as follows. Chapter 2 first explains the important role of failure reproduction; second, it presents the design of the Event Chaining approach in details and third, it discusses limitations of the design. Chapter 3 examines the root cause of Event Chaining’s poor scalability, and presents RA-EC, which tames Event Chaining’s combinatorial explosion problem, enabling Event Chaining to scale properly for complex distributed system failures. Chapter 4 describes Pensieve-D’s design that verifies the dependencies generated by Event Chaining, detects any inaccuracies, and generates refinement feedback for Event Chaining. Chapter 5 explains Pensieve-D’s design that infers inter-thread/process timing dependencies from causal dependencies generated by Event Chaining and enforces such timing dependencies at run time while avoiding introducing artificial deadlocks. Chapter 6 discusses Pensieve-D’s implementation choices.
that rely on the JVM Tool Interface[19]. Finally, the results of evaluating Pensieve on 18 randomly sampled distributed system failures are presented in Chapter 7.
Chapter 2

Background

2.1 The Role of Reproduction

This section presents a study taken from [4] that provides an understanding of the role of reproduction in postmortem diagnosis, where the following questions are studied: (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 2.1 shows the results of the study.

<table>
<thead>
<tr>
<th>System</th>
<th>Reproduced</th>
<th>Time (%)</th>
<th>Time (absolute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>80%</td>
<td>78%</td>
<td>92 days</td>
</tr>
<tr>
<td>HBase</td>
<td>55%</td>
<td>71%</td>
<td>93 days</td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>85%</td>
<td>57%</td>
<td>53 days</td>
</tr>
<tr>
<td>Total</td>
<td>73%</td>
<td>69%</td>
<td>79 days</td>
</tr>
</tbody>
</table>

Table 2.1: 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.

Developers reproduced a majority (73%) of 60 randomly sampled 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 bug reporter being the same as person assigned to fix the bug (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, it is considered as not reproduced. Among the 16 failures 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, the following comments indicate 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 2.1 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, a separate round of failure selection was carried out 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 2.1 shows an example (HDFS-6130) highlighting the importance of failure reproduction. The failure symptoms involved data loss, therefore the bug received the
highest 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 it. After reproducing the failure, it only took the developer 8 minutes to develop a patch that resolved the issue.

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.”

2.2 Event Chaining Algorithm

This section presents the detailed design of the Event Chaining approach taken from [4]. This section starts by defining the failure replication problem, then illustrates the Partial Trace Observation, which inspired the design of Event Chaining, with a motivating example. The chapter then explains the design and operations of the Event Chaining algorithm.

2.2.1 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 error log messages that can be used to characterize the failure [5].) 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 DFSTestingUtils for HDFS, HBaseAdmin and HBaseTestingUtility for HBase).

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.
void transferBlock(Block b, ..) {
  if (!isValid(b)) {
    LOG.info("Can't send invalid block " + b);
    return;
  }
}

boolean isValid(Block b) {
  ReplicaInfo r = volumeMap.get(b);
  if (r == null) { throw IOException(..); }
  return b.generationStamp==r.generationStamp;
}

void setGenStamp(long stamp) {
  generationStamp = stamp;
}

// updatePipeline() executes on client
void updatePipeline(Block b) {
  long newGS = b.generationStamp + 1;
  b.setGenerationStamp(newGS);
  LOG.info("updatePipeline(block=" + b + ")");
}

// This is a thread entry method
void DataStreamer.run() {
  updatePipeline(b);
}

// appendFile() is an external HDFS API
void appendFile(..) {
  streamer.start(); // -> DataStreamer.run()
}

Figure 2.2: Simplified HDFS code from a real-world failure.

These systems also allow configuration parameters to be set using external API calls.

2.2.2 Motivating Example

This section first uses 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 2.2 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
Chapter 2. Background

to occur. This condition is a control dependency [20] for the event at line 3. In general, if an event is necessary for the failure to occur, then any control dependency of the event is also necessary. In practice, some unlikely control dependencies are ignored, 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 is considered unlikely.

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 [21].

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. The observation is summarized as follows:

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

If an exception-not-thrown condition is necessary, this will be detected by the dynamic verification phase (described in §4.3).
Chapter 2. Background

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 [18], 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. In the end Pensieve infers a chain of events, shown in Figure 2.3 that captures all dependencies discussed above.

Figure 2.3: The event chain inferred by Pensieve on the HDFS code snippet shown in Figure 2.2. Each event is in the format <event-id, type, event> (task IDs are not shown 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.

2.2.3 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 established by Pensieve, and a task ID identifying the process and thread that the event occurs in. The logical timestamps imply a partial order among the events [22]. 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. 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 2.3. The analysis begins with a single output event \( e_1 \). Pensieve explains \( e_1 \) by finding the log printing statement at line 3 in Figure 2.2 creating a location event \( e_2 \).

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 §2.2.6) an "invocation-ID" consisting of a method name and a numerical ID. For example, “transferBlock0” in Figure 2.3 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 2.3 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 [23]. 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 the 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 the 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 the 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 (§4.3) 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 [24] 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 Pensieve first checks if any previous APIs return an object of the same type. If so, Pensieve uses that return value. Otherwise Pensieve constructs 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 2.3.

2.2.4 Selective Search for Dependencies

This section explains the 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 called 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 [25]. In addition, Pensieve generates an invocation event indicating that the method containing \( L \) needs to be invoked. In Figure 2.3, 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.

2.2.5 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 loca-
tions 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 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 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 \([26, 18, 27]\)). Pensieve maintains a logMap data structure that maps each LPS \(L\) to the logs output by \(L\). Figure 2.4 shows the logMap for the LPS at line 19 of Figure 2.2. 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

![Figure 2.4: The logMap for a log printing statement.](image-url)
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.

### 2.2.6 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 method previously not encountered by the analysis. 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 2.5: Example of invocation-ID assignment.](image)

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 2.5 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 2.6: Three types of loops that are handled differently.

### 2.2.7 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, Pensieve 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. \( v \) is called 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 2.6(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 2.6(B), Pensieve explains the condition $\text{flag}!=0$ 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 2.6(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 2.6(C), Pensieve explains $\text{sum}>3$ 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 $N>3$.

Intuitively, the proportion of loops in Java software computing map values should be sufficiently large for the “map value” heuristic to be frequently applicable. The observations on the studied software seem to match this intuition. We randomly sampled 95 loops in HDFS and classified computed values which escape the loop. The majority of the loops (77%) computed a map value, 18% computed a reduction (of the form $v = v \text{ op } w$), and only 24% of loops computed any values or side effects that did not fit a map/reduce pattern.

### 2.3 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.
Chapter 3

Redundancy-Avoidant Event Chaining

This chapter first examines the root cause of Event Chaining’s combinatorial explosion problem, then describes RA-EC’s modifications to the Event Chaining algorithm that are necessary to overcome its scalability limits.

Specifically, Event Chaining’s design, which enumerates the causal dependencies for all possible executions in separate event chains, is not scalable. RA-EC carries out the following strategies to allow Event Chaining to scale on real-world failures:

- RA-EC enforces that any events shared across different event chains are only explained in one of the event chains. By “coalescing” the event chains at the shared events, the size of the Event Chaining analysis becomes relative to the total number unique events encountered in the analysis and is no longer exponentially. (Note that the task of identifying shared events is not as straightforward as skipping the Event Chaining analysis of an event when the same event is already analyzed in another event chain. §3.2.2 will discuss the challenges in identifying all instances of shared events.)

