Kairux: Distributed System Fault Localization based on The Inflection Point Hypothesis

January 1, 2023

ARTICLE: RESEARCH

Authors: Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan

Article shepherded by: Rik Farrow

We have built a tool that performs fault localization for Java-based distributed systems. Using the Inflection Point Hypotheses, that is, finding the last point in the failure execution where the failure can still be avoided, our tool can uncover fault locations reliably for distributed system failures and provide a better explanation compared to current approaches. In this article, we explain how this is done.

Introduction

Fault localization (i.e., identifying the root cause of a failure) in distributed systems is a daunting task, as the failure execution trace on real-world distributed systems could contain millions to billions of instructions, and locating the instruction that is the root cause is like finding a needle in a haystack. Most existing approaches for fault localization use a probabilistic approach [4]. They infer the 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 falls short of providing an effective explanation – the execution context leading to the root cause and the propagation from the root cause to the failure, which is typically needed by developers to understand the root cause and develop a fix. In our paper, we propose the Inflection Point Hypothesis – a principle that captures both a potential root cause of a failure and its explanation, and we introduce Kairux, a tool that automatically pinpoints the root cause of a failure in a distributed system based on the Inflection Point Hypothesis. We show Kairux's effectiveness on failures from widely-deployed, real-world distributed

systems, namely, HBase, HDFS, and ZooKeeper.

The Inflection Point Hypothesis

Inflection Point Hypothesis. *If we model an execution as a totally ordered sequence of instructions, then the root cause can be identified by the first instruction (**inflection point**) 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.*

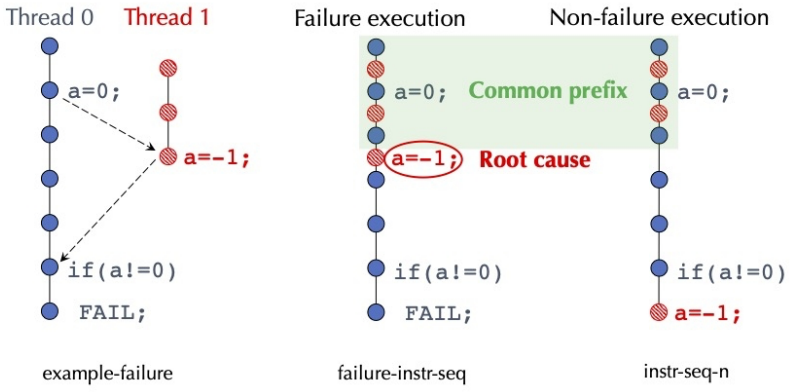


Figure 1: An abstract failure example simplified from a real-world data race on HDFS.

As an abstract example, consider a failure caused by a read-after-write data race. Figure 1 depicts an execution example-failure, 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, failure-instr-seq, obtained from example-failure, 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 failure-instr-seq, then we should find one, instr-seq-n (shown on the right of the figure), that has the longest common prefix with failure-instr-seq. According to the Inflection Point Hypothesis, the inflection point of the failure is located at the first instruction in failure-instr-seq that differs from instr-seq-n, namely a=-1 in thread 1. Intuitively, it is clear there can be no valid, failure-free instruction sequence that has a longer common prefix with failure-instr-seq: 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 failure-instr-seq.

Intuitively, the inflection point is the last point in the failure execution where the failure can still be avoided. It is effective to capture the root cause of a failure in distributed systems because: (1) Our study [5] reveals that most (77%) distributed system failures happen due to interaction of executions triggered by multiple input

events. (2) As shown by the example, the Inflection Point Hypothesis naturally captures the last interaction point that made the failure inevitable. The Inflection Point Hypothesis transforms root cause analysis into a principled search problem to identify the valid non-failure execution that has the longest common prefix with the failure execution.

