# The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure

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

Kirk Rodrigues
University of Toronto
kirk.rodrigues@mail.utoronto.ca

Yu Luo
University of Toronto
jack.luo@mail.utoronto.ca

Michael Stumm
University of Toronto
stumm@ece.utoronto.ca

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

## Abstract

The end goal of failure diagnosis is to locate the root cause. Prior root cause localization approaches almost all rely on statistical analysis. This paper proposes taking a different approach based on the observation that if we model an execution as a totally ordered sequence of instructions, then the root cause can be identified by the first instruction where the failure execution deviates from the non-failure execution that has the longest instruction sequence prefix in common with that of the failure execution. Thus, root cause analysis is transformed into a principled search problem to identify the non-failure execution with the longest common prefix. We present Kairux, a tool that does just that. It is, in most cases, capable of pinpointing the root cause of a failure in a distributed system, in a fully automated way. Kairux uses tests from the system's rich unit test suite as building blocks to construct the non-failure execution that has the longest common prefix with the failure execution in order to locate the root cause. By evaluating Kairux on some of the most complex, real-world failures from HBase, HDFS, and ZooKeeper, we show that Kairux can accurately pinpoint each failure's respective root cause.

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

***Keywords*** Failure diagnosis, distributed systems, root cause, debugging

**ACM Reference Format:**
Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3341301.3359650



*SOSP '19, October 27–30, 2019, Huntsville, ON, Canada*

ACM ISBN 978-1-4503-6873-5/19/10.
https://doi.org/10.1145/3341301.3359650

## 1 Introduction

Postmortem failure diagnosis is possibly the most critical yet time consuming software engineering task today. Studies have shown that the software industry is spending more than $100 billion on failure diagnosis [41], and programmers are spending over 50% of their time debugging [32]. The end goal for postmortem debugging is to locate the root cause of the failure, or the fault [26]. The *root cause* is the most basic reason for a failure which, if corrected, would have prevented the failure from occurring [47].

There are two major tasks towards locating the root cause: the first is to reconstruct a failure execution, and the second is to then find the actual cause of the failure. Reconstruction of failure executions has been well studied, and the trade-offs between comprehensiveness and intrusiveness are well understood. For example, deterministic replay tools [2, 3, 12, 14, 15, 29, 35, 36, 40, 42, 44] provide comprehensive execution traces, yet are heavily intrusive; tracing [5, 6, 17, 30] is less intrusive but provides a less comprehensive trace; non-intrusive approaches reconstruct partial execution traces from logs [50, 56–58], the core-dump [31, 52], and hardware support such as Intel PT [13, 25].

Reconstructing a failure execution path provides a critical first step towards identifying the root cause. However, identifying the cause for the failure from the failure execution path still often needs to be done manually today. It is a daunting task, as the number of events involved in the failure path is beyond the processing power of humans given the speed of today's processors and the scale of today's distributed systems. Even with tools that aim to reconstruct a minimal partial trace, the number of events involved is still large. For example, Pensieve [56] is able to reconstruct a minimal event chain that contains only the events that must have occurred to lead to the failure, yet this chain can still contain hundreds of events for a typical failure on today's distributed systems.

Almost all existing approaches for fault localization use a probabilistic approach [7, 16, 24, 25, 27, 28]. They infer the

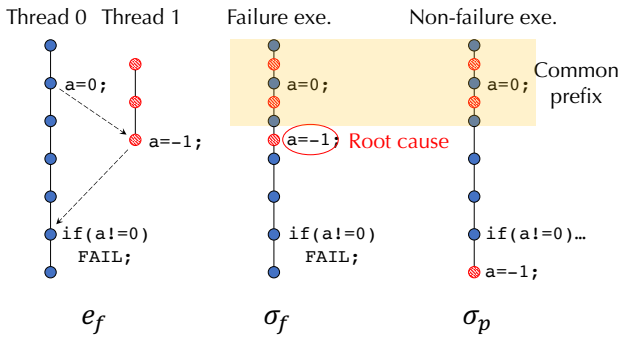


**Figure 1.** An abstract failure example. § 2 shows a real-world data race on HDFS that has a similar pattern.

predicates (e.g., branch conditions or whether a function return value is 0) that have the strongest statistical correlation with the failure execution. While such approaches can be effective, the outcome is also fundamentally probabilistic. It also does not include the execution context that explains why the predicate has occurred, which is typically needed by developers to understand the root cause. In addition, certain bugs are notoriously difficult to track down using statistical approaches; in particular, recording the ordering of events that may lead to data races can become a combinatorial problem.

We propose a new approach towards identifying failure causes which is substantially different from previous approaches. It is based on the *Inflection Point Hypothesis* we introduce in this paper. Suppose we have a failure execution from which we extract an instruction sequence, $\sigma_f$, that leads to the failure. We compare $\sigma_f$ against every instruction sequence belonging to the set $\Sigma_v$ of all failure-free instruction sequences that produce the correct result. We select the instruction sequence $\sigma_p$ in $\Sigma_v$ that has the longest common prefix with $\sigma_f$. We define the ***inflection point*** of $\sigma_f$ to be the earliest position in $\sigma_f$ where $\sigma_f$ and $\sigma_p$ deviate. The hypothesis states that the instruction at the inflection point is the root cause of the failure. That point in $\sigma_f$ is called an inflection point because from that point on, any subsequent execution path cannot possibly *not* fail.

As an abstract example, consider a failure caused by a read-after-write data race. Figure 1 depicts an execution $e_f$, where thread 1 modifies `a` to be -1, ultimately triggering a failure in thread 0. Bugs like these are often notoriously difficult to debug, especially when there is a long propagation from the write (`a=-1`) to the read (`if(a!=0)`). The middle of the figure shows a possible instruction sequence, $\sigma_f$, obtained from $e_f$, that contains instructions from both threads. If we enumerate all possible instruction sequences obtained from all possible non-failure executions that produce correct results, and we compare each such sequence against $\sigma_f$, then we should find one, $\sigma_p$ (shown on the right of the figure), that has the longest common prefix with $\sigma_f$. According to the Inflection Point Hypothesis, the inflection point of the failure is located at the first instruction in $\sigma_f$ that differs from $\sigma_p$, namely `a=-1` in thread 1. Intuitively, it is clear there can be no valid, failure-free instruction sequence $\sigma_v$ that has a longer common prefix with $\sigma_f$: any sequence that includes `a=-1` as the next instruction will form the same read-after-write dependency that leads to the failure, and any instruction sequence that has `if(a!=0)` before `a=-1` will have a shorter prefix in common with $\sigma_f$.

This simple hypothesis is powerful. It transforms root cause analysis into a principled search problem. The root cause of the failure is identified by systematically searching for the valid non-failure instruction sequence $\sigma_p$ that has the longest common prefix with the failure instruction sequence. This is a fundamentally different approach to root cause analysis as it suggests that the search for the root cause should start from the beginning of the execution. Conventional wisdom has it that the root cause is likely close to the failure, and as a result, many tools use a backward search strategy [13, 25, 43]. While these techniques can be effective in diagnosing many failures, their effectiveness is limited when the propagation from fault to the failure is long. Such failures are also the most difficult to debug.

The search for the inflection point (and hence the root cause), as described above, is clearly impractical since any real distributed system would have infinitely many valid instruction sequences that do not lead to failure. However, by carefully selecting non-failing instruction sequences, it is possible to heuristically search for the non-failing instruction sequence that has the longest prefix in common with the failed instruction sequence in a tractable way with a high success rate. We have designed and implemented a tool, Kairux, capable of locating the root causes of most distributed system failures in a fully automated manner. Kairux takes three inputs: (1) the steps to reproduce the failure, typically packaged in a unit test [56]; (2) the failure symptom; and (3) all the code's unit tests. Kairux outputs: (1) the inflection point; (2) the non-failure instruction sequence $\sigma_p$ having the longest common prefix with $\sigma_f$; and (3) the steps needed to reproduce $\sigma_p$ in the form of a unit test.

Following the Inflection Point Hypothesis, Kairux's central task is to find $\sigma_p$, the non-failure instruction sequence that has the longest common prefix. However, instead of trying to enumerate all possible failure-free sequences, Kairux only considers failure-free sequences obtained from the system's existing unit tests[1] and quickly discards all instruction sequences unrelated to the failure sequence. Real systems typically have a rich unit test suite [49]—many enforce a code commit policy whereby every code change must be accompanied either by a new unit test or by a change to an

[1]We use the term "unit test" as used in many open source projects even though they include function tests, component tests, integration tests, etc. as well.

$e_i$, $e_f$, ...: executions
$\sigma_x$, $\sigma_y$, ...: instruction sequences
$\sigma_f$: instruction sequence leading to failure
$\sigma_p$: non-failure instruction sequence with longest prefix
$\Sigma_x$, $\Sigma_y$, ...: set of instruction sequences

**Table 1.** Notation used in paper

| System | # tests | class cov. | func. cov. | stmt. cov. |
|---|---|---|---|---|
| Cassandra 3.11.4 | 3,229 | 88.2% | 76.8% | 73.6% |
| HDFS 3.1.2 | 7,846 | 87.7% | 90.4% | 90.1% |
| ZooKeeper 3.4.11 | 554 | 90.6% | 90.3% | 88.4% |

**Table 2.** Number of unit tests and their coverage on three popular distributed systems.

existing one. Consequently, as shown in Table 2, unit tests achieve high coverage. On average, 86% of the functions are covered by unit tests.