- RA-EC enables a prioritization mechanism that selectively produces simpler failure explanations, which are preferred by developers for diagnosis. This further limits the Event Chaining analysis to a subset of event chains that produce unit tests with fewest user inputs and shortest causal dependency chains.
3.1 Event Chaining’s Scalability Limits

Event Chaining forks an event chain when explaining an event results in multiple possible definitions, invocation, or exception throw points. As described in §2.2.3, Event Chaining eliminates impossible forks as soon as possible by performing a series of sanity checks. For definitions, Event Chaining eliminates unsatisfiable values. For invocations, Event Chaining eliminates infeasible methods whose object types could never have been initialized in the code. Finally, for exception throws, Event Chaining eliminates those that do not throw the expected type of exception etc. However, despite these measures, a large number of forked event chains survive.

Further, out of the surviving event chains, Pensieve heavily prioritizes ones that match existing logs, because such event chains are more likely to have happened in the original failure. The rest of the event chains that do not match with logs are heavily deprioritized and thus essentially excluded from further analysis. However, log-based prioritization is not perfect; despite deprioritizing more than 95% of all event chains for HDFS-4022, the number of active event chains grows to close to 50000, after which the analysis progress slowed down significantly by multiplexing time between the large number of event chains. After running the Event Chaining analysis on HDFS-4022 for several hours, no unit test was produced by any event chain.

The lack of log printing statements in the same basic block or a dominating block of the event leaves the validity of many forked event chains difficult to determine. While in theory one can enable more verbose logging to collect finer-grained log traces, this is impractical as it may lead to unacceptable program latency and log file size. In addition, just increasing log verbosity is not enough: for log matching to be precise, logging may need to be re-engineered to allow precise reconstruction of the original execution. For example, in some instances, to confidently match an event to a log, logs printed by concurrent threads need to be distinguished through thread identifiers. Logs should also identify the loop iteration etc.

Pensieve does deploy an additional scheduling algorithm to penalize forking and ensure fairness between event chains, but the algorithm does not effectively prioritize for event chains that will successfully lead to unit tests: the scheduler cannot distinguish between event chains that will survive or become eliminated in the future, nor can it predict which event will more quickly lead to an external input event, such as an API call. Effective prioritization rules are difficult to implement on top of Event Chaining because Event Chaining lacks the design to facilitate the accurate prediction of the outcome of each event chain and the global comparison of all event chains’ outcomes. Instead,
regardless of event chains’ eventual fates, they are mostly scheduled in a round robin fashion, without effective prioritization.

The following sub-section §3.1.1 first observes that Event Chaining’s poor scalability is fundamentally attributed to its combinatorial behaviour, which causes the number of event chains to grow excessively and wastes much of Event Chaining’s analysis time on redundant analysis of already explained events. The subsequent section §3.2 provides a solution to overcome Event Chaining’s poor scalability by modifying Event Chaining to avoid redundant analysis on shared events. §3.2.2 explains the challenges in identifying all forms of shared events. With this modification to Event Chaining, it also becomes possible to embed all event chains into a global graph, as shown in §3.2, which enables the implementation of more effective prioritization schemes, as §3.3 will explain in detail.

### 3.1.1 Event Chaining as a Combinatorial Algorithm

Event Chaining is inherently combinatorial: given a failure point, Event Chaining will enumerate all feasible event chains, each representing the causal dependencies of a different execution that can trigger the failure. It is highly likely that different executions overlap in causal dependencies. As a result, distinct event chains often share events derived from common causal dependencies in execution. Since each event chain is explained individually, such shared events become repeatedly explained across different event chains, leading to redundant analysis.

To illustrate Event Chaining’s combinatorial behaviour, consider the simplified failure scenario in Figure 3.1. The failure is indicated by an error message at line 3, which is triggered when both the field “a” and “b” equal to “0”.

void foo() {
    if (this.a == 0 && this.b == 0) {
        System.out.println("Error");
    }
}

void bar(int a1) {
    this.a = a1;
}

void baz(int a2) {
    this.a = a2;
}

void foo(int b1) {
    this.b = b1;
}

void foobar(int b2) {
    this.b = b2;
}

Figure 3.1: A simplified failure scenario.

Applying one step of Event Chaining to the error message generates an initial event chain as shown in Figure 3.2. As defined in §2.2.3, an event identifies a point in time during execution. An event chain contains a set of causally related events that describe the failure execution; events in the chain form a directed acyclic graph, with each edge representing a happens-before relationship.

Figure 3.2: Event chain after applying one step of the Event Chaining analysis. It contains three events: e1, e2, and e3. e1 represents the program location of the failure, which is the failure log printing statement; each of e2 and e3 represents a condition that must hold in order for e1 to occur. Each edge represents a happens-before relationship.

Subsequently Event Chaining explains e2 and e3, giving rise to 4 possible explanations for the failure, each represented by an event chain shown in Figure 3.3.
Figure 3.3: The complete result of applying the naïve Event Chaining analysis to the failure example shown in 3.1. Event Chaining inferred 4 event chains, each presenting an execution that will trigger the failure. The unit tests that can trigger the failure are also shown.

The effect of Event Chaining’s redundant explanation is evident in the overlaps in events between the 4 event chains in Figure 3.3. For example, ec1 and ec3 both contain the event “b1 == 0”, whereas ec2 and ec4 both contain the event “b2 == 0.” In fact, each of the 4 event chains represents an unique combination, created by picking one event out of each set of outcomes for the event “a == 0” and event “b == 0”.

When applying Event Chaining to real-world failures with complex executions such as HDFS-4022, Event Chaining’s design to enumerate the causal dependencies for all possible executions in separate event chains causes the number of event chains to grow exponentially to significantly large numbers and any shared events become repeatedly explained by the large number of event chains.

### 3.2 Avoiding Redundant Explanations

Redundant explanations are both the result and cause of Event Chaining’s combinatorial explosion problem: Event Chaining’s design to enumerate the causal dependencies for all possible executions in separate event chains causes redundant explanations; at the same time, each duplicate explanation of an event may lead to further event chain forking, contributing to the explosive growth of the number of event chains.

It is possible to alleviate the combinatorial explosion problem by parallelizing the
analysis for each event chain. However, the number of event chains can grow to such high numbers that even a reasonably sized research server cluster may not have enough hardware threads to dedicate to the analysis of each event chain.

A better solution exploits the fact that the enumeration of all possible combinations of events in separate event chains is, in fact, not necessary – the analysis has enough information to produce all possible unit tests as long as each unique outcome of explaining an event exists in any one of the event chains. Consider the example illustrated in the previous section, where explaining the initial event chain shown in Figure 3.2 gives rise to 4 event chains shown in Figure 3.3. In this case, since ec1 and ec2 already contain the results from explaining the event “b == 0”, ec3 and ec4 can refer to ec1 and ec2 for the results, without having to explain the event again.

By restricting the Event Chaining analysis to only analyze each shared event in one of the event chains, the size of the analysis now becomes relative to the total number unique events encountered in the analysis. In fact, by doing so, Event Chaining will typically only need to analyze a subset of the instructions in the codebase once, producing working unit tests within minutes even when run on a single thread. This modified version of Event Chaining carries out no redundant explanations and is therefore called Redundancy-Avoidant Event Chaining, or RA-EC for short.

