Causality-Guided Adaptive Interventional Debugging

Anna Fariha
University of Massachusetts Amherst
afariha@cs.umass.edu

Suman Nath
Microsoft Research
Suman.Nath@microsoft.com

Alexandra Meliou
University of Massachusetts Amherst
ameli@cs.umass.edu

ABSTRACT

Runtime nondeterminism is a fact of life in modern database applications. Previous research has shown that nondeterminism can cause applications to *intermittently* crash, become unresponsive, or experience data corruption. We propose Adaptive Interventional Debugging (AID) for debugging such intermittent failures.

AID combines existing statistical debugging, causal analysis, fault injection, and group testing techniques in a novel way to (1) pinpoint the root cause of an application's intermittent failure and (2) generate an explanation of how the root cause triggers the failure. AID works by first identifying a set of runtime behaviors (called predicates) that are strongly correlated to the failure. It then utilizes temporal properties of the predicates to (over)-approximate their causal relationships. Finally, it uses fault injection to execute a sequence of interventions on the predicates and discover their true causal relationships. This enables AID to identify the true root cause and its causal relationship to the failure. We theoretically analyze how fast AID can converge to the identification. We evaluate AID with six real-world applications that intermittently fail under specific inputs. In each case, AID was able to identify the root cause and explain how the root cause triggered the failure, much faster than group testing and more precisely than statistical debugging. We also evaluate AID with many synthetically generated applications with known root causes and confirm that the benefits also hold for them.

KEYWORDS

root-causing; trace analysis; group testing; concurrency bug

ACM Reference Format:
Anna Fariha, Suman Nath, and Alexandra Meliou. 2020. Causality-Guided Adaptive Interventional Debugging . In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389694

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'20, June 14–19, 2020, Portland, OR, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389694

1 INTRODUCTION

Modern data management systems and database-backed applications run on commodity hardware and heavily rely on asynchronous and concurrent processing [16, 25, 52, 61]. As a result, they commonly experience runtime nondeterminism such as transient faults and variability in timing and thread scheduling. Unfortunately, software bugs related to handling nondeterminism are also common to these systems. Previous studies reported such bugs in MySQL [11, 46], PostgreSQL [45], NoSQL systems [40, 71], and database-backed applications [8], and showed that they can cause intermittent crashes, unresponsiveness, and data corruptions. It is, therefore, crucial to identify and fix these bugs as early as possible.

Unfortunately, localizing root causes of intermittent failures is extremely challenging [43, 47, 72]. For example, concurrency bugs such as deadlocks, order and atomicity violation, race conditions, etc. may appear only under very specific thread interleavings. Even when an application executes with the same input in the same environment, these bugs may appear only rarely (e.g., in *flaky* unit tests [47]). When a concurrency bug is confirmed to exist, the debugging process is further complicated by the fact that the bug cannot be consistently reproduced. Heavy-weight techniques based on record-replay [3] and fine-grained tracing with lineage [2, 55] can provide insights on root causes after a bug manifests; but their runtime overheads often interfere with thread timing and scheduling, making it even harder for the intermittent bugs to manifest in the first place [38].

Statistical Debugging (SD) [30, 32, 42, 44] is a data-driven technique that partly addresses the above challenge. SD uses lightweight logging to capture an application's runtime (mis)-behaviors, called *predicates.* An example predicate indicates whether a method returns null in a particular execution or not. Given an application that intermittently fails, SD logs predicates from many successful and failed executions. SD then uses statistical analyses of the logs to identify *discriminative predicates* that are highly correlated with the failure.

SD has two key limitations. First, SD can produce many discriminative predicates that are correlated to, but not a true cause of, a failure. Second, SD does not provide enough insights that can explain how a predicate may eventually lead to the failure. Lack of such insights and the presence of many non-causal predicates make it hard for a developer to identify the true root cause of a failure. SD expects that a developer has sufficient domain knowledge about if/how a predicate

can eventually cause a failure, even when the predicate is examined in isolation without additional context. This is often hard in practice, as is reported by real-world surveys [56].

Example 1.1. *To motivate our work, we consider a recently reported issue in Npgsql [53], an open-source ADO.NET data provider for PostgreSQL. On its GitHub repository, a user reported that a database application intermittently crashes when it tries to create a new PostgreSQL connection (GitHub issue #2485 [54]). The underlying root cause is a data race on an array index variable. The data race, which happens only when racing threads interleave in a specific way, causes one of the threads to access beyond the size of the array. This causes an exception that crashes the application.*

We used SD to localize the root cause of this nondeterministic bug (more details are in Section 7). SD identified 14 predicates, only 3 of which were causally related to the error. Other predicates were just symptoms of the root cause or happened to co-occur with the root cause.

In Section 7, we describe five other case studies that show the same general problem: SD produces too many predicates, only a small subset of which are causally related to the failure. Thus, SD is not specific enough, and it leaves the developer with the task of identifying the root causes from a large number of candidates. This task is particularly challenging, since SD does not provide explanations of how a potential predicate can eventually lead to the failure.

In this paper, we address these limitations with a new data-driven technique called *Adaptive Interventional Debugging* (AID). Given predicate logs from successful and failed executions of an application, AID can pinpoint *why* the application failed, by identifying one (or a small number of) predicate that indicates the real root cause (instead of producing a large number of potentially unrelated predicates). Moreover, AID can explain *how* the root cause leads to the failure, by automatically generating a causal chain of predicates linking the root cause, subsequent effects, and the failure. By doing so, AID enables a developer to quickly localize (and fix) the bug, even without deep knowledge about the application.

AID achieves the above by combining SD with causal analysis [49–51], fault injection [2, 24, 35], and group testing [27] in a novel way. Like SD, it starts by identifying discriminative predicates from successful and failed executions. In addition, AID uses temporal properties of the predicates to build an *approximate causal DAG* (Directed Acyclic Graph), which contains a superset of all true causal relationships among predicates. AID then progressively refines the DAG. In each round of refinement, AID uses ideas from adaptive group testing to carefully select a subset of predicates. Then, AID re-executes the application during which it *intervenes* (i.e., modifies application's behavior by e.g., injecting faults) to forcefully alter values of the selected predicates. Depending on whether the intervention still causes the application to fail or not, AID confirms or discards causal relationships in the approximate causal DAG, assuming counterfactual causality (C is a counterfactual cause of F iff F would not occur unless C occurs) and a single root cause. A sequence of interventions enables AID to identify the root cause and generate a *causal explanation path*, a sequence of causally-related predicates that connect the root cause to the failure.