Kairux applies a number of techniques in its attempt to construct $\sigma_p$ in a tractable way. First, it removes instructions from the sequences causally unrelated to the failure symptom. Therefore, it only operates on partially ordered sequences of instructions instead of totally ordered ones. It then separates target instruction sequences into separate subsequences belonging to different threads and initially processes each one independently. Kairux does this because (1) failures in real distributed systems typically require that multiple operations be triggered [49]; (2) each operation is typically processed by its own independent thread; and (3) in most cases, each unit test only exercises a few operations. Further, Kairux uses a heuristic to prioritize the unit tests most likely to have instruction sequences in common with the failure execution. When a test instruction sequence diverges from the failure sequence, Kairux attempts to modify the target unit test's input parameters in an attempt to reduce the divergence. Finally, it splices together subsequences from multiple threads and multiple tests to construct $\sigma_p$. Details on Kairux's design are provided in § 4.

We evaluated Kairux on 10 randomly sampled real-world failures reported in Hadoop, HBase, and ZooKeeper. Kairux can accurately locate the root cause in 7 of them. Some of these failures are among the ones that are the most difficult to debug, such as data races.

This work has a number of limitations. First, Kairux can only locate the root cause if there is a deviation in the instruction sequence between the failure and non-failure executions. For some failures, a failure execution may produce the wrong output using the same sequence of instructions as a non-failure execution, and in that case Kairux will not be able to detect the root cause. However, we should point out that this does not mean that Kairux cannot detect failures caused by incorrect data-flow. Incorrect data-flow typically results in control-flow deviation as the data is used in branch conditions. In fact, the data race example in Figure 1 is caused by an incorrect data-flow. Second, Kairux only locates the root cause. It is still the developers' responsibility to find a fix for the bug, which can be time-consuming in and of itself even after the failure has been diagnosed. Finally, Kairux requires the failure to be reproducible. However, as mentioned earlier, failure reproduction is well explored. For example, Pensieve [56] is capable of inferring the sequence of commands, packaged in a unit test, that can reproduce the failure by analyzing the log and program code. Many bug reporting systems such as the ones for Firefox and Chrome require the reporter to provide steps to reproduce the bug.

## 2 Motivating Real-world Example

We use a real-world failure as a motivating example to explain the Inflection Point Hypothesis and how Kairux works. The specific real-world failure we consider is from HDFS: HDFS-10453 [22]. A user frequently observed that after a data node is decommissioned, (*i*) HDFS fails to replicate any data blocks that are under-replicated, (*ii*) HDFS outputs an exception log statement that complains there is not enough capacity on any data node, and (*iii*) the thread that is responsible for block replication, `ReplicationMonitor`, freezes. This failure was extremely difficult to debug, taking the user a month to track down the root cause [20, 21]; the user had to do multiple rounds of "printf-debugging": instrumenting the system with new log-printing statements, waiting for the failure to occur, and then reading the log only to find it is necessary to add new log printing statements. When the user finally identified the root cause, he wrote a detailed blog post describing this frustrating debugging experience [21].

Figure 2 shows the simplified code snippet for HDFS-10453. The symptom is at line 28, where the exception occurs and leads to the printing of an error log message (not shown in the figure). The loop above it at line 19 attempts to replicate an under-replicated block `numNeeded` times. It does so by randomly selecting a data node and replicating the block if the node's capacity is larger than the size of the block. The exception occurs because the loop failed to replicate the block. Because this loop checks every data node in the system, it takes a long time to complete. In the meantime, the `neededReplications` queue quickly grows large, and it appears that the `ReplicationMonitor` thread freezes and stops replicating any under-replicated blocks.

Figure 3 shows part of the failure execution's instruction sequence. Kairux is able to automatically identify the inflection point of this failure as line 5 according to the Inflection Point Hypothesis (and it would be able to do so whenever lines 33 and 34 are interleaved by line 5 in the execution sequence). The inflection point (line 5), its cause (data-flow from line 33), and its consequences (the branch condition at line 22 is not satisfied) are included in Kairux's analysis

```
1 /****** ReplicationMonitor thread ******/
2 void replicateBlocks() {
3   while(namesystem.isRunning()) {
4     for (Block b : neededReplications)
5       chooseTarget(b.numReplicas,b.size);
6     Thread.sleep(...);
7   }
8 }
9
10 void chooseTarget(int nReplicas,long bSize){
11   if (nReplicas <= 1)
12     chooseRemoteRack(nReplicas, bSize);
13   else
14     chooseRandom(nReplicas, bSize);
15 }
16
17 void chooseRandom(int nReplica,long blkSize){
18   int numNeeded=REPLICATION_FACTOR-nReplica;
19   while(numNeeded > 0) { // replicas needed
20     node = chooseNextRandomNode(..);
21     if (node==NULL) break;//Checked all
22     if (blkSize <= node.capacity) {
23       ... // replicate the block to this node
24       numNeeded--;
25     }
26   }
27   if (numNeeded > 0)
28     throw new NotEnoughReplicasException();
29 }
30
31 /******** DeleteBlock thread *********/
32 void deleteBlock(Block blk) {
33   blk.size = Long.MAX_VALUE; // modified!
34   neededReplications.remove(blk);
35 }
```

**Figure 2.** Simplified code snippet for HDFS-10453.

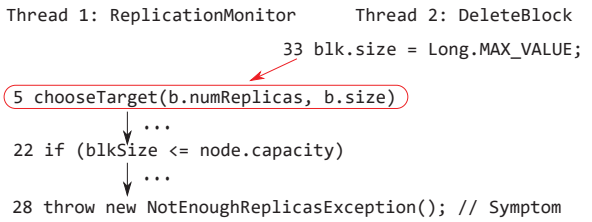


**Figure 3.** The root cause, highlighted in red, inferred by Kairux for the HDFS failure.

report. This clearly explains this bug. It is a data race similar to the abstract failure example we showed in Figure 1: decommissioning a data node triggers a delete operation on a block that in this case is under-replicated (line 32). The data node sets the block's size to `Long.MAX_VALUE` (line 33) because HDFS uses the value `Long.MAX_VALUE` to indicate that the block is being deleted. At the same time, a background `ReplicationMonitor` thread is trying to replicate this block. It attempts to find a data node whose capacity is larger than the size of the block (line 22), but since the block's size is set to `Long.MAX_VALUE`, it can never find a data node with enough capacity.

# 3 The Inflection Point Hypothesis

## 3.1 Definitions and Assumptions

We consider an execution of a software system by a series of (API) input commands. We model the execution as a totally ordered *instruction sequence* obtained by interleaving the instructions executed by each thread, process, and node in the system, taking constraints from existing synchronization into account. We use total order of instructions in this section to simplify the definitions and explanations. Our tool, Kairux, operates only on a partially ordered instruction sequence obtained by dynamic slicing (§ 4.2). We say two executions are the same if they have the exact same instruction sequence. For a (reproduced) failure execution, we consider any instruction sequence obtained from the failed execution. For each non-failure execution, we consider (in theory) every possible interleaving as a separate instruction sequence.

We refer to an execution as being *correct* when, given a series of input commands, the execution produces results according to the system's specification. When referring to "non-failure executions," we always assume these executions are correct. The term "correct execution" is used more weakly in practice; for example, after fixing a bug having caused a failure, the term may be used to say that the execution no longer fails and still passes all of the unit tests. We generally assume the system under consideration has been reasonably well tested and that there exist non-failing unit tests for the system. We do so to avoid having to consider pathological cases that have no non-failure executions (such as programs that fail on the first instruction regardless of input).

The *root cause* of a failure is, intuitively, the condition that developers should correct. A number of studies have provided definitions for root cause [47, 51]. We adopt the one from Wilson *et al.* [47] as it is one of the most precise and widely adopted definitions [46]: *"Root cause is that most basic reason for an undesirable condition or problem which, if eliminated or corrected, would have prevented it from existing or occurring."*

We now refine the definition to relate it to the context of computer systems. We first define *root cause candidate*:

**Definition 1.** *A **root cause candidate** is the location of an instruction in the failure execution sequence where changing this instruction results in correct executions that avoid the associated failure.*

A failure could have multiple root cause candidates. For example, a failure can be prevented by either disabling the software component that contains the underlying bug through a configuration change or fixing the bug itself. Therefore, the location of the first instruction that is executed in the

true branch of `if (ENABLED(X))` and the location of the instruction that corresponds to the bug are both root cause candidates of the failure.

The original definition of root cause further constrains the root cause to be the one that is the *most basic*. Fixing the bug is more basic than changing the configuration. Less formally, the root cause candidate that is located further down in the failure execution is more basic because if changed it affects a smaller portion of the execution. This leads to our definition of $\rho$-cause:

**Definition 2.** *The **$\rho$-cause** of a failure is the root cause candidate that has the longest distance from the first instruction of the failure execution.*

Note that the $\rho$-cause of a failure may or may not be the root cause. Identifying the root cause is fundamentally a subjective exercise and it may even be ambiguous. For example, a system administrator or support engineer may argue that identifying the high-level configuration that can be used to disable the buggy software component is more meaningful. In comparison, $\rho$-cause is well-defined and unambiguous. In our experiment, we found that for all of the real-world failures where Kairux successfully identified the $\rho$-causes, the $\rho$-cause was the same as the root cause. However, in theory there can be cases where the root cause is not the $\rho$-cause. We discuss them in § 3.4 and show that even for the cases where the $\rho$-causes are not the root causes, the root cause can be easily identified once the $\rho$-cause is located.