The output of RA-EC, an analysis which only analyzes each unique event once, can be expressed in a global event chain, where each unique event, and the outcomes of analyzing this event, only appears once globally. For the failure scenario in Figure 3.1, RA-EC would have produced a global event chain shown in Figure 3.4. All 4 of the final event chains in Figure 3.3 are now embedded in this single graph. Note that different from a normal event chain, where the relations between any two outgoing edges from an event can only be “AND”, in this global event chain, relations between two outgoing edges can also be “OR”.

![Diagram of global event chain](image)

Figure 3.4: A global event chain constructed for the failure scenario in Figure 3.1

Moreover, such a single global event chain makes prioritization rules originally difficult to realize in the naïve Event Chaining analysis, such as prioritizing for simpler reproductions, easy to apply, as § 3.3 will explain.
3.2.1 Redundancy-Avoidant Event Chaining

Redundancy-Avoidant Event Chaining modifies Event Chaining to produce the global event chain introduced in the previous section, while preserving the semantics of the original analysis as much as possible. RA-EC carries out the following tasks:

- RA-EC starts Event Chaining with its normal event explanation process and allows Event Chaining to fork event chains.
- Before explaining an event, RA-EC checks if the event $e_A$ will produce the same outcomes as any of the explained events: if so, this event will not be explained and it will refer to the outcomes of the explained event $e_B$. In this case, the explained event $e_B$ is said to block the current event $e_A$.
- After the explanation process is concluded for all event chains, all surviving event chains are merged together to produce the global event chain.

In RA-EC, an important task is to identify when an event $e_A$ is equivalent to an explained event $e_B$ such that $e_B$ should block $e_A$ to avoid redundant explanation. This identification process is not as straightforward as finding events that are exactly equal, which only represent a subset of equivalent events. The following section explains this identification process in detail.

3.2.2 Identifying Equivalent Events

Intuitively, two events do not have to be exactly equal to have equivalent effects on the output of Pensieve: as long as the events can lead to the same unit tests through equivalent sets of event chains, they are considered equivalent. As a result, just avoiding analyzing an event when the exact same event has been analyzed is not enough. (Out of all the events that RA-EC blocked, 33% matched with a non-equal but equivalent event.) In fact, as long as two events have a subset of matching properties such as event type, bytecode location etc, they are said to be equitable, and may potentially lead to equivalent event chains.

Two events are equitable if there exists a one-to-one mapping between every associated invocation-ID and loop-ID from one event A to another event B, such that if each id in event A are substituted with the corresponding id in event B, or vice versa, the two events will be equal.

In other words, two events may be associated with different iteration-IDs and loop-IDs and still produce equivalent sets of event chains. (Recall that new id assignment is random.) Figure 3.5 illustrates an example where two events, e5 and e9, each produces an
event chain equivalent to the other, even though the events are identified with different invocation-IDs, and are therefore not the same. In this case, only one of the events needs to be explained.

Figure 3.5: An example of two equitable events $e_5$ and $e_9$, each belonging to a different event chain in the same analysis. The initial failure event is omitted. The first event chain is shown starting from $e_4$ and the second from $e_8$. $e_5$ and $e_9$ are equitable and starting from $e_5$ and $e_9$, the two event chains become equivalent. Each number beside a method name indicates the invocation-ID. Notice that $e_5$ and $e_9$ are assigned different invocation-IDs for the method “bar”.

However, not every pair of equitable events can produce equivalent sets of event chains. Additional contexts from the event chains of the equitable events need to be considered; only a subset of equitable events satisfies the requirement to be considered equivalent:

**Two events are equivalent** if applying the Event Chaining algorithm to each of them produces a set of event chains such that for every event chain in one set there is an equivalent event chain in the other set, and vice versa.

where,

**Two event chains are equivalent** if there exists a one-to-one mapping between every invocation-ID and loop-ID associated with one event chain $A$ to another event chain $B$ such that if each id in event chain $A$ are substituted with the corresponding ids in event chain $B$, or vice versa, the two events chains will be two equal directed graphs.

Figure 3.6 shows an example where two equitable events are not equivalent due to a condition event in the existing event chain that constrains the outcomes of future explanations. The two event chains are generated from the code snippet shown in Figure 3.7. In the case, RA-EC determines that the two equitable events $e_7$ and $e_{12}$ are equitable. However, $e_{10}$ contains a condition event, $e_5$, that requires the parameter of “func0” to be “true”, whereas $e_{11}$ contains a condition event, $e_{10}$, that requires the parameter of “func0” to be “false”. As a result, “func1”, which passes “true” to “func0”, is the only
featural caller of “func0” in ec0, whereas “func2” is the only feasible caller of “func0” in ec1. Due to the different condition events, explaining e7 and e12 will produce non-equivalent event chains, and the two events are not equivalent despite being equitable.

Figure 3.6: An example of equitable events not being able to produce equivalent event chains due to existing condition events. The two event chains in the figure are generated from the code snippet shown in Figure 3.7. The initial failure event is omitted. The first event chain is shown starting from e4 and the second from e9. Explaining e4 produces the condition event e5 and location event e7. Explaining e9 produces the condition event e10 and location event e12. The two different condition events, e5 and e10, causes the two equitable events, e4 and e9, to produce non-equivalent event chains.
As a result, if each event is associated with a condition event, RA-EC checks that the two associated condition events are equivalent.

In addition to condition events, any pre-assigned callers of the method that contains the equitable events also affects the outcomes of future explanations. Events are pre-assigned callers only in the case of explaining return values, where before Event Chaining enters a callee to explain its return value, the caller is already known. Figure 3.8 illustrates one such example: even though e5 and e10 are equitable, their callers are not equivalent: “foo” is invoked by “bar” in one event chain and “baz” in another. As both events are further explained, the value “foo.flag” will be substituted with “bar.param” or “baz.param”, according to their different callers, producing two different event chains.
Figure 3.8: An example of equitable events not being able to produce equivalent event chains due to different pre-assigned callers of the method that contain the equitable events. The two event chains in the figure are generated from the code snippet shown in Figure 3.9. The initial failure event is omitted. The first event chain is shown starting from e4 and the second from e9. When explaining e4’s dominating condition event, e5, which is located in “bar”, the caller of “foo” is pre-assigned to be “bar”. On the other hand, when explaining e9’s dominating condition event, e10, which is located in “baz”, the caller of “foo” is pre-assigned to be “baz”.

```java
     1 void foo (boolean flag) {
     2       return !flag;
     3   }
     4
     5 void bar (boolean param) {
     6       if (foo (param)) {
     7           var1 = "first;"
     8         } else {
     9           var1 = "";
    10         }
    11     }
    12
    13 void baz (boolean param) {
    14       if (foo (param)) {
    15         var2 = "second;"
    16         } else {
    17           var2 = "";
    18     }
    19   }
```

Figure 3.9: An example code snippet.

As a result, if the stack of callers for the method that contains each event is known, RA-EC checks that the two stacks are equal.
The complete algorithm used to identify equivalent events is presented in Algorithm 1. 