A key benefit of AID is its efficiency—it can identify root-cause and explanation predicates with significantly fewer rounds of interventions than adaptive group testing. In group testing, predicates are considered independent and hence each round selects *a random subset* of predicates to intervene on and makes causality decisions about only those intervened predicates. In contrast, AID uses potential causality among predicates (in the approximate causal DAG). This enables AID to (1) make decisions not only about the intervened predicates, but also about other predicates; and (2) carefully select predicates whose intervention would maximize the effect of (1). Through theoretical and empirical analyses we show that this can significantly reduce the number of required interventions. This is an important benefit in practice since each round of intervention involves executing the application with fault injection and hence is time-consuming.

We evaluated AID on 3 open-source applications: Npgsql, Apache Kafka, Microsoft Azure Cosmos DB, and on 3 proprietary applications in Microsoft. We used known issues that cause these applications to intermittently fail even for same inputs. In each case, AID was able to identify the root cause of failure and generate an explanation that is consistent with the explanation provided by respective developers. Moreover, AID achieved this with significantly fewer interventions than traditional adaptive group testing. We also performed sensitivity analysis of AID with a set of synthetic workloads. The results show that AID requires fewer interventions than traditional adaptive group testing, and has significantly better worst-case performance than other variants.

Contributions. We make the following contributions:

- We propose Adaptive Interventional Debugging (AID), a data-driven technique that localizes the root cause of an intermittent failure through a novel combination of statistical debugging, causal analysis, fault injection, and group testing (Section 2). AID provides significant benefits over the state-of-the-art SD techniques by (1) pinpointing the root cause of an application's failure and (2) generating an explanation of how the root cause triggers the failure (Sections 3–5).
- We use information theoretic analysis to show that AID, by utilizing causal relationship among predicates, can converge to the true root cause and explanation significantly faster than traditional adaptive group testing (Section 6).

- We evaluate AID with six real-world applications that intermittently fail under specific inputs (Section 7). AID was able to identify the root causes and explain how the root causes triggered the failure, much faster than adaptive group testing and more precisely than SD. We also evaluate AID with many synthetically generated applications with known root causes and confirm that the benefits hold for them as well.

2 BACKGROUND AND PRELIMINARIES

AID combines several existing techniques in a novel way. We now briefly review the techniques.

Statistical Debugging. Statistical debugging (SD) aims to automatically pinpoint likely causes for an application's failure by statistically analyzing its execution logs from many successful and failed executions. It works by instrumenting an application to capture runtime *predicates* about the application's behavior. Examples of predicates include "the program takes the `false` branch at Line 31", "the method `foo()` returns `null`", etc. Executing the instrumented application generates a sequence of predicate values, which we refer to as *predicate logs*. Without loss of generality, we assume that all predicates are Boolean.

Intuitively, the true root cause of the failure will cause certain predicates to be true only in the failed logs (or, only in the successful logs). Given logs from many successful executions and many failed executions of an application, SD aims to identify those *discriminative* predicates. Discriminative predicates encode program behaviors of failed executions that deviate from the ideal behaviors of the successful executions. Without loss of generality, we assume that discriminative predicates are `true` during failed executions. The predicates can further be ranked based on their *precision* and *recall*, two well-known metrics that capture their discriminatory power.

$$precision(P) = \frac{\text{\#failed executions where } P \text{ is } \texttt{true}}{\text{\#executions where } P \text{ is } \texttt{true}}$$

$$recall(P) = \frac{\text{\#failed executions where } P \text{ is } \texttt{true}}{\text{\#failed executions}}$$

Causality. Informally, causality characterizes the relationship between an event and an outcome: the event is a cause if the outcome is a consequence of the event. There are several definitions of causality [23, 58]. In this work, we focus on *counterfactual* causes. According to *counterfactual causality*, C causes E iff E would not occur unless C occurs. Reasoning about causality frequently relies on a mechanism for *interventions* [26, 57, 62, 69], where one or more variables are forced to particular values, while the mechanisms controlling other variables remain unperturbed. Such interventions uncover counterfactual dependencies between variables.

Trivially, executing a program is a cause of its failure: if the program was not executed at the first place, the failure would not have occurred. However, our analysis targets *fully*-discriminative predicates (with 100% precision and recall), eliminating trivial predicates that are program invariants.

Fault Injection. In software testing, *fault injection* [2, 24, 35, 48] is a technique to force an application, by instrumenting it or by manipulating the runtime environment, to execute a different code path than usual. We use the technique to *intervene* on (i.e., repair) discriminative predicates. Consider a method `ExecQuery()` that returns a result object in all successful executions and `null` in all failed executions. Then, the predicate "`ExecQuery()` returns `null`" is discriminative. The predicate can be intervened by forcing `ExecQuery()` to return the correct result object. Similarly, the predicate "there is a data race on X" can be intervened by delaying one access to X or by putting a lock around the code segments that access X to avoid simultaneous accesses to X.

Group Testing. Given a set of discriminative predicates, a naïve approach to identify which predicates cause the failure is to intervene on one predicate at a time and observe if the intervention causes an execution to succeed. However, the number of required interventions is linear in number of predicates. *Group testing* reduces the number of interventions.

Group testing refers to the procedure that identifies certain items (e.g., defective) among a set of items while minimizing the number of *group tests* required. Formally, given a set $\mathcal{P}$ of N elements where D of them are defective, group testing performs k group tests, each on group $\mathcal{P}_i \subseteq \mathcal{P}$. Result of test on group $\mathcal{P}_i$ is positive if $\exists P \in \mathcal{P}_i$ s.t. P is defective, and negative otherwise. The objective is to minimize k, i.e., the number of group tests required. In our context, a group test is simultaneous intervention on a group of predicates, and the goal is to identify the predicates that cause the failure.

Two variations of group testing are studied in the literature: *adaptive* and *non-adaptive*. Our approach is based on adaptive group testing where the i-th group to test is decided *after* we observe the results of all $1 \leq j < i$ previous group tests. A trivial upper bound for adaptive group testing [27] is $O(D \log N)$. A simple binary search algorithm can find each of the D defective items in at most $\log N$ group tests and hence a total of $D \log N$ group tests are sufficient to identify all defective items. Note that if $D \geq \frac{N}{\log N}$, then a linear strategy is preferable over any group testing scheme. Hence, we assume that $D < \frac{N}{\log N}$.

3 AID OVERVIEW

Adaptive Interventional Debugging (AID) targets applications (e.g., *flaky* tests [47]) that, even with the same inputs, intermittently fail due to various runtime nondeterminism such as thread scheduling and timing. Given predicate logs of successful and failed executions of an application, the goals of AID are to (1) identify *what* predicate actually causes

the failure, and (2) generate an explanation of *how* the root cause leads to the failure (via a sequence of intermediate predicates). This is in contrast with traditional statistical debugging, which generates a set of potential root-cause predicates (often a large number), without any explanation of how each potential root cause may lead to the failure.