We also assume $\Sigma_v \neq \emptyset$ is the set of all possible non-failure instruction sequences that could be executed by the target system. Obviously $\Sigma_v$ includes all the test run executions if the system has a perfectly thorough test suite (i.e., achieving 100% coverage in any testing coverage criteria).

Finally, we define *inflection point*:

**Definition 3.** *The **inflection point** in execution $\sigma_x$ when compared to execution $\sigma_y$ is the location of the first instruction in $\sigma_x$ that differs from the instruction at the same point in $\sigma_y$.*

### 3.2 Inflection Point Hypothesis

We can now formally introduce the Inflection Point Hypothesis. We first introduce a theorem.

**Theorem 1.** *The $\rho$-cause of a failure execution $\sigma_f$ is located at its inflection point when compared to the execution $\sigma_p$, where $\sigma_p$ is the execution in $\Sigma_v$ that has the longest common prefix with $\sigma_f$.*

This theorem follows naturally from the definitions above. Consider an instruction sequence that leads to a failure, $\sigma_f$, and its associated $\sigma_p \in \Sigma_v$ which has the longest common prefix with $\sigma_f$. We first note that by definition, there can be only one inflection point for the failure execution $\sigma_f$. Let $r$ be the inflection point and $s$ be the $\rho$-cause. Clearly, $r$ is a root cause candidate because we can change $r$ in $\sigma_f$ to be the instruction in $\sigma_p$ so that it leads to a correct execution $\sigma_p$ that avoids the failure. Then by Definition 2, the root cause candidate $s$ must occur after or be equal to $r$. However, it cannot be after because otherwise we would have found an instruction sequence with a longer common prefix. Therefore, $r = s$.

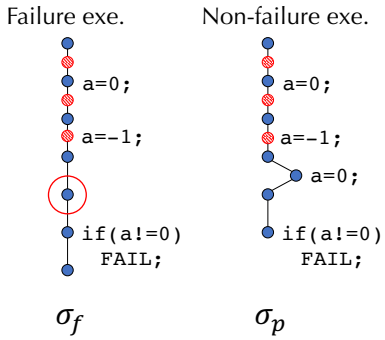


**Figure 4.** An example showing that the root cause may not be the root cause candidate. The same failure execution as in Figure 1 is on the left. The right non-failure sequence further overwrites `a` with 0. The circled instruction is the inflection point that is identified as the root cause candidate by our definition.

**Inflection Point Hypothesis.** *The root cause of a failure is located at its inflection point identified in Theorem 1.*

In other words, the hypothesis states that the $\rho$-cause of a failure is the root cause. This is because Theorem 1 already states that the $\rho$-cause is located at the inflection point.

### 3.3 HDFS Failure Example Revisited

Referring back to Figures 2 and 3, the Inflection Point Hypothesis captures the root cause as line 5 because we will be able to find non-failure executions that overlap with the failure instruction sequence $\sigma_f = (\ldots, 33, 5, \ldots, 22, \ldots, 28\ldots)$ up to line 33, but not beyond. For example, there is a non-failure execution sequence $\sigma_i = (\ldots, 33, 34, 5, \ldots)$, which avoids a failure, because with line 34 ahead of line 5, the block will be removed from the shared `neededReplications` queue and line 5 does not lead to failure. Hence, line 5 is the point of inflection: there is no non-failure execution that has a longer overlap, because once line 5 is scheduled after line 33 (and before line 34), the failure is inevitable.

### 3.4 Caveats

For many failures, there are multiple root cause candidates. Picking the one root cause can be fundamentally subjective. Our definition of $\rho$-cause picks the one that comes last and that is what the Inflection Point Hypothesis identifies as the root cause. However, cases can be made that the other causes are better choices.

As an example, our hypothesis locates the root cause in the instruction sequence while, in some cases, one could argue that the root cause is a failure-inducing input. For example,

for failures that are caused by misconfigurations, the root cause should be the wrong configuration value. We note, however, that in this case, as long as the wrong configuration value is used in a branch condition, our methodology will identify the first instruction that is executed after this branch instruction as the location of the $\rho$-cause. The user can still pinpoint the wrong configuration from the variable used in the branch condition that resulted in the point of inflection.

Figure 4 shows a more subtle, contrived example that demonstrates $\rho$-cause may not always capture the underlying root cause. In this example, the failure execution is the same as the one in Figure 1. However, the difference is that now there can be a non-failure execution that overwrites `a` with `0` in thread `0` after thread `1` (the red thread) writes `-1` to `a`. This non-failure execution will have a longer common prefix with the failure execution, so according to our definition, the $\rho$-cause is not the data race from the red thread that writes `a=-1`. Instead, the $\rho$-cause is at the instruction that took the place of the instruction that overwrites `a` back to `0`. While one could argue that this indeed can be the root cause since the failure can be avoided by forcing the overwrite to occur in the original failure execution, it does not fix the underlying data race bug. However, the data race that is the root cause can be easily identified from understanding the inflection point (where `a` is not overwritten to `0` in the failure execution) and examining the data-flow of `a`.

Finally, a failure can have multiple underlying causes. A common example is a bug or a user misconfiguration that triggers an exception, and the exception handling logic has another bug. Many of the most catastrophic software failures had this pattern [37, 39]. Fixing any of the bugs could prevent the failure from occurring, but ideally all should be fixed. However, in these cases, our definition and hypothesis will only identify the last bug as the root cause, i.e., the bug in the error handling logic.

# 4 Design of Kairux

This section explains how Kairux works in general as well as how it works when applied to the HDFS failure (§ 2) in particular. We start by describing some of the challenges in turning the Inflection Point Hypothesis into a practical and tractable solution.

## 4.1 Challenges and Ideas

If the set of all possible non-failure instruction sequences $\Sigma_v$ is available, then theoretically we could search for the sequence $\sigma_p$ that has the longest common prefix with the failure sequence $\sigma_f$ and locate the point of divergence as the root cause candidate. In practice, it is not possible to obtain $\Sigma_v$ for any reasonably complex system as there will be infinitely many non-failure executions. One approach we considered was symbolic execution which, in principle, can explore all possible execution paths [9, 52]; but we rejected this approach due to its difficulties in scaling to complex distributed systems [56].

Instead, Kairux uses the rich unit test suite that is an integral part of any real system to obtain a set of non-failure executions, and then uses the instruction sequences from test executions as building blocks to construct $\sigma_p$. As shown in Table 2 in § 1, unit tests for distributed systems cover over 86% of the functions, so the functions in $\sigma_f$ are likely to be exercised by the tests. In addition, Kairux uses only partial order instead of total order when comparing the common prefix between failure and non-failure instruction sequences; this is to eliminate the large number of possible instruction sequences caused by non-determinism.

However, in most cases, we will not be able to find a unit test with an instruction sequence that closely matches $\sigma_f$. For example, $\sigma_f$ may involve multiple operations (like in the HDFS example), yet each unit test typically only exercises a few operations that are likely to be a subset of the required operations. Hence, it may be necessary to combine instruction sequences from multiple tests in order to construct $\sigma_p$.

As another example, a test execution $\sigma_t$ may deviate from $\sigma_f$, yet the deviation could be oblivious to the root cause. Such a deviation may be caused by a configuration value, a different thread scheduling, or a different branch direction that is not related to the root cause. Hence, when $\sigma_t$ deviates from $\sigma_f$ at a point $d$, Kairux checks if the deviation is failure oblivious. It does so by modifying the test execution to make it take the same execution path as $\sigma_f$ at $d$. If the modified test execution, $\sigma_{t'}$, still does *not* lead to the failure, the deviation at $d$ is failure oblivious, and we have constructed a non-failure execution $\sigma_{t'}$ that is closer to $\sigma_f$ than $\sigma_t$. On the other hand, if after the modification $\sigma_{t'}$ results in failure, then $d$ could be a root cause candidate.

## 4.2 Kairux Algorithm Overview

Algorithm 1 shows the algorithm used by Kairux to identify the root cause candidate. The inputs to Kairux are: (1) the failure execution $\sigma_f$; (2) the unit test suite, and (3) the failure symptom, which is a point in the program execution. The symptom can be further associated with constraints, e.g., variable `a` must be less than `0` at this point. Only when the failure program point is reached, and the constraints are satisfied, will Kairux conclude that the failure has occurred. The output is the inflection point $P$ and the non-failure execution's instruction sequence $s_p$ that has the longest common prefix with $\sigma_f$. ($s_p$ is a subsequence of the instruction sequence $\sigma_p$.) Kairux also packages steps to reproduce $s_p$ and $\sigma_p$ into a unit test. Initially, $P$ is set to the location of the symptom and $s_p$ is set to empty.