\textit{areEventsEquivalent} is used to identify whether two events are equivalent: it takes in two events as arguments and returns “true” to indicate the events are equivalent and “false” otherwise.

\begin{verbatim}
Algorithm 1: Identifying Equivalent Events

1 Function areEventsEquivalent (e1, e2)
2     if e1.type \neq e2.type then
3         return false
4     end
5     foreach p1 \in e1.properties, p2 \in e2.properties do
6         if not arePropertiesEqual(p1, p2) then
7             return false
8         end
9     end
10     E1[n] \leftarrow getRelevantConditionEvents(p1)
11     E2[m] \leftarrow getRelevantConditionEvents(p2)
12     foreach ce1 \in E1[n] do
13         if \nexists ce2 \in E2[m] | areEventsEquivalent(ce2, ce1) then
14             return false
15         end
16     end
17     foreach ce2 \in E2[m] do
18         if \nexists ce1 \in E1[n] | areEventsEquivalent(ce1, ce2) then
19             return false
20         end
21     end
22     return true

Function arePropertiesEqual (p1, p2)
23     if is_id(p1.type) then
24         return true
25     end
26     foreach c1 \in p1.children, c2 \in p2.children do
27         if not arePropertiesEqual(c1, c2) then
28             return false
29         end
30     end
31     if p1.value \neq p2.value then
32         return false
33     end
34     return true
\end{verbatim}

First, \textit{areEventsEquivalent} checks that both events are of the same type. For example, if the first event is a condition event, for the second event to be potentially equivalent to the first one, the second event should also be a condition event.
In addition to a type, each event also has a description, as explained in §2.2.3. Each event description is represented by an event’s properties. Two events of the same type must have the same types of properties listed in the same order. RA-EC subsequently checks every pair of properties from the two events in `arePropertiesEqual` to make sure the two events satisfy the requirements to be equitable. Properties that are IDs including invocation-IDs and loop-IDs are ignored, since they do not affect whether two events are equitable, as explained in §3.2.2. A property may consist of children properties, in which case `arePropertiesEqual` will be recursively performed on the children properties.

If the two events pass the process of property matching, then they must satisfy the requirement that if the stack of callers for the method that contains each event is known, the two stacks must be equal. This is because the stack of callers is contained in the property tree of an event: each event contains a location property that consists of the method of the event as well as the stack of callers of that method if applicable.

In addition, each event may have a set of relevant condition events that can affect the outcomes of analyzing the event. For example, for an invocation event, a condition event can affect the outcomes of explaining the event, if it specifies a constraint on one of the parameters of the method being invoked. In such cases, for the two events to be equivalent, for each relevant condition event in one set, there must be an equivalent condition event in the other set, and vice versa.

Finally, the `areEventsEquivalent` function presented in Algorithm 1 can be modified to obtain a constant time complexity, if it instead matches two events based on their hashcodes. The hashcode of an event is computed from the relevant properties and condition events associated with each event.

### 3.2.3 Heuristics for Cyclic Dependencies and Time Complexity of RA-EC

Preventing redundant explanations cannot avoid the exponential complexity that can arise from analyzing events with cyclic dependencies. For example, even though Event Chaining limits the number of iterations which can be explained in each loop, in nested loops, Event Chaining could still need to analyze a total of $l^n$ iterations, where $l$ is the iteration limit, and $n$ is the depth of nested loops.

However, as observed in §2.2.7, most loops can be skipped or only analyzed once. Further, many events with cyclic dependencies do not have straightforward control or

---

1Two events with cyclic dependencies between them take on different forms and are not equitable. For example, the location event for the instruction “$a = a + 1$” in the first iteration will take on the form of “$a = a + 1$” but in the second iteration will take on the form of “$a = a + 1 + 1$”
data flow dependencies on external user inputs. For example, a thread’s main loop may iterate indefinitely on its own. As the goal of Pensieve is to generate unit tests consisting of external user inputs, explaining such internal cyclic dependencies may not always be required. As a heuristic, RA-EC first analyzes each cyclic dependency for one iteration, and establishes whether the dependency directly relates to external inputs. Subsequently, only cyclic dependencies that relate to external inputs are explained for additional iterations.

In the first pass of RA-EC, each cyclic dependency is only explained for one iteration. In this case, the maximum number of unique events that needs to be analyzed is on the order of $O(n^2)$, where $n$ is the number of instructions in the codebase. In contrast, the best-case complexity for naïve Event Chaining is still exponential. In the subsequent pass, RA-EC’s complexity depends on the cyclic dependencies in the program. The worst case complexity is still exponential, as in the case of nested loops.

3.3 Restricting Search Space

After RA-EC creates a combined view of all the event chains in the global event chain, it still needs to complete the step to produce unit tests based on the global event chain. This process of enumerating unit tests is unfortunately also combinatorial in nature; exhaustively enumerating all possible unit tests may not be possible for failures with more complex global event chains. However, by exploiting developers’ preference for simpler failure explanations, Pensieve can limit the search space in the global event chain to a much smaller subset and only produce unit tests with fewer external inputs and shorter causal dependency chains. The rest of this section explains the search restrictions that help achieve this goal.

1. Eliminate infeasible events. In naïve Event Chaining, many events are only found to be infeasible after many additional steps of explanation. Before the infeasible events are eliminated, Event Chaining may continue to analyze other events in the soon-to-be eliminated event chains, wasting analysis time. With RA-EC, all such events are automatically eliminated without incurring useless explanations. Figure 3.10 illustrates

---

2Consider the case where each method invocation is restricted by a constraint on the parameter specified by a condition event. Let $n =$ the number of instructions in the codebase, $r0 =$ the proportion of instructions that Pensieve considers to be invocation location events, $r1 =$ the proportion of instructions that Pensieve considers to be condition events, and $r2 =$ the proportion of other types of instructions. There are therefore $n/(1 – r1 – r)$ invocation location events, $n/(1 – r0 – r2)$ condition events, and $n/(1–r0 – r1)$ other events, producing $n/(1 – r1 – r2) * n/(1 – r0 – r2) => O(n^2)$ possible unique equivalent events. Similarly, the number of events that RA-EC needs to analyze, when each event is differentiated by pre-assigned callstacks, is also $O(n^2)$. The derivation is omitted here.
an example of eliminating infeasible events.

Figure 3.10: Two infeasible events are eliminated due to an unsatisfied condition. This global event chain is constructed for the failure in Figure 3.1, with the modification that “bar()” now assigns “1” to “a1” and “baz()” assigns “0” to “a2”.

2. Prefer events leading to external user-actionable inputs. Not all event chains lead to external user-actionable inputs. Figure 3.11 illustrates an example where explaining the condition event e2 eventually gives rise to two outcomes: one reaches the output of “random()”, which is hard to reverse-engineer and another reaches the parameter of an API method “param”, a value that can be manipulated in the generated unit test. In this case, the latter is preferred, and the two events that cannot reach external inputs are eliminated.

Figure 3.11: Two events are eliminated because they are unable to reach external inputs. This global event chain is constructed for the failure in Figure 3.1, with the modification that “bar()” now assigned “random()” to “a1”.

To apply this rule, the analysis first identifies, from the global event chain, leaf events that are external user-actionable inputs, such as API calls or API method parameters.
etc. Starting with these leaf events, every successor event is iteratively marked, until the root of the graph is reached. Subsequently, unmarked events are removed.

3. **Prefer minimal reproductions with least number of external inputs.** When multiple unit tests can reproduce the same failure, instead of enumerating all of them, prioritize for the production of the simplest tests. Figure 3.12 illustrates an example where explaining the condition event e3 produces two outcomes: one leads to two API calls and another leads to just one. In this case, the second outcome, which leads to fewer external inputs, is preferred.