Figure 1 shows an overview of AID. First, the framework employs standard SD techniques on predicate logs to identify a set of *fully*-discriminative predicates, i.e., predicates that *always* appear in the failed executions and *never* appear in the successful executions. Then, AID uses the temporal relationships of predicates to infer *approximate causality*: if P_1 temporally precedes P_2 in *all* logs where they both appear, then P_1 *may* cause P_2. AID represents this approximate causality in a DAG called *Approximate Causal DAG* (AC-DAG), where predicates are nodes and edges indicate these possible causal relationships. We describe the AC-DAG in Section 4.

Based on its construction, the AC-DAG is guaranteed to contain all the true root-cause predicates and causal relationships among predicates. However, it may also contain additional predicates and edges that are not truly causal. The key insight of AID is that we can refine the AC-DAG and prune the non-causal nodes and edges through a sequence of interventions. To intervene on a predicate, AID changes the application's behavior through fault injection so that the predicate's value matches its value in successful executions. If the failure does not occur under the intervention, then, based on counterfactual causality, the predicate is guaranteed to be a root cause of the failure. Over several iterations, AID intervenes on a set of carefully chosen predicates, refines the set of discriminative predicates, and prunes the AC-DAG, until it discovers the true root cause and the path that leads to the failure. We describe the intervention mechanism of AID in Section 5.

We now describe how AID adapts existing approaches in SD and fault injection for two of its core ideas: predicates and interventions. We refer to our technical report [18] for additional details and discussion.

3.1 AID Predicates

Predicate design. Similar to traditional SD techniques, AID is effective only if the initial set of predicates (in the predicate logs) contains a *root-cause predicate* that causes the failure. Predicate design is orthogonal to AID. We use predicates used by existing SD techniques, especially the ones used for finding root causes of concurrency bugs [30], a key reason behind intermittent failures [47]. Figure 2 shows examples of predicates in AID (column 1).

Predicate extraction. AID automatically instruments a target application to generate its *execution trace* [18]. The trace contains each executed method's start and end time, its

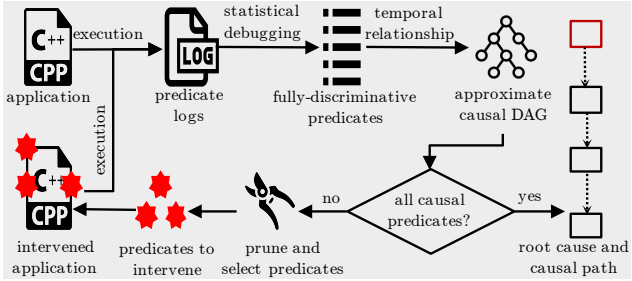

Figure 1: Adaptive Interventional Debugging workflow.

thread id, ids of objects it accesses, return values, whether it throws exception or not, and so on. This trace is then analyzed offline to evaluate a set of predicates at each execution point. This results in a sequence of predicates, called *predicate log*. The instrumented application is executed multiple times with the same input, to generate a set of predicate logs, each labeled as a successful or failed execution. Figure 2 shows the runtime conditions used to extract predicates (column 2).

Modeling nondeterminism. In practice, some predicates may cause a failure nondeterministically: predicates A and B in *conjunction* cause a failure. AID does not consider such predicates as they are not fully discriminative (recall < 100%). However, AID can still model these cases with *compound* predicates, adapted from state-of-the-art SD techniques [30], which model conjunctions. These compound predicates ("A and B") would *deterministically* cause the failure and hence be fully discriminative. Note that AID focuses on counterfactual causality and thus does not support *disjunctive* root causes (as they are not counterfactual). In Section 5, we discuss AID's assumptions and their impact in practice.

3.2 AID Interventions

Intervention mechanism. AID uses an existing fault injection tool (similar to LFI [48]) to intervene on fully-discriminative predicates; interventions change a predicate to match its value in a successful execution. In a way, AID's interventions try to locally "repair" a failed execution. Figure 2 shows examples of AID's interventions (column 3). Most of the interventions rely on changing timing and thread scheduling that can occur naturally by the underlying execution environment and runtime. More specifically, AID can slow down the execution of a method (by injecting delays), force or prevent concurrent method executions in different threads (by using synchronization primitives such as locks), change the execution order of concurrent threads (by injecting delays), etc. Such interventions can repair many concurrency bugs.

Validity of intervention. AID supports two additional intervention types, return-value alteration and exception handling, which, in theory, can have undesirable runtime side-effects. Consider two predicates: (1) method `QueryAvgSalary`

(1) Predicate	(2) Extraction condition	(3) Intervention mechanism
There is a data race involving methods M_1 and M_2	M_1 and M_2 temporally overlap accessing some object X while one of them is a `write`	Put locks around the code segments within M_1 and M_2 that access X
Method M fails	M throws an exception	Put M in a `try-catch` block
Method M runs too fast	M's duration is less than the minimum duration for M among all successful executions	Insert delay before M's `return` statement
Method M runs too slow	M's duration is greater than the maximum duration for M among all successful executions	Prematurely return from M the correct value that M returns in all successful executions
Method M returns incorrect value	M's return value $\neq x$, where x is the correct value returned by M in all successful executions	Alter M's `return` statement to force it to return the correct value x

Figure 2: Few example predicates, conditions used to extract them, and the corresponding interventions using fault injection.

fails returning `null` and (2) method `UpdateSalary` fails returning `error`. AID can intervene to match their return values in successful executions, e.g., `50` and `OK`, respectively. The intervention on the first predicate does not modify any program state and, as the successful execution shows, the return value `50` can be safely used by the application. However, altering the return value of `UpdateSalary`, but not updating the salary, may not be sufficient intervention: other parts of the application that rely on the updated salary may fail. Inferring such side-effects is hard, if not impossible.

AID is restricted to safe interventions. It relies on developers to indicate which methods do not change (internal or external) application states and limits return-value interventions to only those methods (e.g., to `QueryAvgSalary`, but not to `UpdateSalary`). The same holds for exception-handling interventions. AID removes from predicate logs any predicates that cannot be safely intervened without undesirable side-effects. This ensures that the rest of the AID pipeline can safely intervene on any subset of predicates. Excluding some interventions may limit AID's precision, as it may eliminate a root-cause predicate. In such cases, AID may find another intervenable predicate that is causally related to the root cause, and is still useful for debugging. In our experiments (Section 7) we did not observe this issue, since the root-cause predicates were safe to intervene.