Kairux first obtains the dynamic program slice $\overline{s_f}$ of the failure symptom from $\sigma_f$ on lines 2–4. A *dynamic program slice* of the symptom consists of the subsequence of instructions in $\sigma_f$ on which the symptom is causally dependent [1, 45]. In other words, if the symptom occurs regardless of

**Algorithm 1:** Design of Kairux.

**Input** : $\sigma_f$: failure execution; unit tests; symptom
**Output**: P: inflection point; $s_p$: instruction sequence with longest prefix

```
1  s_p = [], P = symptom;
2  locations = static_slicing(symptom);
3  s_f = run_and_record(σ_f, locations);
4  s̄_f = dynamic_slicing(s_f, symptom);
5  ⟨s̄_f|t_0, s̄_f|t_1, .., s̄_f|t_s⟩ = group_by_thread(s̄_f);
6  thread_queue = [⟨s_f|t_s, symptom⟩];
7  while thread_queue≠ ∅ do
8      ⟨s̄_f|t, pt⟩ =thread_queue.dequeue();
9      s_p|t = ∅;
10     sorted_tests = sort_tests(s̄_f|t, pt);
11     foreach u in the first N tests of sorted_tests do
12         s_u = run_and_record(u, locations);
13         ⟨s_u'|t, prefix⟩ =
             compare_and_modify(s̄_f|t, s_u|t, u);
14         if prefix.length > s_p|t.length then
15             s_p|t = s_u'|t;
16         end
17     end
18     s_p = merge(s_p, s_p|t);
19     foreach read-after-write dependency w|t' → r|t
         do
20         thread_queue.enqueue(⟨s̄_f|t', w|t'⟩);
21     end
22 end
23 P = find_deviation(s̄_f,s_p);
```

$s_f$: subsequence of $\sigma_f$ recorded at static slice locations
$\overline{s_f}$: dynamic slice of the failure execution
$\overline{s_f}|t$: subsequence of $\overline{s_f}$ that is from thread $t$
$u$: a unit test
$s_u$: subsequence of $\sigma_u$ recorded at static slice locations
$s_u|t$: subsequence of $s_u$ that is from thread $t$
$t_s$: the thread that contains the symptom
$s_p|t$: subsequence of $s_p$ that is from thread $t$

**Table 3.** Symbols used in Algorithm 1.

whether an instruction $i$ executes or executes differently, then $i$ is not in the program slice. Moreover, only instructions that occur before the symptom can be in $\overline{s_f}$. A slice is called *dynamic* because it only contains instructions that actually occurred in the execution. For example, line 24 in Figure 2 is not in the dynamic slice for the HDFS failure because it is not executed in the failure execution, even though the symptom would be causally dependent on it if it were executed. Kairux only operates on $\overline{s_f}$ instead of the complete instruction sequence in $\sigma_f$ so that it can ignore the parts that are oblivious to the failure, e.g., failure-irrelevant threads.

In theory, we can obtain the dynamic program slice by recording a trace of every instruction that is executed and then inferring the slice from this trace. In practice, however, doing so has high overhead that can be prohibitive. Therefore, Kairux first uses static analysis to obtain the static program slice of the symptom, which includes only those instructions that *may* have a causal dependency on the program location of the symptom [45]. Obtaining the static slice is an iterative process. At the start, only the symptom's instruction is in the slice. We then work backwards, iteratively analyzing the control- and data-flow of each instruction in the slice to add more instructions into the slice. For the HDFS failure example in Figure 2, starting from the symptom at line 28, Kairux follows the control-flow to infer that line 27 is part of the slice; Kairux further analyzes the data-flow of `numNeeded`, because it is used as a branch condition variable at line 27, and adds line 24 to the slice. By repeating this process, every statement in Figure 2 will be included in the static slice. The static slice will be a super-set of the instructions that belong in the dynamic slice of any failure execution.

Kairux then sets a breakpoint at each program location in the static slice and reproduces the failure. It records each breakpoint that was hit to obtain a trace $s_f$ (line 3 in Algorithm 1) and then performs a similar dependency analysis on $s_f$ to obtain the dynamic slice. Kairux acquires the dynamic slice across the network by annotating network communication libraries, such as Google Protocol Buffers [23] and Apache Thrift [18], in a manner similar to Pensieve [56]. We use the bar in $\overline{s_f}$ to indicate that it is a dynamic slice of $s_f$.

Kairux further separates $\overline{s_f}$ into different subsequences, each belonging to a separate thread (line 5). We use $\overline{s_f}|t$ to represent the instruction subsequence in $\overline{s_f}$ that belongs to thread $t$. Breaking $\overline{s_f}$ into subsequences by thread allows us to effectively use unit test executions to compose $s_p$. Each unit test typically only exercises a few operations, each of which is often processed by one thread; yet $s_p$ typically requires a specific set of operations. It is unlikely that there exists a unit test that contains all of the required operations, but it is likely that each required operation is processed by at least one unit test.

Kairux uses a `thread_queue` to iteratively analyze each thread in $\overline{s_f}$. Each element in the queue is a tuple $\langle \overline{s_f}|t, pt\rangle$, where $pt$ is the location of an instruction in $\overline{s_f}|t$. At the start, `thread_queue` only contains $\langle \overline{s_f}|t_s, symptom\rangle$ where $t_s$ is the thread that contains the failure symptom.

Each iteration of the loop at line 7 processes the subsequence $\overline{s_f}|t$ belonging to thread $t$. It attempts to construct the part of $s_p$, $s_p|t$ (initialized to empty at line 9), that matches the longest prefix in $\overline{s_f}|t$ up to the instruction at $pt$. To do so, it first ranks all of the unit tests by their similarity to $\overline{s_f}|t$ in `sort_tests()` at line 10. The sort uses a simple ranking algorithm where each unit test is ranked by the number of

executed functions it has in common with the functions on the stack when the failure execution $\overline{s_f}|t$ reaches $pt$. In the HDFS failure example, three functions are on the stack when the symptom occurs: `chooseRandom()`, `chooseTarget()`, and `replicateBlocks()`. A unit test will be ranked highest if it executes all three functions.

We have found this simple ranking algorithm to be quite effective. It focuses on the most important functions leading to the symptom, yet allows tests with small deviations to still achieve a high rank. Kairux only analyzes the top $N$ ranked tests for performance reasons ($N = 100$ in our experiment).

For each highly ranked unit test, Kairux first runs it to obtain a trace $s_u$ (line 12), recording only those instructions that are in the failure's static slice (obtained at line 2). Recording only the same instructions allows Kairux to compare the two instruction sequences $\overline{s_f}|t$ and $s_u|t$, where $s_u|t$ is the instruction subsequence in $s_u$ that belongs to thread $t$ (as $s_u$ might also contain multiple threads).

Given $\overline{s_f}|t$ and $s_u|t$, `compare_and_modify()` at line 13 compares the two instruction sequences to find their common prefix. It also tries to modify the test $u$ to produce a non-failure execution that has longer common prefix to $\overline{s_f}|t$ than $s_u|t$. We explain in § 4.3 how this function works in detail. `compare_and_modify()` returns a tuple: a new instruction subsequence $s_{u'}|t$ obtained by modifying $u$, and the common prefix between $s_{u'}|t$ and $\overline{s_f}|t$. It also returns the modified unit test $u'$ that can reproduce $s_{u'}|t$ so that $u'$ can be used to compose the unit test that produces $s_p$. If the common prefix obtained by modifying $u$ is the longest at this point (line 14), Kairux then updates $s_p|t$ to be this prefix.

After analyzing the top $N$ unit tests, $s_p|t$ contains the non-failure instruction subsequence found with the longest prefix common with $s_f|t$. Kairux then merges $s_p|t$ into $s_p$ to form a longer $s_p$ (line 18 in Algorithm 1). At the start, when $s_p$ is empty, $s_p$ simply becomes $s_p|t$. If $s_p$ already has instruction sequences from other threads, then Kairux merges the two instruction sequences by trying to follow the same interleaving as in the failure execution $s_f$. Note that sometimes enforcing a particular interleaving will result in a failure, and Kairux then has to change the interleaving to produce a non-failure execution in $s_p$. In the HDFS example, after the first iteration of the `while` loop of Algorithm 1, $s_p$ will have the instruction sequence from the `ReplicationMonitor` thread (the sequence shown on the left in Figure 3). In the second iteration Kairux will further construct $s_p|t$ from the `DeleteBlock` thread. When merging this $s_p|t$ into $s_p$, Kairux finds that if it enforces the same interleaving from line 33 to line 5 as shown in Figure 3, the failure will occur, so that it has to schedule line 33 after line 5 to produce a non-failure execution $s_p$.

After analyzing thread $t$'s instruction subsequence, Kairux checks to see if there is any data-flow dependency between $\overline{s_f}|t$ and another thread $t'$. If so, then the source write instruction of the data-flow $w|t'$ is already in the dynamic slice $\overline{s_f}$. Kairux then adds the tuple $\langle \overline{s_f}|t', w|t' \rangle$ to `thread_queue`, and repeats the same analysis again on $\overline{s_f}|t'$. Eventually, Kairux will construct a non-failure execution $s_p$, and the inflection point $P$ will simply be the first point where $\overline{s_f}$ deviates from $s_p$.

**Applying the algorithm on the HDFS example.** We now explain how Kairux works on the HDFS example in Figure 2. Kairux first constructs the dynamic slice $\overline{s_f}$. There are two threads, `ReplicationMonitor` and `DeleteBlock`, in $\overline{s_f}$, and Kairux separates $\overline{s_f}$ into two instruction subsequences. It first analyzes the $\overline{s_f}|t$ for `ReplicationMonitor`, because it contains the symptom, and finds there is a unit test `testChangeColdRep` that is ranked highest since it executes every function that is on the stack when the symptom occurs. The simplified code of the test is as follows:

```
short replication_factor = 3;
createFile("/foo" replication_factor);
setReplication("/foo", 5);
/* Wait for the replication */
```

It first creates a file with replication factor set to 3. It then increases the factor to 5, which triggers the `ReplicationMonitor` thread to replicate this block. The test then waits for the replication to complete and asserts that there are indeed 5 replicas of this block.