Design of Kairux

Based on the Inflection Point Hypothesis, we have designed and implemented a tool, Kairux, capable of locating the root causes of most distributed system failures and providing an explanation automatically. Kairux takes three inputs: (1) the steps to reproduce the failure, typically packaged in a unit test; (2) the failure symptom; (3) source code; and (4) all the code's unit tests. Kairux outputs: (1) the inflection point; (2) the non-failure instruction sequence instr-seq-n having the longest common prefix with failure-instr-seq; and (3) the steps needed to reproduce instr-seq-n in the form of a unit test. The comparison between instr-seq-n and failure-instr-seq provides an explanation for the root cause (inflection point).

The search for the inflection point, as described above, is 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. Kairux utilizes the following key ideas:

- **Adaptive Dynamic Slicing**. Kairux 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. It initially processes the thread containing the failure symptom and adaptively extends its analysis to other threads.

- **Utilizing Unit Tests**. Instead of trying to enumerate all possible failure-free sequences, Kairux only considers failure-free sequences obtained from the system's existing unit tests, as we observed real-world distributed systems' unit tests achieve high coverage (86%). Kairux uses a heuristic to prioritize the unit tests most likely to have instruction sequences in common with the failure execution. Kairux also stitches multiple tests together when needed.

- **Valid Execution Modification**. 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. The modifications include modifying parameters and scheduling. It ensures that the modifications will always result in a valid execution that can be reproduced by a unit test. The modification attempt stops when any longer common prefix would end up in failure.

Descriptions of Kairux's algorithms performing adaptive dynamic slicing, comparison between failure execution trace and unit test traces, execution modification, and more details of our design can be found in our paper [6].

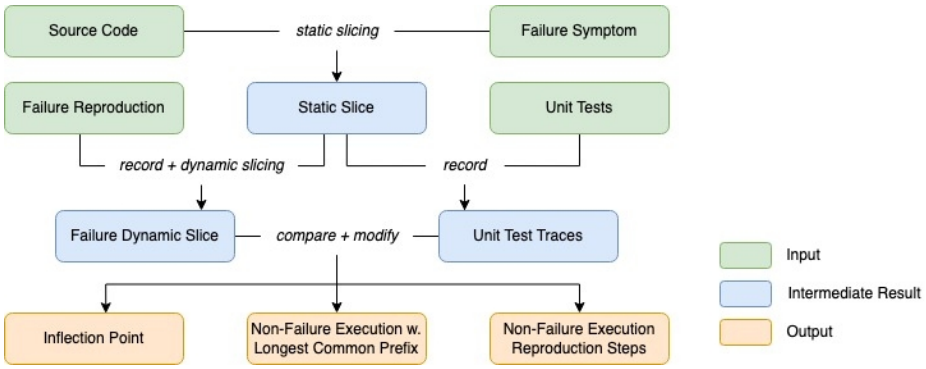


Figure 2: Architecture of Kairux.

Architecture and Implementation of Kairux

In theory, we can obtain dynamic program slices by recording a trace of every instruction that is executed and then inferring the slice from this trace. In practice, however, doing so has a high overhead that can be prohibitive. Therefore, as shown in Figure 2, 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 [3]. The static slice will be a super-set of the instructions that belong in the dynamic slice of any failure execution. Kairux uses the Chord [2] static analysis framework to perform static slicing.

Kairux then uses the JVM Tool Interface (JVM TI) [1] to set 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 and then performs a similar dependency analysis on it to obtain the dynamic slice. Kairux acquires the dynamic slice across the network by annotating network communication libraries. During execution modification, Kairux also uses JVM TI to enforce different thread schedulings by ordering the breakpoints. For each dynamic object used in each instruction, Kairux assigns a unique tag using JVM TI to differentiate different runtime instances of the same source code object and track data flow.

Kairux also includes Python programs to parallelize and accelerate 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.