4 APPROXIMATING CAUSALITY

AID relies on traditional SD to derive a set of fully-discriminative predicates. Using the logs of successful and failed executions, AID extracts temporal relationships among these predicates, and uses temporal precedence to approximate causality. It is clear that in the absence of feedback loops, a cause temporally precedes an effect [59]. To handle loops, AID considers multiple executions of the same program statement (e.g., within a loop, recursion, or multiple method calls) as separate instances, identified by their relative order of appearances during program execution, and maps them to separate predicates [18]. This ensures that temporal precedence among predicates correctly *over-approximates* causality.

Approximate causal DAG. AID represents the approximation of causality in a DAG: each node represents a predicate, and an edge $P_1 \rightarrow P_2$ indicates that P_1 temporally precedes P_2 in *all* logs where both predicates appear. Figure 4(a) shows an example of the approximate causal DAG (AC-DAG). We use circles to explicitly depict *junctions* in the AC-DAG; junctions are not themselves predicates, but denote splits or merges in the precedence ordering of predicates. Therefore, each predicate has in- and out-degrees of at most 1, while junctions have in- or out-degrees greater than 1. For clarity of visuals, in our depictions of the AC-DAG, we omit edges implied by transitive closure. For example, there exists an edge $P_3 \rightarrow P_5$, implied by $P_3 \rightarrow P_4$ and $P_4 \rightarrow P_5$, but it is not depicted. AID enforces an assumption of counterfactual causality by excluding from the AC-DAG any predicates that were not observed in *all* failed executions: if some executions failed without manifesting P, then P cannot be a cause of the failure.

Completeness of AC-DAG. The AC-DAG is complete with respect to the available, and safely-intervenable, predicates: it contains all fully-discriminative predicates that are safe to intervene, and if P_1 causes P_2, it includes the edge $P_1 \rightarrow P_2$. However, it may not be complete with respect to all possible true root causes, as a root cause may not always be represented by the available predicates (e.g., if the true root cause is a data race and no predicate is used to capture it). In such cases, AID will identify the (intervenable) predicate that is closest to the root cause and is causally related to the failure.

Since temporal precedence among predicates is a necessary condition for causality, the AC-DAG is guaranteed to contain the true causal relationships. However, temporal precedence is not sufficient for causality, and thus some edges in the AC-DAG may not be truly causal.

Temporal precedence. Capturing temporal precedence is not always straightforward. For simplicity of implementation, AID relies on computer clocks, which works reasonably well in practice. Relying on computer clocks is not always precise as the time gap between two events may be too small for the granularity of the clock; moreover, events may occur on different cores or machines whose clocks are not perfectly synchronized. These issues can be addressed with the use of logical clocks such as Lamport's Clock [39].

Another challenge is that some predicates are associated with time windows, rather than time points. The correct

policy to resolve temporal precedence of two temporally overlapping predicates often depends on their semantics. However, the predicate types give important clues regarding the correct policy. In AID, predicate design involves specifying a set of rules that dictates the temporal precedence of two predicates. In constructing the AC-DAG, AID uses those rules.

For example, suppose that `foo()` calls `bar()` and waits for `bar()` to end—so, `foo()` starts *before* but ends *after* `bar()`.

- (Case 1): Consider two predicates P_1: "`foo()` is running slow" and P_2: "`bar()` is running slow". Here, P_2 can cause P_1 but not the other way around. In this case, AID uses the policy that *end-time implies temporal precedence*.
- (Case 2): Now consider P_1: "`foo()` starts later than expected" and P_2 : "`bar()` starts later than expected". Here, P_1 can cause P_2 but not the other way around. Therefore, in this case, *start-time implies temporal precedence*.

AID works with any policy of deciding precedence, as long as it does not create cycles in the AC-DAG. Since temporal precedence is a necessary condition for causality, any conservative heuristic for deriving temporal precedence would work. A conservative heuristic may introduce more false positives (edges that are not truly causal), but those will be pruned by interventions (Section 5).

5 CAUSAL INTERVENTION

In this section, we describe AID's core component, which refines the AC-DAG through a series of *causal interventions*. An intervention on a predicate forces the predicate to a particular state; the execution of the application under the intervention asserts or contradicts the causal connection of the predicate with the failure, and AID prunes the AC-DAG accordingly. Interventions can be costly, as they require the application to be re-executed. AID minimizes this cost by (1) smartly selecting the proper predicates to intervene, (2) grouping interventions that can be applied in a single application execution, and (3) aggressively pruning predicates even without direct intervention, but based on outcomes of other interventions. Figure 3 summarizes the notations used in this section.

We start by formalizing the problem of *causal path discovery* and state our assumptions (Section 5.1). Then we provide an illustrative example to show how AID works (Section 5.2). We proceed to describe *interventional pruning* that AID applies to aggressively prune predicates during group intervention rounds (Section 5.3). Then we present AID's causality-guided group intervention algorithm (Section 5.4) which administers group interventions to derive the causal path.

5.1 Problem Definition and Assumptions

Given an application that intermittently fails, our goal is to provide an informative explanation for the failure. To that end, given a set of fully-discriminative predicates $\mathcal{P}$, we want to find an ordered subset of $\mathcal{P}$ that defines the causal path from the root-cause predicate to the predicate indicating the failure. Informally, AID finds a chain of predicates that starts from the root-cause predicate, ends at the failure predicate, and contains the maximal number of explanation predicates such that each is caused by the previous one in the chain. We address the problem in a similar setting as SD, and make the following two assumptions:

Notation	Description
$\mathcal{G}$	Approximate causal DAG (AC-DAG)
$\mathbb{P}$	Causal path
F	Failure indicating predicate
$\mathcal{P}$	Set of predicates
$P(r)$	Predicate P is observed in execution r
$\neg P(r)$	Predicate P is not observed in execution r
$P_1 \rightsquigarrow P_2$	There is a path from P_1 to P_2 in $\mathcal{G}$

Figure 3: Summary of notations used in Section 5.

Assumption 1 (Single Root-cause Predicate). The root cause of a failure is the predicate whose absence (i.e., a value of `false`) certainly avoids the failure, and there is no other predicate that causes the root cause. We assume that in all the failed executions, there is exactly one root-cause predicate.

This assumption is prevalent in the SD literature [30, 42, 44], and is supported by several studies on real-world concurrency bug characteristics [46, 65, 68], which show that a vast majority of root causes can be captured with reasonably simple single predicates [18]. In practice, even with specific inputs, a program may fail in multiple ways. However, failures by the same root cause generate a unique *failure signature* and hence can be grouped together using metadata (e.g., stack trace of the failure, location of the failure in the program binary, etc.) collected by failure trackers [21]. AID can then treat each group separately, targeting a single root cause for a specific failure. Moreover, the single-root-cause assumption is reasonable in many simpler settings such as unit tests that exercise small parts of an application.