Kairux's `compare_and_modify()` finds that the $s_u|t$ of this unit test has a common prefix with $\overline{s_f}|t$ from the beginning to the branch at line 22 in Figure 2, where the two executions deviate. It cannot find a modification to this unit test to obtain a non-failure execution that has a longer common prefix, so the $s'_u|t$ is the same as $s_u|t$.

Kairux then identifies a data-flow dependency between thread `ReplicationMonitor` and another thread (`DeleteBlock`): line 33 → line 5. It thus adds a new tuple $\langle \overline{s_f}|t',$ line 33$\rangle$ into `thread_queue`, where $t'$ is `DeleteBlock`. This time Kairux finds another test, `testRemove`, that executes `DeleteBlock` and matches every instruction in $\overline{s_f}|t'$, without the need for modification. When Kairux tries to merge $s_p|t'$, which contains instructions from `DeleteBlock`, with $s_p$, which contains instructions from `ReplicationMonitor`, it finds that enforcing the same interleaving from line 33 to line 5 will lead to failure. Therefore, the $s_p$ it produces has line 33 scheduled to be after line 5. Thus, the point where $s_p$ and $\overline{s_f}$ deviate is at line 5, which Kairux declares to be the inflection point and hence the root cause candidate. Kairux also produces a unit test:

```
short replication_factor = 3;
createFile("/foo" replication_factor);
setReplication("/foo", 5);
deleteFile("/foo");
```

that can be used to reproduce the non-failure execution $s_p$. Note that $s_p$ also requires a particular interleaving among threads; such timing is further enforced by executing the unit

**Algorithm 2:** `compare_and_modify`.

**Input** : $\overline{s_f}|t, s_u|t, u$
**Output**: $s_{u'}|t$, prefix, $u'$: modified unit test

```
1  while true do
2      ⟨P_t, prefix⟩ = compare(s_f|t, s_u|t);
3      u' = modify(P_t, u);
4      if u' ≠ ∅ AND σ_u' ∈ Σ_v then
5          s_u|t = s_u'|t;
6          u = u';
7      end
8      else
9          break;
10     end
11 end
```

```
1 int a[3]={1,2,3};
2 void f(int n) {
3   g(1);
4   if (g(n) != 0)
5     FAILURE;
6 }
7 int g(int n) {
8   int i = 0;
9   while (i<n && i<3) {
10    sum += a[i];
11    i++;
12  }
13  if (sum < 3)
14    return -1;
15  return 0;
16 } // end of f1()
```

**Figure 5.** A code example to help explain `compare()`.

test on Kairux. Kairux will set breakpoints at corresponding instructions to control which instruction is scheduled first [56].

### 4.3 Compare and Modify

Algorithm 2 shows the logic of `compare_and_modify()`. It performs two tasks. First, it *compares* $\overline{s_f}|t$ and $s_u|t$ to find the common prefix and identify the point $P_t$ where they diverge. It then attempts to *modify* the unit test $u$ to $u'$, so that $u'$ takes the same path as $\overline{s_f}|t$ at $P_t$ but still represents a non-failure execution (i.e., $\sigma_{u'} \in \Sigma_v$). If it is successful, then $P_t$ is failure oblivious and Kairux has constructed a new non-failure execution that has a longer common prefix. It continues to compare $\overline{s_f}|t$ with this new execution until it finds a divergence point that is not failure oblivious or the two sequences match.

**Compare** Comparing $\overline{s_f}|t$ and $s_u|t$ is complicated by the fact that $\overline{s_f}|t$ is a dynamic slice while $s_u|t$ is a trace obtained by recording at every instruction in the *static* slice. We cannot perform dynamic slicing on $s_u$ because it does not include the symptom. In addition, as we will explain later, the dynamic slice $\overline{s_f}$ may not actually contain the instruction that is the actual inflection point; but we can remedy this by using $s_u$.

We explain the compare algorithm using the code example in Figure 5. The failure symptom is at line 5. Figure 6 shows two instruction sequences: $\overline{s_f}$, which is the dynamic slice

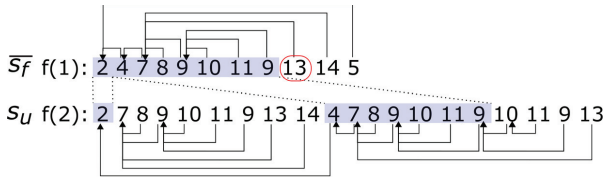


**Figure 6.** $\overline{s_f}$ and $s_u$ for the example in Figure 5. Each instruction is represented by its line number. An arrow between two instructions $i_x \rightarrow i_y$ indicates that the *context* of $i_y$ is $i_x$. The highlighted subsequences are the common prefix between the two as determined by `compare()`.

obtained by issuing the API command `f(1)` and $s_u$, which is the static slice obtained by issuing the command `f(2)`. In this example, `g()` is invoked twice (lines 3 and 4); however, the symptom is only causally dependent on `g()`'s return value in its second invocation. Therefore, the instructions from `g()`'s first invocation are not in $\overline{s_f}$ as they are pruned out during dynamic slicing. In $\overline{s_f}$, the subsequence $(7, 8, ..13)$ stems from the *second* invocation of `g()` at line 4. In $s_u$, however, subsequence $(7, 8, ..13)$ can be found twice; the first instance stems from the first invocation of `g()` and is included because $s_u$ is obtained by recording the execution of every instruction in the static slice. Because of this, a naive lockstep comparison would conclude that the two sequences deviate at the second instruction, when in fact, the second instruction in $\overline{s_f}$ should be matched with $s_u$'s tenth instruction.

To address this, Kairux assigns each instruction in $\overline{s_f}|t$ and $s_u|t$ to a *context* (Figure 6 shows the context of each instruction in $\overline{s_f}$ and $s_u$). The context of an instruction $i$ is another instruction $i'$ that occurred before $i$ and is defined as follows. (1) If $i$ is a function entry, then its context $i'$ is the instruction that invoked the function. For example, the context of the instruction at line 7 in $\overline{s_f}$ is its invocation instruction at line 4. (2) If $i$ is inside of a loop body, then its context $i'$ is the loop guard instruction, i.e., the conditional jump instruction that controls whether to enter the loop. The context of the loop guard instruction itself, starting from the second iteration of the loop, is defined to be the same loop guard instruction from the previous loop iteration. This allows us to use context to precisely follow different iterations of the loop. For example, the context of the instruction at line 9 in $\overline{s_f}$ during the second iteration is line 9 in the first iteration. (3) Otherwise, $i$'s context is the function entry instruction of the function that contains $i$. Note that in Figure 6, the second instruction in $s_u$, 7, does not have a context because its invocation instruction at line 3 is not part of the static slice, thus it is not recorded in $s_u$.

Given contexts, Kairux compares $\overline{s_f}|t$ and $s_u|t$ as follows. At the beginning, the first instructions from $\overline{s_f}|t$ and $s_u|t$ will be treated as matched (i.e., they are the function entry instruction of the thread $t$'s entry function, which is typically

`run()` for Java). Kairux then iterates through each instruction in $\overline{s_f}|t$ and tries to find the next matching instruction in $s_u|t$. Two instructions are treated as a match if and only if (1) they are from the same location in the program's byte code, and thus have the same opcode, and (2) their contexts match. This process stops when Kairux cannot find a matching instruction for $i$ from $\overline{s_f}|t$. For the example in Figure 6, the first 8 highlighted instructions in $\overline{s_f}$ are matched with the 8 highlighted instructions from $s_u$.

Kairux makes an exception to this algorithm when it comes to the handling of *map* loops [56]. Intuitively, a map loop is a loop where Kairux need not enforce that the number of iterations match when comparing $\overline{s_f}|t$ and $s_u|t$. For example, consider the `while` loop at line 3 in the HDFS example in Figure 2. The `chooseTarget()` within this loop gets called in different numbered iterations of this loop in $\sigma_f$ and $\sigma_u$. If we use the algorithm described above, Kairux would stop after detecting that the number of iterations of this `while` loop in $\overline{s_f}|t$ and $s_u|t$ are different. However, we should not be concerned with how many iterations of this loop have occurred before `chooseTarget()` is called.

More precisely, a loop $L$ is considered a *map loop* if it does not have any instruction in its body that uses a variable value that has a *loop-carried data dependency* other than the loop index variable [56]. (With containers in Java, the loop index variable is the container index.) The `while` loop at line 3 in Figure 2 is a map loop because it does not have any instruction which uses a value that depends on previous iterations of this loop. Similarly, the `for` loop at line 4 is also a map loop because `b` has a loop-carried dependency only with the container index. However, the `while` loop in Figure 5 at line 9 is *not* a map loop because `sum` has a loop carried dependency of its value from the previous iteration.

During dynamic slicing, Kairux checks whether a loop is a map loop. If so, it only includes a single iteration of this loop (which is the one eventually leading to the failure). For the HDFS example in Figure 2, Kairux keeps only one iteration for both the `while` loop at line 3 and the `for` loop at line 4 that leads to the failure symptom. However, $s_u|t$ includes instructions from each iteration of the map loops because we do not perform dynamic slicing on it. For each iteration of the map loop, Kairux will assign the context of its loop guard instruction to be the same as the context of the first loop guard, instead of the loop guard instruction from the previous iteration. This allows the instructions of the map loop in $\overline{s_f}|t$ to match the instructions from any of the loop's iterations in $s_u|t$.