![Figure 3.12](image.png)

Figure 3.12: Out of the two outcomes of e2, the outcome that produces a simpler unit test with fewer external inputs is preferred. This global event chain is constructed for the failure in Figure 3.1 with the modification that “bar()” now assigns the sum of “f1” and “f2” to “a1”. “f1” is a field that is set by “api1()” and “f2” is a field that is set by “api2()”.

To apply this restriction, the analysis starts again from the leaf events of the global event chain that are external user actionable inputs. Each leaf event is initialized with a set that contains one initial unit test consisting of the external input itself. The set of unit tests for each event is propagated to its successor iteratively. If an event has more than one predecessor, the unit tests possible at the event is produced from combining the unit tests possible at each predecessor, according to the logical relation between the predecessors. For example, if an event e1 has predecessors p1, p2, and p3 in the global event graph and the relation between the predecessors is expressed as p1&p2 | p3, then the unit tests possible at e1 is represented by: \((\text{Test}(p1) \times \text{Test}(p2)) \cup \text{Test}(p3)\).

After the possible unit tests at an event are produced, the set of unit tests are sorted in the order of increasing number of external input events, and the top \(L\) tests are kept.
4. Prefer event chains with shortest distances to external inputs when multiple event chains can reach the same unit test. Intuitively, a call to a single API method causes changes in many internal program states, such as the values of fields. Not surprisingly, the same API call can be frequently traced from distinct chains of causal dependencies. In fact, explaining only one of these chains will suffice for reaching the API call, and the shortest one can be picked.

This restriction also relies on the same unit test propagation logic described in the previous rule. In addition, for each external input event in an unit test, the distance from the external input event to the current event, in the number of events, is kept. Each distance is incremented once when the unit test is propagated to a successor event.

When identical unit tests, each of which contains the same set of external inputs, are propagated to the same event, the unit tests are ranked according to the sum of the distances. The unit test with the shortest total distance is kept.
Chapter 4

Pensieve-D Verification and Refinement

The Event Chaining algorithm trades accuracy for scalability by aggressively skipping instructions likely irrelevant to the failure execution. As a result, an event chain may not capture all the required dependencies or may contain invalid dependencies, and the unit tests generated from initial event chains are not guaranteed to reproduce the failure. (§4.2 describes the design choices in Event Chaining that can lead to incorrect event chains.)

This chapter describes Pensieve-D - Pensieve’s dynamic phase that verifies the correctness of an event chain. Pensieve-D detects any inaccuracies that cause the unit test execution to diverge from its intended failure symptom and instructs Event Chaining to refine the event chain accordingly.

4.1 Verification and Divergence Detection

To verify the correctness of the event chain, 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.

To detect divergence points, Pensieve first checks whether the orders at which events happened in the unit test violates any causal dependencies specified by the event chain.
Pensieve-D uses the JVM Tool Interface [19] 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 leaf 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 removes the breakpoint for $e$, and sets breakpoints for any successor nodes of $e$ whose predecessors have all been visited. 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 events corresponding to the breakpoints are diverging points. A diverging point is identified as an event that did not occur, but all of its predecessor events in the event chain have occurred.

Because events inferred by Event Chaining describe the dynamic execution, more than one event may be located at the same bytecode location. Further, because an event chain only contains a partial trace, a bytecode may be executed more times than the number of events mapped to it. In order to determine whether a hit bytecode instruction actually corresponds to an event, Pensieve-D matches the callstack of the instruction against the callstack inferred from an event’s invocation ids. Further, the loop iterations for each level of nested loops the event and executed instruction reside in also needs to be compared, if precise loop iterations for the events are inferred by Event Chaining. However, for the majority of the loops Event Chaining only analyzes the loop body once and generates one event, whereas dynamically multiple loop iterations may be executed. Pensieve-D tolerates such imprecision, as it does not require an exact mapping from a statically inferred event to the dynamic instances: Pensieve-D will accept the first occurrence of the desirable event inside the loop body, and ignore occurrences in subsequent iterations.

4.2 Identifying Causes of Divergence

For each detected divergence point, the cause of divergence is identified. The following causes of divergence are possible as a result of Event Chaining’s particular design choices:

1. **an condition event becomes violated.** A condition event can become violated when its variables take on unexpected values, causing the branch instruction before the divergence point to take an undesirable target. This can happen with:
   - Mismatched object references - since Event Chaining does not analyze the object references of fields, a pair of use and definition of a field can be performed on different objects, creating invalid dependencies.
Variable Redefinitions: Event Chaining does not reason through the entire execution and is therefore unaware of redefinitions.

To determine the actual cause of unexpected variable values, the deterministic (Chapter 5 explains how Pensieve ensures deterministic reproductions.) unit test is re-run, during which Pensieve watches for any redefinitions that happen between the intended definition and the use instructions. If the variable is a field, Pensieve also checks whether object reference of the field at the use and definition instruction match. The details of implementation are explained in §6.1.1 and §6.1.2.

Redefinitions can happen in the same thread as the intended definition or from a different thread, which results in race conditions. §5.2 explains how such race conditions are avoided by Pensieve-D’s timing component. §4.3 discusses how redefinitions in the same thread are avoided.

2. the control flow becomes redirected or the program terminates without violating existing condition events.

Even when all relevant branch instructions take the expected target, the control flow can still become redirected or terminated by events such exception throws, process exits and thread interrupts etc. §6.2.1 and §6.2.2 describes how Pensieve-D identifies the most common causes.

3. the program “freezes”.

The program is prevented from making progress to the divergence point before the unit test times out. The program may appear to “freeze” for many reasons. It is impossible to differentiate between the program being “stuck” and being “slow” as famously demonstrated by the undecidable halting problem [28]. While deadlocks caused by logic built on top of Java synchronization primitives may be detected as §6.2.2 discusses, heuristics such as timeouts must be used to guess that the program may be stuck in infinite loops.

4.3 Refinement

According to the identified the diverging causes, Pensieve-D generates feedback for Event Chaining and restarts Event Chaining to refine the event chain. There are three types of scenarios for refinement:

1. For undesirable events such as exception throws or redefinitions etc, prevent the event from happening by negating the branch conditions leading to the event:
Figure 4.1: Refinement of Event Chaining based on feedback from Pensieve-D.

Figure 4.1 shows an example. Initially, Event Chaining 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 e2 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, Event Chaining’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-D forks the analysis with one chain for each, and prioritizes the chain that contains the negated condition closest to the point of divergence.

2. For desirable events that did not occur, for example, loop exits for infinite loops, ensure the event will happen by including the path conditions leading to the event: Both the loop exit event and its path conditions are incorporated into the existing event chain. Event Chaining is restarted to find explanations for the newly added condition events.

3. For invalid dependencies resulting from mismatched object references for fields, choose a different event chain that enumerated a correct dependency: The current event chain enumerated a bad definition for a field and needs to be given up in favour of another event chain with a correct definition.
Chapter 5

Pensieve-D Timing

In order to create deterministic reproductions for concurrency bugs that manifest through specific interleavings of threads or processes, Pensieve-D infers timing dependencies for events across parallel-running threads or processes and enforces them during unit test execution. This chapter describes the design of said procedures.

5.1 Inferring Timing Dependencies

Event Chaining’s fuzzy search captures causal dependencies regardless of whether the dependencies are within the same thread/process or across different threads/processes. To determine whether certain casual dependencies translate into timing dependencies between parallel-running threads or processes, Pensieve-D needs to uniquely identify the process and thread of an event.