Note that this assumption does not imply that the root cause consists of a single event; a predicate can be arbitrarily complex to capture multiple events. For example, the predicate "there is a data race on X" is `true` when two threads access the same shared memory X at the same time, the accesses are not lock-protected, and one of the accesses is a write operation. Whether a single predicate is sufficient to capture the root cause depends on predicate design, which is orthogonal to AID. AID adapts the state-of-the art predicate design, tailored to capture root causes of concurrency bugs [30], which is sophisticated enough to capture all common root causes using single predicates. If no single predicate captures the true root cause, AID still finds the predicate closest to the true root cause in the true causal path.

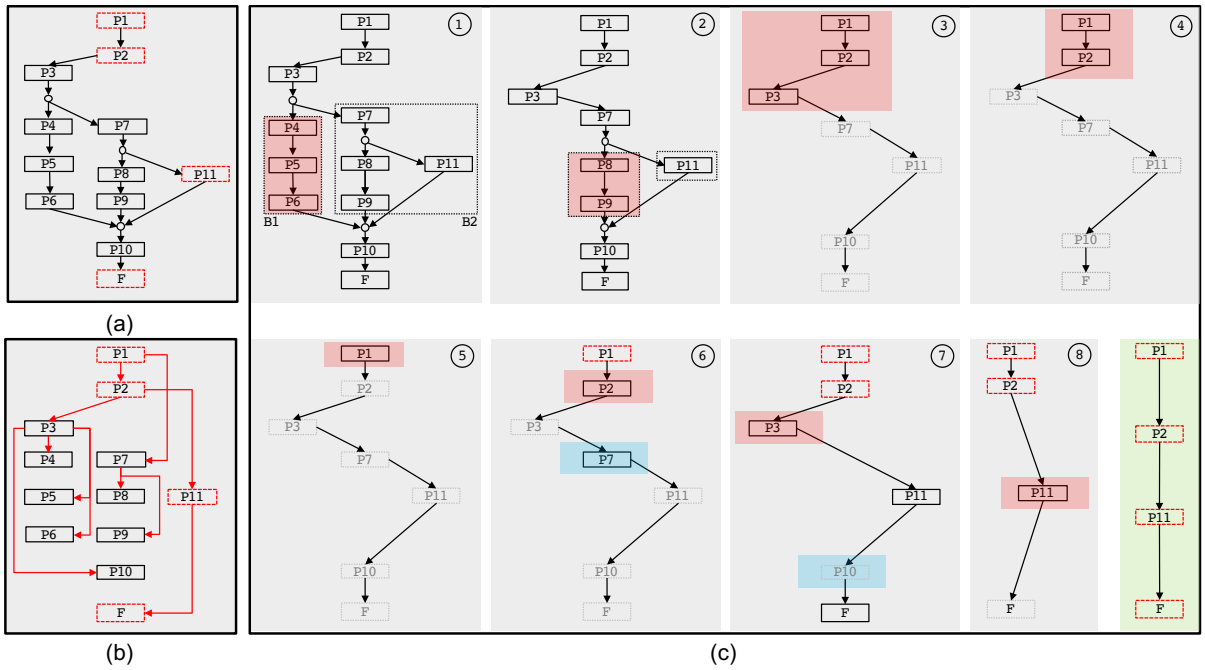

Figure 4: (a) The AC-DAG includes all edges implied by transitive closure, but we omit them for clarity of the visuals. We indicate the predicates in the causal path with the dashed red outline. (b) The actual causal DAG is a subgraph of the AC-DAG. (c) Step by step illustration to discover the causal path (shown at bottom right). Steps ① and ② perform branch pruning, steps ③–⑧ perform group intervention with pruning on the predicate chain, steps ⑥ and ⑦ apply interventional pruning.

Assumption 2 (Deterministic Effect). A root-cause predicate, if triggered, causes a fixed sequence of intermediate predicates (i.e., effects) before eventually causing the failure. We call this sequence *causal path*, and we assume that there is a unique one for each root-cause-failure pair.

Prior work has considered, and shown evidence of, a unique causal path between a root cause and the failure in sequential applications [31, 63]. The unique causal path assumption is likely to hold in concurrent applications as well for two key reasons. First, the predicates in AID's causal path may remain unchanged, despite nondeterminism in the underlying instruction sequence. For example, the predicate "there is a data race between methods X and Y" is not affected by which method starts first, as long as they temporally overlap. Second, AID only considers fully-discriminative predicates. If such predicates exist to capture the root cause and its effects, by the definition of being fully discriminative, there will be a unique causal path (of predicates) from the root cause to the failure. In all six of our real-world case studies (Section 7), such predicates existed and there were unique causal paths from the root causes to the failures.

Note that it is possible to observe some degree of disjointness within the true causal paths. For example, consider a case where the root cause C triggers the failure F in two ways: in some failed executions, the causal path is $C \rightarrow A_1 \rightarrow B \rightarrow F$ and, for others, $C \rightarrow A_2 \rightarrow B \rightarrow F$. This implies that neither A_1 nor A_2 is fully discriminative. Since AID only considers fully-discriminative predicates, both of them are excluded from the AC-DAG. In this case, AID reports $C \rightarrow B \rightarrow F$ as the causal path; this is the shared part of the two causal paths, which includes all counterfactual predicates and omits any disjunctive predicates. One could potentially relax this assumption by encoding the interaction of such predicates through a fully-discriminative predicate (e.g., $A = A_1 \vee A_2$ encodes disjunction and is fully discriminative).

Based on these assumptions, we define the causal path discovery problem formally as follows.

Definition 5.1 (Causal Path Discovery)**.** *Given an approximate causal DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a predicate $F \in \mathcal{V}$ indicating a specific failure, the* ***causal path discovery*** *problem seeks a path $\mathbb{P} = \langle C_0, C_1, \ldots, C_n \rangle$ such that the following conditions hold:*

- *C_0 is the root cause of the failure and $C_n = F$.*
- *$\forall\, 0 \leq i \leq n, C_i \in \mathcal{V}$ and $\forall\, 0 \leq i < n, (C_i, C_{i+1}) \in \mathcal{E}$.*
- *$\forall\, 0 \leq i < j \leq n, C_i$ is a counterfactual cause of C_j.*
- *$|\mathbb{P}|$ is maximized.*

5.2 Illustrative Example

AID performs causal path discovery through an intervention algorithm (Section 5.4). Here, we illustrate the main steps and intuitions through an example. Figure 4(a) shows an

AC-DAG derived by AID (Section 4). The true causal path for the failure F is $P1 \rightarrow P2 \rightarrow P11 \rightarrow F$, depicted with dashed red outline. The AC-DAG is a superset of the actual causal graph, which is shown in Figure 4(b).