Evaluating Kairux

We evaluated Kairux on randomly sampled, real-world failures from HBase, HDFS, and ZooKeeper. We reproduced each failure using a series of commands packaged in a unit test. These systems' unit test framework simulates a real environment by using threads and processes to simulate nodes. Overall, Kairux can successfully locate the root cause in 70% of cases. Kairux effectively reduces the number of instructions to be examined to understand the root cause: 0.2% of the instructions in the failure execution. In addition, Kairux is effective in identifying and explaining

failures caused by “missing events”, i.e., events that should have taken place but did not, in contrast to failures caused by the occurrence of an anomalous event. Kairux is able to identify such missing events by comparing the failure execution against unit test executions. A full set of statistics and a case study can be found in the paper [6].

Caveats

For many failures, there are multiple root cause candidates. Picking the one root cause can be fundamentally subjective. Our definition of inflection point picks the one that comes last as the root cause. Our evaluation shows its effectiveness for fault localization in distributed systems. However, cases can be made that the other causes are better choices. In addition, 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. In these cases, our hypothesis will only identify the last bug as the root cause, i.e., the bug in the error handling logic. Removing root causes from the failure execution and applying Kairux repeatedly could identify multiple failure-inducing causes.

References:

[1] Oracle Corporation. Java™ virtual machine tool interface (JVM TI). http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/, 2018.

[2] Mayur Naik. Chord: Java bytecode analysis. https://bitbucket.org/psl-lab/jchord/, 2015.

[3] Mark Weiser. Program slicing. In Proceedings of the 5th International Conference on Software Engineering (ICSE), 1981.

[4] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. IEEE Transactions on Software Engineering, 2016.

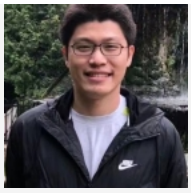
[5] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In Proceedings of the 11th Conference on Operating Systems Design and Implementation (OSDI), 2014.

[6] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP), 2019.

Tags: distributed systems, failure diagnosis, debugging

Last updated January 25, 2023

AUTHORS:



Yongle Zhang is an assistant professor in the Computer Science Department at Purdue University. His research interests are in systems software with a focus on improving the reliability and availability of complex, real-world systems. Some of his recent research includes designing tools that help developers with failure detection and diagnosis in production cloud systems, as well as design and implementation of diagnosable software systems.

✉ yonglezh@purdue.edu

Kirk Rodrigues is a co-founder of YScope and a 5th-year PhD candidate under the supervision of Ding Yuan at the University of Toronto. Our Distributed Systems Research Group focuses on creating ways to efficiently diagnose performance and reliability issues in large-scale distributed systems.

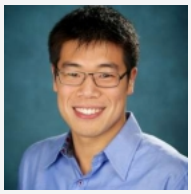
✉ kirk.rodrigues@yscope.com

Yu (Jack) Luo is a co-founder of YScope and a 5th-year PhD candidate under the supervision of Ding Yuan at the University of Toronto. His research interest is in system software, with a focus on developing practical solutions to improve the availability and performance of large software systems. Many of our research group's pioneer work is in the field of failure diagnosis in large distributed systems via log analysis. Our ultimate goal is to build effective and practical tools which can triage and help users perform post-mortem failure analysis as well as providing non-intrusive monitoring on full-stack distributed systems.

✉ jack.luo@mail.utoronto.ca

Michael Stumm is a Professor in the Computer Engineering Department at the University of Toronto. Dr. Stumm's research interests are in the general area of computer systems software with an emphasis on operating systems for distributed systems and multiprocessors. While professor, Stumm co-founded two companies, SOMA Networks, and OANDA, a currency trading company. He ran OANDA from 2001 until 2012.

✉ michael@stumm.ca



Ding Yuan is an associate professor in the Electrical and Computer Engineering Department and (by courtesy) Department of Computer Science, University of Toronto, as well as a co-founder of YScope. He is a Canada Research Chair in Systems Software and a recipient of McCharles Prize for Early Career Research Distinction. His research interest is systems software, with a focus on developing practical solutions to improve the availability and performance of large software systems.

✉ yuan@ece.utoronto.ca

Sign up for Our Newsletter:

First Name | Last Name | Email

I'm not a robot
reCAPTCHA
Privacy - Terms

Submit