Finally, it is possible the dynamic slice $\overline{s_f}$ does not include the actual inflection point. Consider the HDFS example in Figure 2. The dynamic program slice of the failure execution will *not* include the branch instruction at line 22, even though line 22 is executed and is included in the static slice. This is because in $\sigma_f$, the symptom has a control dependency on `if (numNeeded>0)` on line 27, but `numNeeded` is never decremented at line 24, so the dynamic slice will not include line 22 (i.e., line 22 is only included in the dynamic slice if line 24 is executed). Consequently, the comparison concludes that line 28, the symptom, is the point where $\overline{s_f}|t$ and $s_u|t$ deviate, as every instruction up to line 27 can be matched.

Kairux solves this problem by noting that the missing instruction is included in $s_u|t$, as it is just a static slice. After it finds the deviation point $P_t$ when comparing $\overline{s_f}|t$ and $s_u|t$, it performs dynamic slicing on $s_u|t$ (from the test execution) starting from the instruction before $P_t$ to obtain its dynamic program slice $\overline{s_u}|t$. Then Kairux compares $\overline{s_u}|t$ with $s_f|t$ using the same algorithm. (Recall that $s_f|t$ is the static slice before dynamic slicing was performed.) Only, this time the comparison switches the roles of the failure and test instruction sequence: for each instruction in $\overline{s_u}|t$, it matches it to an instruction in $s_f|t$. This comparison will return the point where the two subsequences deviate, $P_t'$, along with the common prefix $prefix'$. Kairux then compares $prefix'$ with $prefix$, the common prefix from the initial comparison, and checks if they are the same. If they are not, Kairux knows that the instructions in $prefix'$ that are not in $prefix$ are the ones that were pruned by dynamic slicing, and adds these instructions into $\overline{s_f}|t$. This process may need to be repeated again because $\overline{s_u}|t$ may also be missing important instructions. It stops when the common prefix no longer changes.

For the HDFS example, after the first round of comparison between $\overline{s_f}|t$ with $s_u|t$, Kairux finds $P_t$ to be at line 28. It takes the instruction before $P_t$, which is the `if` statement at line 27, and performs dynamic slicing on $s_u|t$ from this instruction to obtain $\overline{s_u}|t$. $\overline{s_u}|t$ will include lines 24 and 22. Kairux then compares $\overline{s_u}|t$ with $s_f|t$. The common prefix, $prefix'$, includes line 22, and the inflection point is the instruction after line 22. Kairux adds the new instructions in $prefix'$, including line 22, into $\overline{s_f}|t$, and concludes that the two subsequences deviate at the instruction after line 22.

**Modify.** After `compare()` identifies that $\overline{s_f}|t$ and $s_u|t$ deviate at $P_t$, `modify()` checks whether $P_t$ is failure oblivious. It does so by attempting to modify the unit test $u$ to $u'$, so that $s_{u'}|t$ has the same common prefix as $s_u|t$ but takes the same path as $\overline{s_f}|t$ at $P_t$. If $u'$ produces a non-failure execution, then we have constructed a new non-failure sequence that has a longer common prefix with $\overline{s_f}|t$.

Consider the HDFS example again. If we do not consider the highest ranked unit test `testChangeColdRep`, the second highest ranked unit test is `testReplication`. This unit test can also be used to construct the same $s_p|t$ as the one using `testChangeColdRep`. However, it needs to be modified. Its simplified code is as follows:

```
short replication_factor = (short)1;
createFile ("/f", replication_factor);
setReplication ("/f", replication_factor+1);
/* Wait for the replication */
```

It first creates a file with replication factor set to 1. It then increases the replication factor to 2, which triggers replication of this block.

When comparing `testReplication`'s $s_u|t$ with $\overline{s_f}|t$, it returns the point of deviation as the instruction after the branch at line 11 in Figure 2. The variable `nReplicas` stores the number of replicas of this block. In $s_u|t$, it has the value 1 because only 1 replica is created, therefore it invokes `chooseRemoteRack()` that replicates the block on a data node on a different rack. However, in $\overline{s_f}|t$, the block still has 2 replicas after the data node is decommissioned and they are located on different racks, therefore $\overline{s_f}|t$ takes the path at line 14.

In `modify()`, Kairux first locates the branch condition $c_f$ that $s_u|t$ should satisfy in order to take the same path as in $\overline{s_f}|t$ at $P_t$. In the HDFS example, $c_f$ is `nReplicas > 1` at line 11. It then uses the same algorithm as in Pensieve [56] to modify the unit test $u$ to $u'$ so that (1) $\overline{s_f}|t$ still has a common prefix with $s_{u'}|t$ up to the instruction just before $P_t$, and (2) $c_f$ is satisfied in $s_{u'}|t$. Pensieve takes the failure log as input and constructs a unit test that can be used to reproduce the failure using a static analysis algorithm called the *event chaining algorithm*. However, the first attempt in the event chaining algorithm may not succeed: it may infer a unit test $u_{pensieve}$ that does not lead to the desired failure. It refines $u_{pensieve}$ by identifying the branch condition $c_{test}$ in the unit test that leads to the deviation from the failure, negates it to $\neg c_{test}$, and feeds this to a *refinement* step (§4 in [56]). The refinement step takes three inputs: (1) $u_{pensieve}$, (2) the condition that should be satisfied ($\neg c_{test}$), and (3) the failure log. Kairux then uses the same event chaining algorithm to search for a modified unit test $u'_{pensieve}$ that now satisfies $\neg c_{test}$ that also produces the same log output (so that it reproduces the failure execution).

Kairux uses the same refinement algorithm as in Pensieve. The three inputs now are (1) $u$ (`testReplication` in the HDFS example), (2) $c_f$ (`nReplicas > 1` at line 11 in HDFS), and (3) the instruction sequence of the common prefix. Because the instruction sequence in the common prefix provides a finer grained trace compared to the failure log, Pensieve's analysis will be more efficient because it benefits from more detailed runtime information. In the HDFS example, we will be able to find that the condition `nReplicas > 1` can be satisfied if we increase the replication factor to be greater than 1 in `createFile()` of the unit test.

## 5 Implementation of Kairux

We implemented Kairux with 3,473 lines of Java code. We use the Chord [34] static analysis framework to perform static and dynamic slicing. We use the JVM Tool Interface (JVM TI) [11] to set breakpoints at bytecode locations, allowing us to record the instruction sequences as well as enforce different thread schedulings by ordering the breakpoints. For each dynamic object used in each instruction, Kairux also assigns a unique tag using JVM TI. This allows Kairux to differentiate different runtime instances of the same source code object. The program controlling JVM TI is written in 2,961 lines of C++ code. We also wrote Python programs to parallelize the execution of unit tests. For HDFS, the system whose unit tests take the longest to run, we were able to reduce the time to run all unit tests from over 6 hours, when running sequentially on a regular file system, to less than 10 minutes, when running in parallel on tmpfs.

## 6 Experimental Evaluation

We answer the following questions in our experimental evaluation: (1) How effective is Kairux in locating the root cause of real-world failures? (2) Can Kairux locate the root cause of failures that are difficult to debug? (3) Why does Kairux fail to locate the root causes in some cases?

We evaluated Kairux on 10 real-world failures from HBase, HDFS, and ZooKeeper. One of them is the HDFS failure shown in Figure 2; we used it as a "training set" to guide the design of Kairux. The 9 remaining failures were randomly sampled from the benchmark used in Pensieve [56] and were not used in designing Kairux. We reproduced each failure using a series of commands packaged in a unit test. 7 out of these 10 bugs involve multiple nodes, and their unit test framework simulates a real environment by using threads and processes to simulate nodes.

### 6.1 Overall Result

Table 4 shows how Kairux performed on the 10 real-world failure cases. Overall, Kairux can successfully locate the root cause in 70% cases. Note that determining the root cause of each failure is fundamentally subjective. One idea we had for objectively evaluating Kairux was to check whether the developers' bug fix was indeed applied to the inflection point. Specifically, if we run the system after the bug fix with the same input and thread scheduling as in the failure execution $\sigma_f$ to obtain the non-failure execution $\sigma_c$, then we can compare where $\sigma_f$ and $\sigma_c$ diverge and check whether this divergence point is the same as the inflection point. This approach may be problematic in practice, however, because the divergence point may not be the actual root cause. For example, in the motivating HDFS example, at first, the developers avoided fixing the bug using proper synchronization (to prevent the data race) because it would incur undue overhead. Instead, they fixed the bug through a hack that allows the data race to occur but undoes its effect later. Only two years later did they change the fix to avoid the data race by using a copy of the block's original size during replication.