AID follows an intervention-centric approach for discovering the causal path. Intervening on a predicate *forces* it to behave the way it does in the successful executions, which is by definition, the opposite of the failed executions. (Recall that, without loss of generality, we assume that all predicates are boolean.) Following the adaptive group testing paradigm, AID performs group intervention, which is simultaneous intervention on a set of predicates, to reduce the total number of interventions. Figure 4(c) shows the steps of the intervention algorithm, numbered ①–⑧.

AID first aims to reduce the AC-DAG by pruning entire chains that are not associated with the failure, through a process called *branch pruning* (Section 5.4). Starting from the root of the AC-DAG, AID discovers the first junction, after predicate $P3$. For each child of a junction, AID creates a compound predicate, called an *independent branch*, or simply *branch*, that is a disjunction over the child and all its descendants that are not descendants of the other children. So, for the junction after $P3$, we get branches $B1 = P4 \vee P5 \vee P6$ and $B2 = P7 \vee P8 \vee P9 \vee P11$. AID intervenes on one of the branches chosen at random—in this case $B1$—at step ①; this requires an intervention on all of its disjunctive predicates ($P4$, $P5$, and $P6$) in order to make the branch predicate `False`. Despite the intervention, the program continues to fail, and AID prunes the entire branch of $B1$, resolving the junction after $P3$. For a junction of B branches, AID would need $\log B$ interventions to resolve it using a divide-and-conquer approach. At step ②, AID similarly prunes a branch at the junction after $P7$. At this point, AID is done with branch pruning since it is left with just a chain of predicates (step ③).

What is left for AID is to prune any non-causal predicate from the remaining chain. AID achieves that through a divide-and-conquer strategy that intervenes on groups of predicates at a time (Algorithm 1). It intervenes on the top half of the chain—$\{P1, P2, P3\}$—which stops the failure and confirms that the root cause is in this group (step ③). With two more steps that narrow down the interventions (steps ④ and ⑤), AID discovers that $P1$ is the root cause. Note that we cannot simply assume that the root of the AC-DAG is a cause, because the edges are not all necessarily causal.

AID now needs to derive the causal path. Continuing the divide-and-conquer steps, it intervenes on $P2$ (step ⑥). This stops the failure, confirming that $P2$ is in the causal path. In addition, since $P7$ is not causally dependent on $P2$, the intervention on $P2$ does not stop $P7$ from occurring. This observation allows AID to prune $P7$ without intervening on it directly. At step ⑦, AID intervenes on $P3$. Under this intervention, the failure is still observed, but $P10$ no longer occurs, indicating that $P10$ is causally connected to $P3$, but not to the failure; this allows AID to prune both $P3$ and $P10$. Finally, at step ⑧, AID intervenes on $P11$ and confirms that it is causal, completing the causal path derivation. AID discovered the causal path in 8 interventions, while naïvely we would have needed 11—one for each predicate.

5.3 Predicate Pruning

In the initial construction of the AC-DAG, AID excludes predicates based on a simple rule: a predicate P is excluded if there exists a program execution r, where P occurs and the failure does not ($P(r) \wedge \neg F(r)$), or P does not occur and the failure does ($\neg P(r) \wedge F(r)$). Intervening executions follow the same basic intuition for pruning the intervened predicate C: By definition C does not occur in an execution r_C that intervenes on predicate C ($\neg C(r_C)$); thus, if the failure still occurs on r_C ($F(r_C)$), then C is pruned from the AC-DAG.

As we saw in the illustrative example, intervention on a predicate C may also lead to the pruning of additional predicates. However, the same basic pruning logic needs to be applied more carefully in this case. In particular, we can never prune predicates that precede C in the AC-DAG, as their potential causal effect on the failure may be muted by the intervention on C. Thus, we can only apply the pruning rule to any predicate X that is not an ancestor of C in the AC-DAG ($X \not\rightsquigarrow C$). We formalize the predicate pruning strategy over $\mathcal{G}(\mathcal{V}, \mathcal{E})$ in the following definition.

Definition 5.2 (Interventional Pruning)**.** *Let R_C be a set of program executions*[1] *intervening on a group of predicates $\mathcal{C} \subseteq \mathcal{V}$. Every $C \in \mathcal{C}$ is pruned from $\mathcal{G}$ iff $\exists r \in R_C$ such that $F(r)$. Any other predicate $P \notin \mathcal{C}$ is pruned from $\mathcal{G}$ iff $\nexists C \in \mathcal{C}$ such that $P \rightsquigarrow C$ and $\exists r \in R_C$ such that $(P(r) \wedge \neg F(r)) \vee (\neg P(r) \wedge F(r))$.*

5.4 Causality-guided Intervention

AID's core intervention method is described in Algorithm 1: Group Intervention With Pruning (GIWP). GIWP applies adaptive group testing to derive causal and spurious (non-causal) nodes in the AC-DAG. The algorithm applies a divide-and-conquer approach that groups predicates based on their topological order (a linear ordering of its nodes such that for every directed edge $P_1 \rightarrow P2$, P_1 comes before P_2 in the ordering). In every iteration, GIWP selects the predicates in the lowest half of the topological order, resolving ties randomly, and intervenes by setting all of them to `False` (lines 4–5). The intervention returns a set of predicate logs.

If the failure is not observed in any of the intervening executions (line 6), based on counterfactual causality, GIWP

[1]Because of nondeterminism issues in concurrent applications, we execute a program multiple times with the same intervention. However, it is sufficient to identify a single counter-example execution to invoke the pruning rule.