To do so, for each event Pensieve-D also infers a process id and thread id. When Event Chaining analysis encounters an invocation to Thread.run(), Pensieve-D treats the invocation as a distinct thread entry point; an invocation to the main() method indicates a unique process entry point. Control-flow successor events of the process or thread entry point will acquire the unique identifier assigned to the process or thread.

Data dependencies across different threads or processes are translated into timing dependencies in the form of a series of read-after-write (RAW) dependencies that must be enforced at run time.

5.1.1 Process-Level Concurrency Bugs

While concurrency bugs are commonly associated with thread level shared variables, in our randomly sampled failures, all timing dependencies occur at the process level.
Consider the case for HDFS-1540, where the timing dependencies manifest during inter-process communication: the datanode carries out two remote procedural calls (RPCs) with the namenode: the first one performs an initial handshake, and the other registers the datanode. After the handshake RPC succeeds, the namenode unexpectedly goes down. As a result, the second RPC to register the datanode fails, which causes the failure to occur. The sequence of events are illustrated in Figure 5.1.

![Figure 5.1: Timing dependencies in HDFS-1540](image)

In this case, the liveness of the namenode can be modeled as a shared boolean variable: the liveness is expected to be true for both RPC calls to succeed; however, in this case, the namenode goes down unexpectedly after the first RPC, setting its liveness variable to false, causing a race condition on the variable. Subsequently, when the second RPC call accesses the liveness variable, it reads an unexpected value and fails.

Modelling the liveness of a process as a shared variable is not a specific solution to HDFS-1540. In fact, all the sampled concurrency failures involve the sudden change of liveness of a process; changes liveness are extremely common in distributed systems and can happen abruptly. Processes can become unresponsive for a variety of reasons, including server crashes, network failures or long-running garbage collection etc. The liveness of a process can, therefore, be a general heuristic for modeling failures that occur due to badly design fault-tolerance mechanisms.

### 5.2 Enforcing Timing Dependencies

This section first explains how Pensieve-D uses artificial delays to enforce the inferred timing dependencies between threads or processes in the form of read-after-write (RAW) dependencies, then discusses the design needed to prevent interfering writes that redefine the output from intended write.

To enforce a pair of RAW dependency on a shared variable, Pensieve ensures that the intended write in the producer thread/process (producer) happens before the read in

---

**Figure 5.1**

**Timing dependencies in HDFS-1540**
the consumer thread/process (consumer): Two breakpoints are set at the read and write respectively. If the breakpoint at the read is hit first, the consumer will be paused until the write is hit and executed.

In the case of HDFS-1540, a breakpoint is set at “namenode.shutdown()” and “namenode.register()”. If the breakpoint at “namenode.register()” is hit first, the datanode process is paused until “namenode.shutdown()” finished executing, ensuring that by the time “namenode.register()” is invoked, namenode is already down.

In addition, for any interfering writes that redefine the output from intended write, Pensieve-D needs to schedule them to happen either before the intended write, or after the read of interest:

Interfering writes that happen in the consumer before the read must be scheduled to happen before the intended write in the producer: To achieve this, if the breakpoint at the intended write is hit first, the producer is paused to wait for the consumer to hit the read of interest. This makes sure that by the time the intended write is executed, interfering writes preceding the read in the consumer already took place. In this case, Pensieve-D enforces an ordering between the two writes, creating an artificial write-after-write (WAW) dependency. Figure 5.2 illustrates a simple scenario where, by requiring that the breakpoints at r1 and w1 both need to be hit before w1 can proceed, Pensieve-D ensures that w2 happens-before w1.

![Figure 5.2](image.png)

**Figure 5.2:** A scenario involving an interfering write: in this case, Pensieve-D infers that the read r1 should consume the output from the write w1. However, an interfering write w2 precedes r1 in the same thread.

In contrast, interfering writes that happen in threads other than the consumer can be delayed to happen until after the read of interest. To achieve this, after both breakpoints at the intended write and read are hit, Pensieve performs the following actions atomically: it first executes the intended write, then sets a watch on the shared variable, and finally resumes both the producer and consumer. Subsequently, if any thread triggers the watch by attempting to write to the shared variable, the offending thread is paused, and only
resumed after the read of interest finishes executing. In such cases, Pensieve makes sure the interfering write happens after the desirable read, implicitly creating a write-after-read (WAR) dependency.

In the case of HDFS-1540, a third WAR dependency is enforced between “handshake()” and “namenode.shutdown()”, as shown in Figure 5.3. Since “handshake()” depends on namenode being live, if “namenode.shutdown()” is hit early, it is delayed until “handshake()” has returned. Since “namenode.shutdown()” is not a simple write instruction, instead of setting a watch, a breakpoint is set at the invocation to “namenode.shutdown()” to detect changes to namenode’s liveness.

![Diagram](image)

**Figure 5.3:** Timing dependencies in HDFS-1540 are enforced as two read-after-write dependencies and one write-after-read dependency.

The above scenarios apply to the simplified case of only one pair of RAW dependency. In the case where multiple reads depend on a single write, the write must wait until the breakpoints at all the reads are reached. Similarly, if a read depends on multiple writes, the read must wait until the breakpoints at all the writes are reached.

### 5.2.1 Avoiding Deadlocks

Introducing artificial delays as described in §5.2 could lead to undesirable deadlocks, and care must be taken to avoid artificially introducing such deadlocks.

Deadlocks may arise when the reads or writes are inside atomic regions protected by locks. (§6.3 describes how atomic regions can be detected.) Consider the case where Pensieve infers that a write of a shared variable should happen before a read, and both the read and write are protected by the same lock. If the consumer thread acquires the lock first, then deadlock would arise because the read would wait for the write, but the producer thread is waiting to acquire the lock. To avoid this type of deadlock, Pensieve needs to consider the entire atomic region as a whole, and set breakpoints at the start of the atomic regions, instead of at each individual instructions.
A second category of deadlocks may arise because Pensieve-D requires that only when the all the breakpoints at the write(s) and read(s) are hit, can the write(s) proceed. Figure 5.4 illustrates two scenarios where deadlock occurs due to this requirement.

To break this kind of deadlock, whenever Pensieve detects circular waits between 2 threads/processes, and one thread/process has a pending write that’s waiting for its reads to be hit, it lets the pending write proceed, and set a watch on the variable just written to. With this rule applied, for both scenarios in Figure 5.4, the desirable timing orders are still enforced without introducing deadlocks: In the case of (a), w1 will be allowed to execute even though the breakpoint at r2 has not been hit. In the case of (b), w3 will be allowed to execute even though the breakpoint at r2 has not been hit.

![Figure 5.4](thread-diagram.png)

Figure 5.4: Two potential scenarios for deadlock. For case (a), Pensieve-D infers that both r1 and r2 depends on w1. Pensieve-D requires that w1 wait until the breakpoints at both r1 and r2 are reached. However, the breakpoint at the r2 cannot be reached until the r1 finishes executing, but the r1, in turn, is waiting for the w1 to execute first. A circular dependency is created in this case and the unit test execution will deadlock. For case (b), Pensieve-D infers that r1 depends on w1, but r1 itself is also a definition: w3, which Pensieve-D infers should be read by r2. After w1 finishes executing, but before r1/w3 gets a chance to execute, an interfering write w2 happens in Thread 1, and Pensieve pauses Thread 1. However, according to Pensieve’s design, w3 needs to wait until the breakpoint at r2 is hit. As a result, w3 in Thread 2 ends up waiting for r2 in Thread 1, but Thread 1 is paused at w2 and waiting for r1 to execute first. Deadlock is again created in this case.