Hence, we used our best effort to objectively determine the root cause of each failure and then examined whether it is located at the inflection point identified by Kairux. In 5 of the 7 failures where Kairux was successful, the inflection point

| | Failure | Description | Success? | Static slice | $s_f$ | $\overline{s_f}$ | prefix | # of tests |
|---|---|---|---|---|---|---|---|---|
| HBase | 3403 | FSCK detects an error after region split fails | Yes | 32,647 | 99,287 | 657 | 231 | 2 |
| | 3627 | Region server crashes during region open operation | Yes | 15,483 | 15,691 | 151 | 150 | 1 |
| | 4078 | A column family is lost due to an HDFS error | No | - | - | - | - | - |
| HDFS | 1540 | Temporary namenode outage brings down all DNs | Yes | 45,560 | 3,955 | 86 | 85 | 1 |
| | 4205 | FSCK fails after creating a symbolic link | Yes | 63,672 | 14,726 | 352 | 336 | 2 |
| | 4558 | Load balancer fails to start | No | - | - | - | - | - |
| | 10453 | Fails to replicate block due to a race condition | Yes | 13,470 | 85,644 | 529 | 94 | 2 |
| Z.K. | 1434 | Checking the status of a non-existent znode fails | No | - | - | - | - | - |
| | 1851 | Client gets disconnected sending a create request | Yes | 15,134 | 1,016,143 | 131 | 42 | 1 |
| | 1900 | Trying to truncate a deleted log file fails | Yes | 13,221 | 14,828 | 257 | 221 | 1 |
| Averages | | | 70% | 26,894 | 178,610 | 309 | 165 | 1.43 |

**Table 4.** Kairux's result on real-world failures. The "Success" column shows whether Kairux can successfully locate the root cause. The next four columns show the number of instructions in the static slice, $s_f$, $\overline{s_f}$, and the common prefix in $\overline{s_f}$ for each case where Kairux is successful. The "# of tests" column shows the number of unit tests that is needed to construct $\sigma_p$.

identified by Kairux is indeed the same as the divergence point; i.e., the fix was applied to the inflection point identified by Kairux. However, in the other 2 cases, the inflection point returned by Kairux locates the root cause, but the developers chose to fix the bug differently, as in the initial fix of the HDFS example.

Many of the failures in Table 4 are indeed complex. On average, there are 26,894 instructions in the static slice of a failure. The developers may need to examine this many instructions to debug the failure if they try to infer its path by analyzing the program code [45]. Similarly, $s_f$ has an average of 178,610 instructions. Our dynamic slicing can significantly reduce the number of instructions from $s_f$. This is because we only keep one iteration of the map loop. For these server applications, a vast majority of the loops are map loops as each iteration processes independent data (e.g., a block or a request). Even then, there are still an average of 309 instructions in $\overline{s_f}$. Yet in 7 of the cases, Kairux can pinpoint the single instruction that is the root cause.

Many of the failures have a long propagation, as indicated by the large difference between the number of instructions in $\overline{s_f}$ and the common prefix. This difference is the length of the fault propagation path from the root cause to the symptom. Furthermore, the propagation may span multiple threads in $\overline{s_f}$: for 3 failures, Kairux needed multiple unit tests to construct $\sigma_p$.

There are 3 cases where Kairux failed to locate the root cause. For ZooKeeper-1434, Kairux failed because of the lack of unit tests. The failure occurred in a command line utility (`zkCli`) that crashed when a user tried to `stat` a non-existent znode. Although the failure is caused by a bug in the ZooKeeper server, there is no unit test for `zkCli`, so Kairux does not have a unit test to start its analysis. HDFS-4558 is similar. In the last case, HBase-4078, the symptom is outside of the HBase system (error state is on disk), so we cannot run Kairux with a symptom as starting point.

Kairux was able to finish the analysis in less than 32 minutes for each case. Obtaining the trace with JVM TI breakpoints takes about 3 minutes. Static analysis to obtain the static slice takes less than 1 minute. The rest is the time taken to run the inflection point detection algorithm.

**Discussions.** We observe two reasons for Kairux's high accuracy on these real-world failures. First, the target systems are well tested. This is shown in Table 2. For example, HDFS' tests achieve over 90% statement coverage. This means it is likely that there is a unit test that exercises the same path as the failure execution and only deviates at the root cause.

Second, the server systems we evaluated often do not have complex computation loops that contain complex loop-carried dependencies. This is evidenced by the relatively short length of Kairux's dynamic slice. Loops with a complex loop-carried dependency could be difficult for Kairux to handle because its comparison would need to follow the precise number of iterations. Unless there is a unit test execution that deviates from the failure execution exactly at the root cause location, it would likely be difficult for Kairux to successfully modify the execution of such loops.

The unit test we used to reproduce each failure only includes operations that are causally related to the failure; there was no noise from operations unrelated to the failure. However, these causally-unrelated executions will not be considered by Kairux since it operates on the dynamic slice. Therefore, they will not affect the efficacy of Kairux.

To validate this, we reproduced the HDFS failure in § 2 using a realistic workload. We performed the reproduction on a cluster of 10 datanodes and introduced noise by running a client that generated a random write workload. We also intentionally introduced additional failure-related operations to modify the block that triggered the data race three times before the failure occurred. The result validates our analysis above. The failure-unrelated operations (i.e., the random writes on other blocks) do not affect Kairux's analysis as they are pruned by dynamic slicing. The three additional write operations to the raced block were included in the dynamic

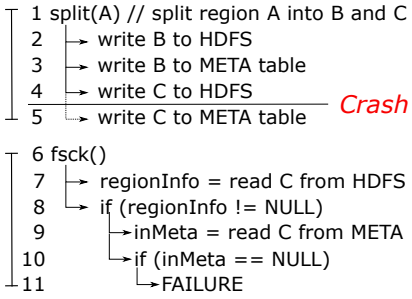


**Figure 7.** The HBase-3403 failure.

slice, increasing the length of $\overline{s_f}$ from 529 as reported in Table 4 to 786. Nonetheless, Kairux is still able to locate the root cause accurately.

### 6.2 Case Study

We discuss another interesting failure example: HBase-3403. This failure represents a class of failures that are caused by a "missing event," i.e., an event that should have taken place but did not, in contrast to failures caused by the occurrence of an anomalous event. Such failures are particularly difficult to debug because one cannot simply find the bug by searching through the failure's execution path. For the same reason, missing events are difficult to detect using existing fault localization techniques.

Figure 7 shows this failure. It involves two threads: `split` and `fsck`. First, `split()` starts to split a region A into B and C. It does so in four steps: write B to HDFS, write B's information to a META table, and then the same for C. However, the system crashes right after C is written to HDFS. When restarted, HBase performs `fsck()`. Because it finds C in HDFS (step 7 in Figure 7), it checks if C's information is in the META table. However, because the system crashed before C's information was written to META, `fsck()` cannot find it and therefore throws an exception at step 11. The failure is because step 5 did *not* occur.

Kairux infers that the inflection point is located after step 4. By inspecting $\sigma_p$ constructed by Kairux which contains step 5, a developer can clearly see that the root cause is that step 5 should have occurred. Kairux starts its analysis of the `fsck` thread, and by comparing with a unit test that performs `fsck` successfully, it finds the deviation point in the `fsck` thread to be at step 11. Kairux then follows the data-flow on `regionInfo` into the `split()` thread, and starts to compare with another unit test that splits a region. Initially Kairux will conclude that there is no deviation point when comparing $\overline{s_f}|$split with $s_u|$split, because step 5 is *not* in $\overline{s_f}|$split. However, Kairux will further attempt to patch the dynamic slice with the missing instructions. By performing dynamic slicing on $s_u$ from step 10, Kairux is able to locate step 5 in $\overline{s_u}$, and in another round of comparison it determines the inflection point is after step 4. Not only did Kairux accurately identify the root cause, it also provides the non-failure execution $\sigma_p$, which clearly tells the user the cause is because step 5 on C is missing. In fact, the HDFS example in Figure 2 is also an example of a missing event. Kairux infers that the deviation point in the `ReplicationMonitor` thread is the instruction after line 22, and when compared with $\sigma_p$, a user would understand that line 24 should have occurred but did not.

## 7 Related Work

**Statistical fault localization:** A large portion of fault localization research uses statistical debugging [7, 16, 24, 25, 27, 28, 33, 48]. For example, Liblit *et al.* [27] record branch conditions, function return values, and whether two variables are equal, and consider each predicate's statistical correlation with the failure runs. Failure sketching [25] uses Intel-PT and hardware watchpoints to record control and data-flow. It uses a statistical method to rank the recorded predicates and presents the result to a user to decide whether the root cause was identified. Others use statistical approaches to learn the program invariant [8, 16, 38]; i.e., the predicates that should hold in a non-failure execution. A violation of an invariant in the failure execution is likely the root cause. Compared to statistical debugging, Kairux is analytical as it is based on analyzing a program's control- and data-flow. As a result, for both the failure and non-failure executions, Kairux is able to produce highly simplified execution paths that contain the key control- and data-flow, as well as how they are matched, as a reference for developers to better understand the root cause. On the other hand, statistical debugging does not rely on the unit test suite and does not require the failure to be reproduced. Therefore, Kairux and statistical debugging are complementary to each other.

**Delta debugging** isolates failure-inducing input [55]. A user provides a failed unit test and a successful test. Delta debugging then systematically replaces input values of a failed test with the corresponding values in the successful test to isolate the smallest set of input values that cause the failure. A variant of delta debugging [53] uses a similar search algorithm to isolate the failure-inducing *code changes* in a program update that caused a failure. Cleve and Zeller further extended delta debugging [10, 54] to isolate failure-inducing program memory states. They compare the memory states at each point of the failure and non-failure executions, and modify the program states in the non-failure execution to derive the smallest set of memory states that, when modified, turns the non-failure execution into a failure one.