Algorithm 1: GIWP ($\mathcal{P}, \mathcal{G}, F$)

Input : A set of candidate predicates, $\mathcal{P}$,
AC-DAG, $\mathcal{G}$
Failure indicating predicate, F
Output : The set of counterfactual causes of F, $\mathcal{C}$
The set of spurious predicates, $\mathcal{X}$

1 $\mathcal{C} = \emptyset$ /* causal predicate set */
2 $\mathcal{X} = \emptyset$ /* spurious predicate set */
3 **while** $\mathcal{P} \neq \emptyset$ **do**
4 $\mathcal{P}_1$ = first half of $\mathcal{P}$ in topological order
5 $R_{\mathcal{P}_1}$ = Intervene ($\mathcal{P}_1$)
6 **if** $\nexists r \in R_{\mathcal{P}_1}$ *s.t.* $F(r)$ **then** /* failure stopped */
7 **if** $\mathcal{P}_1$ *contains a single predicate* **then**
8 $\mathcal{C} = \mathcal{C} \cup \mathcal{P}_1$ /* a cause is confirmed */
9 **else** /* need to confirm causes */
10 $\mathcal{C}', \mathcal{X}' = \textbf{GIWP}(\mathcal{P}_1, \mathcal{G}, F)$
11 $\mathcal{C} = \mathcal{C} \cup \mathcal{C}'$ /* confirmed causes */
12 $\mathcal{X} = \mathcal{X} \cup \mathcal{X}'$ /* spurious predicates */
/* interventional pruning */
13 **if** $\exists r \in R_{\mathcal{P}_1}$ *s.t.* $F(r)$ **then** /* failure didn't stop */
14 $\mathcal{X} = \mathcal{X} \cup \mathcal{P}_1$ /* pruning */
15 **foreach** $P \in \mathcal{P} - \mathcal{P}_1$ *s.t.* $\forall P' \in \mathcal{P}_1 \;\; P \not\rightsquigarrow P'$ **do**
16 **if** $\exists r \in R_{\mathcal{P}_1}$ *s.t.* $(P(r) \wedge \neg F(r)) \vee (\neg P(r) \wedge F(r))$ **then**
17 $\mathcal{X} = \mathcal{X} \cup \{P\}$ /* pruning */
18 $\mathcal{P} = \mathcal{P} - (\mathcal{C} \cup \mathcal{X})$ /* remove confirmed and spurious predicates from candidate predicate pool */
19 **return** $\mathcal{C}, \mathcal{X}$

concludes that the intervened group contains at least one predicate that causes the failure. If the group contains a single predicate, it is marked as causal (line 8). Otherwise, GIWP recurses to trace the causal predicates within the group (line 10). During each intervention round, GIWP applies Definition 5.2 to prune predicates that are determined to be non-causal (lines 13–17). First, if the algorithm discovers an intervening execution that still exhibits the failure, then it labels all intervened predicates as spurious and marks them for removal (line 14). Second, GIWP examines each other predicate that does not precede any intervened predicate and observes if any of the intervened executions demonstrate a counterfactual violation between the predicate and the failure. If a violation is found, that predicate is pruned (line 17).

At completion of each intervention round, GIWP refines the predicate pool by eliminating all confirmed causes and spurious predicates (Line 18) and enters the next intervention round . It continues the interventions until all predicates are either marked as causal or spurious and the remaining predicate pool is empty. Finally, GIWP returns two disjoint predicate sets—the causal predicates and the spurious predicates (Line 19).

Branch Pruning. GIWP is sufficient for most practical applications and can work directly on the AC-DAG. However,

Algorithm 2: Branch-Prune ($\mathcal{G}, F$)

Input : AC-DAG, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Failure indicating predicate, F
Output : Reduces $\mathcal{G}$ to an approximate causal chain

1 $\mathcal{C} = \emptyset$ /* potential causal predicate set */
2 $\mathcal{X} = \emptyset$ /* spurious predicate set */
3 **while** $\mathcal{V} - \mathcal{C} \neq \emptyset$ **do**
4 $\mathcal{P}$ = predicates at the lowest topological level in $\mathcal{V} - \mathcal{C}$
5 **if** $\mathcal{P}$ *contains a single predicate* **then**
6 $\mathcal{C} = \mathcal{C} \cup \mathcal{P}$ /* add to potential causal set */
7 **else** /* this is a junction */
8 $\mathcal{B} = \emptyset$
9 **foreach** $P \in \mathcal{P}$ **do**
10 $\mathcal{B}_P = \bigvee\{Q : P \rightsquigarrow Q \wedge \forall P' \in \mathcal{P} - \{P\}\; P' \not\rightsquigarrow Q\}$
11 $\mathcal{B}_P = P \vee \mathcal{B}_P$
12 $\mathcal{B} = \mathcal{B} \cup \{\mathcal{B}_P\}$ /* set of branches */
13 $\mathcal{C}', \mathcal{X}' = \textbf{GIWP}\,(\mathcal{B}, \mathcal{G}, F)$
14 $\mathcal{C} = \mathcal{C} \cup \mathcal{C}'$ /* add to potential causal set */
15 $\mathcal{X} = \mathcal{X} \cup \mathcal{X}'$ /*add to spurious set */
16 $\mathcal{U} = \{U : \mathcal{C} \neq \emptyset \wedge \forall C \in \mathcal{C}\; C \not\rightsquigarrow U\}$ /*unreachable*/
17 $\mathcal{V} = \mathcal{V} - \mathcal{X}$ /* remove spurious predicates */
18 $\mathcal{V} = \mathcal{V} - \mathcal{U}$ /* remove unreachable predicates */

when the AC-DAG satisfies certain conditions (analyzed in Section 6.2.2), we can reduce the number of required interventions through a process called *branch pruning*. The intuition is that since there is a single causal path that explains the failure, junctions (where multiple paths exist) can be used to quickly identify independent branches to be pruned or confirmed as causal as a group. The branches can be used to more effectively identify groups for intervention, reducing the overall number of required interventions. AID optionally invokes branch pruning before the divide-and-conquer group intervention through GIWP.

Branch pruning iteratively prunes branches at junctions (steps ① and ② in the illustrative example) to reduce the AC-DAG to a chain of predicates. The process is detailed in Algorithm 2. The algorithm traverses the DAG based on its topological order, and does not intervene while it encounters a single node at a time, which means it is still in a chain (line 5). When it encounters multiple nodes at the same topological level, it means it encountered a junction (line 7). A junction means that the true causal path can only continue in one direction, and AID can perform group intervention to discover it. The algorithm invokes GIWP to perform this intervention over a set of special predicates constructed from the branches at the encountered junction (lines 10–12). A branch at predicate P is defined as a disjunctive predicate over P and all descendants of P that are not descendants of any other predicate at the same topological level as P. An example branch from our illustrative example is $B1 = P4 \vee P5 \vee P6$. To intervene on a branch, one has to

intervene on all of its disjunctive predicates. The algorithm defines $\mathcal{B}$ as the union of all branches, which corresponds to a completely disconnected graph (no edges between the nodes), thus all branch predicates are at the same topological level. GIWP is then invoked (line 13) to identify the causal branch. The algorithm removes any predicate that is not causally connected to the failure (line 17) or is no longer reachable from the correct causal chain (line 18), and updates the AC-DAG accordingly. At the completion of branch pruning, AID reduces the AC-DAG to simple chain of predicates.