Additionally, Pensieve-D only allows one such pending write to proceed each time, and only after the output of this write has been read, will Pensieve-D let another pending write execute. This is because letting two such pending writes proceed at the same time can also cause deadlocks. Consider the situation in Figure 5.5 Table 5.1 illustrates a sequence of events that leads to deadlock due to letting two pending writes proceed together.
Figure 5.5: A potential scenario for deadlock: there are two pairs of interacting threads: Thread 1 and 2, and Thread 3 and 4. For each pair, there is another interfering write from a third thread: w5 and w6.

Table 5.1: A sequence of events leading to deadlock.

Deadlocks that still happen indicate that the statically generated read-write dependencies are likely impossible, and Figure 5.6 demonstrates one such situation, where the timing orders are impossible to enforce.
Figure 5.6: A set of impossible timing dependencies. In this case, r1 reads from w1 and r2 reads from w3. w2 redefines the output from w1, so w2 should happen after r1. However, w4 redefines the output from w3 and should happen before w3. It is impossible to meet both requirements.
Chapter 6

Implementation

This chapter discusses the implementation choices made in Pensieve’s Verification and Timing Analysis components, many of which are heavily dependent on the design of the Java Virtual Machine (JVM), particularly the JVM Tool Interface[19] (JVMTI), which provides its user the ability to control the execution of the JVM and inspect states of the program.

6.1 Identifying Causes of Violated Condition Events

Recall from §4.2 that a condition event can become violated when its variables take on unexpected values, causing the corresponding branch instruction to take an undesirable target.

6.1.1 Detecting Object Reference Mismatches

Because Pensieve deploys fuzzy search and does not check the object references of fields, it may explain the use of a field with a definition from a different object, causing divergence in the unit test execution later. JVMTI enables detecting such object reference mismatches: at runtime, a unique numerical tag can be added to each object, and Pensieve can simply compare the tags at the field modification and access. Tagging the object is required because the references may change after garbage collection, whereas the tag will remain constant for the life time of the object.

6.1.2 Detecting Redefinitions

Redefinitions are another cause of divergence that Event Chaining does not check in its initial analysis. To detect redefinitions on field variables, a watch is set on the field
variable at the intended write, and the object reference is tagged. Before the field is accessed by the intended read, any modifications to the field triggers a handler. The handler filters out any false redefinition whose object tag is different from the object of interest; otherwise, the real redefinitions are recorded as feedback for refinement. When Pensiee suspects possible redefinitions on a local variable, it determines all the instructions in the method that can modify the variable, and re-runs the unit test with a breakpoint set at each of the instructions. Triggering any of the breakpoints between the intended write and read instructions indicates a redefinition.

6.2 Identifying Causes of Control Flow Interruptions

Even if no condition events are violated, the unit test could still be prevented from reaching a desirable event because the control flow becomes redirected or stuck.

6.2.1 Detecting Exceptions

Event Chaining does not consider exception throw instructions that can redirect the control flow away from desirable events. While running the unit test, Pensiee records all exception throw and catch events, both of which can be observed through the JVMTI. Exceptions thrown before the divergence point, but never caught or caught after the divergence point are deemed to have caused the divergence.

6.2.2 Detecting Other Forms of Control Flow Interruptions

There exist many additional forms of control flow interrupts than the events that can be easily observed through the JVMTI. Divergence causing program exits, thread interrupts etc., are not straight forward to detect and require setting a breakpoint at each possible location. Deadlocks may be detected by examining the monitors a thread holds and are waiting on if the Java monitor is used to implement the locks. Rare causes of divergences, including those caused by events from the environment, for example, signals to terminate the JVM, lacks support from the JVM to be detected.

6.3 Detecting Critical Regions

Enforcing timing dependencies on shared variables inside critical regions requires that the critical region surrounding a read or write be treated as a whole. It is possible to detect the critical regions if the locks are implemented on top of the Java monitor: the
bytecode MONITOR-ENTER and MONITOR-EXIT signify the entry into and exit from
the critical regions.
Chapter 7

Evaluation

This evaluation attempts to answer the following questions: (1) How many read-world distributed system failures can Pensieve reproduce? Why does Pensieve fail to reproduce some of the failures? (2) What percentage of failures requires each of the components described in this thesis: RA-EC, Pensieve-D Refinement, and Pensieve-D Timing? (3) How effective is RA-EC in addressing the the naïve Event Chaining’s scalability issues? What are the effects of the different search restrictions?

To answer these questions, Pensieve is evaluated on 18 failures, randomly sampled from four popular distributed systems: HDFS, HBase, ZooKeeper, and Cassandra, after excluding failures whose failure symptoms are not expressed in printed output such as printed logs or stacktraces etc. Table 7.1 shows an overview of the evaluation.

Overall, Pensieve is able to reproduce 13 (72%) of the cases, out of which, 10 requires RA-EC to scale. 6 cases require refinement, and 2 requires enforcing timing dependencies.

Pensieve is unable to reproduce 2 of the failures because the failures are not fully described by logs. For example, HBase-2312’s failure results from a data loss and the generic error message reporting data inconsistency does not fully character the specific data loss scenario. Pensieve cannot reproduce Cassandra 1299 because it requires writing a very large an amount of data to the table for Pensieve to complete.

For another 2 of the cases, Pensieve can only infer partial reproductions, since both failures require a prior version of the system to reproduce.
### Table 7.1: Pensieve’s result on four real-world systems

<table>
<thead>
<tr>
<th>Failure</th>
<th>Description</th>
<th>Success</th>
<th>RA-EC</th>
<th>Refine</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>2312</td>
<td>Newly written data is permanently lost</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3403</td>
<td>A region cannot be accessed after region split fails</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>3627</td>
<td>Region server crashes during region open operation</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>4078</td>
<td>A column family is lost due to HDFS error</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>5003</td>
<td>Master hangs on startup due to invalid rootdir</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>7433</td>
<td>Fails when client and server use different versions</td>
<td>P</td>
<td>-</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>1540</td>
<td>Temporary namenode outage brings down all DNs</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>3415</td>
<td>Namenode cannot start with modified version file</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>3436</td>
<td>File append fails due to datanode shutdown</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>3875</td>
<td>Corrupted block on one DN disables other DNs</td>
<td>N</td>
<td>failure not fully specified by logs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4022</td>
<td>Block always remains under-replicated</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>4205</td>
<td>FSCK fails after creating a symbolic link</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>4558</td>
<td>Load balancer fails to start</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>6130</td>
<td>Dataloss due to an invalid fsimage</td>
<td>P</td>
<td>-</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>1434</td>
<td>Checking status of a non-existent znode fails</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>1851</td>
<td>Client gets disconnected sending create request</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>1900</td>
<td>Trying to truncate a deleted log file fails</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>1299</td>
<td>Garbage data is written to user table</td>
<td>N</td>
<td>(requires simulating table with &gt;200,000 columns)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Averages | 76% 59% 35% 12% |

Table 7.1: Pensieve’s result on four real-world systems. “Success” shows whether Pensieve can successfully reproduce the failure (‘P’ indicates a partial reproduction whose event chain is helpful for debugging). “RA-EC” indicates whether RA-EC is required, “Refine” states whether refinement is required, and “Timing” states whether the failure requires enforcing timing dependencies. For cases that Pensieve failed to reproduce, the reason is explained in the last 3 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.