While Kairux uses a similar approach to modify execution states, the difference is substantial. First, delta debugging requires the user to provide a unit test that is similar to the failure, otherwise the result will not be meaningful, whereas Kairux automatically searches for it. In addition, failures on complex distributed systems require combining multiple unit tests. Third, Cleve and Zeller do not check whether the modified execution can indeed be reproduced by a series

of commands. Finally, Cleve and Zeller return a chain of multiple memory states as possible root causes, whereas Kairux only returns one inflection point.

**Other approaches that compare execution traces.** Triage [43] is able to record and replay executions using lightweight checkpointing. During replay, it builds a basic block vector that counts the number of appearances of each basic block for each failure and non-failure execution. Triage then finds the non-failure execution that has the most similar vector when compared against the vector of the failed execution. It returns the edit distance between two to help identify the root cause.

Given a set of non-failure executions, Guo *et al.* [19] find the one most similar to a failure execution. They report all the points where the two executions deviate as possible root causes. In contrast, Kairux is able to infer a single root cause. In addition, Kairux's goal is not to find the most similar test execution, but instead to construct one. The search for the similar test execution is also different. Whenever there is a deviation, Kairux checks if the non-failure execution can be modified to take the same path as the failure execution, while Guo *et al.* [19] merely check whether there is a program point in the future where the two executions merge back.

X-ray [4] is a tool that diagnoses performance bugs. It compares a pair of slow and normal execution paths by extracting the longest common subsequence (LCS) from basic block traces. The basic blocks that are not in the LCS are considered diverging basic blocks. X-ray then attributes a performance cost to each diverging basic block and ranks them based on this performance cost. It is difficult to apply X-ray to correctness bugs because X-ray prunes failure-unrelated basic blocks by ranking their performance costs, which is unavailable in correctness bugs, whereas Kairux does so by using dynamic slicing. Similarly, it is also difficult to apply Kairux on performance bugs.

**Pensieve.** Pensieve [56] can construct a unit test that can be used to reproduce the production failure from the log. Pensieve and Kairux solve different problems, and they use fundamentally different techniques. Pensieve uses *static program analysis* to analyze the system's byte code to reconstruct the *failure* path leading to the printing of the logs, whereas Kairux reconstructs the *non-failure* path $\sigma_p$ using *dynamic analysis* on the execution traces. Kairux uses unit tests whereas Pensieve does not.

## 8 Concluding Remarks

This paper proposes Kairux, a tool that adopts a new approach to root cause localization. It is based on the Inflection Point Hypothesis, which states that the root cause is located at the first instruction where the failure execution deviates from the non-failure execution that has the longest instruction sequence prefix in common with that of the failure execution. This transforms root cause analysis into a principled search to identify the non-failure execution with the longest common prefix. By evaluating Kairux on real-world failures from distributed systems, we show that it can successfully pinpoint the root cause in 70% of the cases.

## Acknowledgements

We greatly appreciate the insightful feedback from our shepherd Baris Kasikci and the anonymous reviewers. We thank David Lion, Xiang (Jenny) Ren, and Adrian Chiu for useful discussions and support. The discussion of failures that are caused by a "missing event" in § 6.2 was inspired by a conversation with Jason Flinn during SOSP 2017. This research is supported by an NSERC Discovery grant, a NetApp Faculty Fellowship, a VMware gift, a Connaught Innovation Award, and a Huawei grant. Yongle Zhang, Kirk Rodrigues, and Yu Luo are supported by an SOSP 2019 student scholarship from the ACM Special Interest Group in Operating Systems to attend the SOSP'19 conference.

## References

[1] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. In *Proceedings of the 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, 246–256.

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

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

[4] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th Conference on Operating Systems Design and Implementation (OSDI '12)*. USENIX, 307–320.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Conference on Operating Systems Design and Implementation (OSDI '04)*. USENIX, 259–272.

[6] Benjamin H. Sigelman and Luiz André Barroso and Mike Burrows and Pat Stephenson and Manoj Plakal and Donald Beaver and Saul Jaspan and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google Incorporated. https://research.google.com/archive/papers/dapper-2010-1.pdf

[7] Lionel C. Briand, Yvan Labiche, and Xuetao Liu. 2007. Using Machine Learning to Support Debugging with Tarantula. In *Proceedings of the 18th IEEE International Symposium on Software Reliability (ISSRE '07)*. IEEE Computer Society, 137–146.

[8] Yuriy Brun and Michael D. Ernst. 2004. Finding Latent Code Errors via Machine Learning over Program Executions. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, 480–490.

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

[10] Holger Cleve and Andreas Zeller. 2005. Locating Causes of Program Failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. ACM, 342–351.

[11] Oracle Corporation. 2018. Java™ Virtual Machine Tool Interface (JVM TI). http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/.

[12] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. 2013. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP '13)*. ACM, 388–405.

[13] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 12th Conference on Operating Systems Design and Implementation (OSDI '18)*. USENIX, 17–32.

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

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

[16] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. ACM, 213–224.

[17] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-Trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th Conference on Networked Systems Design and Implementation (NSDI '07)*. USENIX, 271–284.

[18] Apache Software Foundation. 2017. *Apache Thrift*. https://thrift.apache.org/lib/

[19] Liang Guo, Abhik Roychoudhury, and Tao Wang. 2006. Accurately Choosing Execution Runs for Software Fault Localization. In *Proceedings of the 15th International Conference on Compiler Construction (CC '06)*. Springer-Verlag, 80–95. https://doi.org/10.1007/11688839_7

[20] Xiaoqiao He. 2019. *NameNode RepicationMonitor Failure Diagnosis (Chinese)*. https://hexiaoqiao.github.io/blog/2016/09/13/namenode-repicationmonitor-exception-trace/

[21] Xiaoqiao He. 2019. *NameNode RepicationMonitor Failure Diagnosis (English Translation)*. https://bit.ly/2UA9q5v

[22] Xiaoqiao He. 2019. *ReplicationMonitor Thread Could Stuck for Long Time Due to the Race Between Replication and Delete of Same File in a Large Cluster*. https://issues.apache.org/jira/browse/HDFS-10453

[23] Google Incorporated. 2019. *Protocol Buffers*. https://developers.google.com/protocol-buffers/

[24] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proceedings of the 20th Conference on Automated Software Engineering (ASE '05)*. ACM, 273–282.

[25] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, 344–360.

[26] Jean-Claude Laprie. 1995. Dependable Computing: Concepts, Limits, Challenges. In *Proceedings of the 25th International Conference on Fault-tolerant Computing (FTCS '95)*. IEEE Computer Society, 42–54.

[27] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, 15–26.

[28] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P. Midkiff. 2006. Statistical Debugging: A Hypothesis Testing-based Approach. *IEEE Transactions on Software Engineering* 32, 10 (Oct. 2006), 831–848.

[29] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. DTHREADS: Efficient Deterministic Multithreading. In *Proceedings of the 23rd Symposium on Operating Systems Principles (SOSP '11)*. ACM, 327–336.

[30] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, 378–393.

[31] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. PSE: Explaining Program Failures via Postmortem Static Analysis. In *Proceedings of the 12th Symposium on Foundations of Software Engineering (SIGSOFT '04/FSE-12)*. ACM, 63–72.

[32] Steve McConnell. 2004. *Code Complete*. Pearson Education.

[33] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th Conference on Networked Systems Design and Implementation (NSDI '12)*. USENIX, 353–366.

[34] Mayur Naik. 2015. Chord: Java Bytecode Analysis. https://bitbucket.org/psl-lab/jchord/.

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

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

[37] Google Cloud Platform. 2016. *Google App Engine Incident 16008*. https://status.cloud.google.com/incident/appengine/16008

[38] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. 2013. Using Likely Invariants for Automated Software Fault Localization. In *Proceedings of the 18th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, 139–152.

[39] Amazon Web Services. 2011. *Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region*. https://aws.amazon.com/message/65648/

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

[41] Gregory Tassey. 2002. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Technical Report 7007.011. National Institute of Standards and Technology.

[42] H. Thane and H. Hansson. 2000. Using Deterministic Replay for Debugging of Distributed Real-time Systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS '00)*. IEEE Computer Society, 265–272.

[43] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. 2007. Triage: Diagnosing Production Run Failures at the User's Site. In *Proceedings of 21st Symposium on Operating Systems Principles (SOSP '07)*. ACM, 131–144.

[44] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proceedings of the 16th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, 15–26.

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

[46] Bill Willson. 2014. *Root Cause*. http://www.bill-wilson.net/root-cause-analysis/rca-wiki/root-cause

[47] Paul F. Wilson, Larry D. Dell, and Gaylord F. Anderson. 1993. *Root Cause Analysis: A Tool for Total Quality Management.* ASQ Quality Press.

[48] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting Large-scale System Problems by Mining Console Logs. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, 117–132.

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

[50] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In *Proceedings of the 15th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*. ACM, 143–154.

[51] Cristian Zamfir, Gautam Altekar, George Candea, and Ion Stoica. 2011. Debug Determinism: The Sweet Spot for Replay-based Debugging. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*. USENIX, Article 18, 5 pages.

[52] Cristian Zamfir and George Candea. 2010. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, 321–334.

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

[54] Andreas Zeller. 2002. Isolating Cause-effect Chains from Computer Programs. In *Proceedings of the 10th Symposium on Foundations of Software Engineering (SIGSOFT '02/FSE-10)*. ACM, 1–10.

[55] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (Feb. 2002), 183–200.

[56] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, 19–33.

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

[58] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. Lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *Proceedings of the 11th Conference on Operating Systems Design and Implementation (OSDI '14)*. USENIX, 629–644.