Table 7.2 shows how RA-EC affects the output and scalability of Pensieve, detailing the effects of each type of optimization, as well as additional factors affecting Pensieve’s scalability, such as the size of the failure log files. The table only displays failures for which Pensieve can produce working reproductions.
Table 7.2: The effects of RA-EC on Pensieve. The failures are limited to those reproducible by Pensieve. “Log Size” shows the size of the failure log files in number of lines. “# Event Chains” shows the estimated number of event chains that would have been produced by the naïve Event Chaining algorithm. “Requires RA-EC?” indicates whether the failure requires RA-EC in order to scale. “# RA-EC Events” shows the number of events produced by RA-EC in the global event chain, “# Feasible Events” shows the number of events remaining in the global event chain after eliminating infeasible events, and “# Reachable Events” shows the number of events reachable from external inputs. “Uses Test Limit” indicates whether the failure requires applying the search restriction to only produce event chains with simplest unit tests. “# Tests” shows the number of successful unit tests against the number of all surviving unit tests that do not contain conflicts between condition events. “# Commands” shows the number of commands in the minimal unit test, where the first number indicates the commands required to reproduce the failure, and the second number indicates the additional commands present in the unit test. “Max Events/Test” shows the maximum average number of events (which may include duplicate equivalent events) in each produced unit test, whereas “Min Events/Test” shows the minimum average number of events, produced by applying the search restriction to favour simpler event chains.

Any failure with non-trivial execution requires RA-EC in order to scale, as shown
in the “Requires RA-EC?” column in Table 7.2. Several factors are indicative of the complexity of a failure execution: for all 3 failures that do not require RA-EC, Pensieve unambiguously found only one possible reproduction (# Tests); each of the tests also consists of only 1 to 2 commands (# Commands). Note also that for each of the 3 failures, the maximum number of estimated event chains is very low, being no more than several hundreds.

The “# Event Chains” column in Table 7.2 shows the number of event chains estimated from the global event chain produced from RA-EC. The number of event chains is estimated based on the fact that, each event, whose explanation is blocked by an equivalent event, would have produced an equal number of forks as the equivalent event. The number of forks of each surviving event chain is estimated as the product of the forks each blocked event would have produced. And the total number of forks is the sum of the estimated forks of each event chain.

Without RA-EC, most failures are estimated to produce an extremely large number of event chains. On average, $5 \times 10^{186}$ event chains are produced for each case. Outside of the 3 simple failures, no case would likely complete in any reasonable amount of time. In contrast, by avoiding redundant explanations and applying the search restrictions, an average of only 5 unit tests are produced for each case, resulting in a $10^{186}$-fold reduction in the number of event chains.

The “# RA-EC Events” column in Table 7.2 shows the number of events in the original global event chain. Restricting the search space of the global event chain to only feasible events can moderately reduce the number of events in the global event chain to on average 72% to the original size. The number of feasible events in each global event chain is shown in the “# Feasible Events” column in Table 7.2. In the naive Event Chaining analysis, however, the 28% infeasible events do add additional pressure to the blowup problem before their event chains are eventually eliminated.

On the other hand, restricting the search space to only events reachable from external inputs significantly reduces the search space to only 16% of the original size. The number of events reachable from external inputs in each global event chain is shown in the “# Reachable Events” column in Table 7.2. Intuitively, an external command may change many states in the system, and it suffices to determine the commands by only tracing a small subset of the changed states that have straight forward data or control dependencies to the commands. In effect, this restriction aggressively favours Event Chaining’s “jumping” behaviour to closest external inputs, by de-prioritizing causal dependencies.

A step limit of 18000 is applied to RA-EC. Without this limit, RA-EC may continue to produce extra unit tests and may not finish running.
stuck in complicated internal logic such as network serialization/deserialization code etc.

After applying the two discussed restrictions, the analysis is now restricted a small subset of the global event graph. However, the number of unit tests generated from the global event chain can still be significant. As shown in the “Uses Tests Limit?” column in Table 7.2, 3 cases require limiting the number of unit tests to the most minimal ones during the unit test enumeration process. This restriction guarantees that the resulting number of tests produced for each case is relatively low, and the minimal test is produced for each of the cases.

Each of the 3 cases that require limiting the unit test count has the highest number of externally reachable events in the global event chain. Further, the shape of the global event graph is a more important contributing factor to the unit test count: the abundance of edges between events directly contributes to greater number of possible unit tests. Also interestingly, the 3 cases all have relatively less complete logs: HBase3627 has only 17 lines of logs, HBase3415 has 131, and Zookeeper1900 54 - more complete logs possibly have the effect of reducing the the number of events as well as the complexity of the graph’s structure.

The “# Tests” column in Table 7.2 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 Pensieve did not infer necessary API calls or constraints. Pensieve generated more than one failure-reproducing unit test for some cases either because there are indeed more than one scenario to trigger the failure (HDFS-3436) or some tests contain irrelevant API calls. For HDFS-3436, Pensieve is able to find a unit test simpler than the one provided by the developer.

After the unit tests for each case are produced, the shortest paths to the external commands are chosen for each test. Applying this rule reduces the number of events in the unit test by an average of 21%, as shown by the “Max Events/Tests” and “Min Events/Tests” columns in Table 7.2
Chapter 8

Related Work

This chapter discusses related work for Pensieve and is taken from [4].

**Static program slicing**, originally formulated by Weiser [29] in 1981 and later extensively refined [30, 31], 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 [31]. (Dynamic slicing [31] 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 [29] 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 [32], 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 [17] 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 \cite{33}. SherLog \cite{18} 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.

**Log analysis tools** are used to detect performance anomalies instead of failure reproduction. lprof \cite{27} 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 \cite{21} uses pattern matching on logs to reconstruct the hierarchical relationship of objects in a system. Other log analyses \cite{34,26} detect anomalies using machine learning techniques.

ReproLite \cite{35} 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.
Chapter 9

Conclusion

Pensieve is a tool designed to automatically reproduce complex distributed system failures. Pensieve addresses the issue of poor scalability of existing approaches, such as symbolic execution, by deploying the static analysis strategy Event Chaining. Inspired by the way human developers diagnose failures, Event Chaining iteratively explains causal dependencies from the failure symptom while aggressively skipping logic likely irrelevant to the failure execution.

This thesis describes the modifications and additions to Event Chaining necessary to enable Pensieve’s goal of automatic failure reproduction. Redundancy-Avoidant Event Chaining enables Event Chaining to scale for complex failures executions by eliminating duplicate explanations of equivalent events, thus preventing the combinatorial explosion problem inherent to the naïve Event Chaining algorithm. Search restrictions aimed at producing most minimal unit tests further reduce the search space of the analysis. This thesis also presents Pensieve-D, a dynamic phase that enables Pensieve to produce reliable and deterministic reproductions. Pensieve-D detects inaccuracies from the initial event chains and instructs Event Chaining to refine its analysis to include additional dependencies relevant to the failure. In addition, Pensieve-D infers timing dependencies between parallel threads/processes and enforces them at run time to enable the deterministic reproduction of concurrency failures.
Bibliography