6 THEORETICAL ANALYSIS

In this section, we theoretically analyze the performance of AID in terms of the number of interventions required to identify all *causal predicates*, which are the predicates causally related to the failure.[2] Similar to the analysis of group testing algorithms, we study the information-theoretic lower bound, which specifies the minimum number of bits of information that an algorithm must extract to identify all causal predicates for any instance of a problem. We also study the lower and the upper bounds that quantify the minimum and the maximum number of group interventions required to identify all causal predicates, respectively, for AID versus a Traditional Adaptive Group Testing (TAGT) algorithm.

Any group testing algorithm takes N items (predicates), D of which are faulty (causal), and aims to identify all faulty items using as few group interventions as possible. Since there are $\binom{N}{D}$ possible outcomes, the information-theoretic lower bound for this problem is $\log \binom{N}{D}$. The upper bound on the number of interventions using TAGT is $O(D \log N)$, since $\log N$ group interventions are sufficient to reveal each causal predicate. Here, we assume $D < \frac{N}{\log N}$; otherwise, a linear approach that intervenes on one predicate at a time is preferable.

We now show that the Causal Path Discovery (CPD) problem (Definition 5.1) can reduce the lower bound on the number of required interventions compared to Group Testing (GT). We also show that the upper bound on the number of interventions is lower for AID than TAGT, because of the two assumptions of CPD (Section 5.1). In TAGT, predicates are assumed to be independent of each other, and hence, after each intervention, decisions (about whether predicates are causal) can be made only about the intervened predicates. In contrast, AID uses the precedence relationships among predicates in the AC-DAG to (1) aggressively prune, by making decisions not only about the intervened predicates but also about other predicates, and to (2) select predicates based on the topological order, which enables effective pruning during each intervention.

[2]Causal predicates correspond to *faulty predicates* in group testing. This distinction in terminology is because group testing does not meaningfully reason about causality.

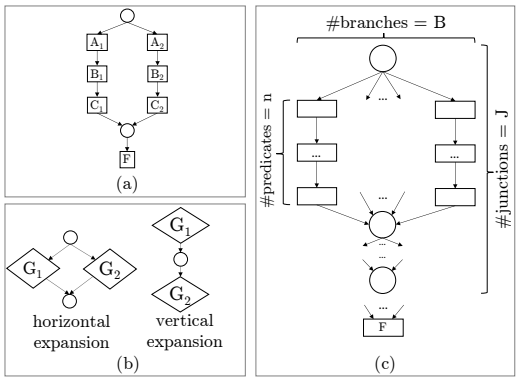

Figure 5: (a) An AC-DAG. (b) Horizontal and vertical expansion. (c) A symmetric AC-DAG with J junctions; each junction has B branches and each branch has n predicates.

Example 6.1. *Consider the AC-DAG of Figure 5(a), consisting of $N = 6$ predicates and the failure predicate F. If AID intervenes on all predicates in one branch (e.g., $\{A_1, B_1, C_1\}$) and finds causal connection to the failure, it can avoid intervening on predicates in the other branch according to the deterministic effect assumption. AID can also use the structure of the AC-DAG to intervene on A_1 (or A_2) before other predicates since the intervention can prune a large set of predicates. Since GT algorithms do not assume relationships among predicates, they can only intervene on predicates in random order and can make decisions about only the intervened predicates.*

6.1 Search Space

The temporal precedence and potential causality encoded in the AC-DAG restrict the possible causal paths and significantly reduce the search space of CPD compared to GT.

Example 6.2. *In the example of Figure 5(a), GT considers all subsets of the 6 predicates as possible solutions, and thus its search space includes $2^6 = 64$ candidates. CPD leverages the AC-DAG and the deterministic effect assumption (Section 5.1) to identify invalid candidates and reduce the search space considerably. For example, the candidate solution $\{A_1, B_2, C_1\}$ is not possible under CPD, because it involves predicates in separate paths on the AC-DAG. In fact, based on the AC-DAG, CPD does not need to explore any solutions with more than 3 predicates. The complete search space of CPD includes all subsets of predicates along each branch of length 3, thus a total of $2 \cdot (2^3 - 1) + 1 = 15$ possible solutions.*

We proceed to characterize the search space of CPD compared to GT more generally. We use $|G|$ to denote the number of predicates in an AC-DAG represented by G, and W_G^{GT} and W_G^{CPD} to denote the size of the search space for GT and CPD, respectively. We start from the simplest case of DAG, a chain, and then using the notions of horizontal and vertical expansion, we can derive the search space for any DAG.

If G is a simple chain of predicates, then GT and CPD have the same search space: $2^{|G|}$. CPD reduces the search space drastically when junctions split the predicates into separate branches, like in Example 6.2. We call this case a *horizontal expansion*: a DAG $\mathcal{G}_H$ is a horizontal expansion of two subgraphs G_1 and G_2 if it connects them in parallel through two junctions, at the roots (lowest topological level) and leaves (highest topological level). In contrast, $\mathcal{G}_V$ is a *vertical expansion*, if it connects them sequentially via a junction. Horizontal and vertical expansion are depicted in Figure 5(b). In horizontal expansion, the search space of CPD is additive over the combined DAGs, while in vertical expansion it is multiplicative.

Lemma 6.1 (DAG expansion). *Let $W^{CPD}_{G_1}$ and $W^{CPD}_{G_2}$ be the numbers of valid solutions for CPD over DAGs G_1 and G_2, respectively. Let $\mathcal{G}_H$ and $\mathcal{G}_V$ represent their horizontal and vertical expansion, respectively. Then:*

$$W^{CPD}_{\mathcal{G}_H} = 1 + (W^{CPD}_{G_1} - 1) + (W^{CPD}_{G_2} - 1)$$
$$W^{CPD}_{\mathcal{G}_V} = W^{CPD}_{G_1} W^{CPD}_{G_2}$$

In contrast, in both cases, the search space of GT is $2^{|G_1|+|G_2|}$.

Intuitively, in horizontal expansion, the valid solutions for $\mathcal{G}_H$ are those of G_1 and those from G_2, but no combinations between the two are possible. Note that both $W^{CPD}_{G_1}$ and $W^{CPD}_{G_2}$ have the empty set as a common solution, so in the computation of $W^{CPD}_{\mathcal{G}_H}$, one solution is subtracted from each search space ($W^{CPD}_{G_i} - 1$) and then added to the overall result.

Symmetric AC-DAG. Lemma 6.1 allows us to derive the size of the search space for CPD over any AC-DAG. To further highlight the difference between GT and CPD, we analyze their search space over a special type of AC-DAG, a symmetric AC-DAG, depicted in Figure 5(c). A symmetric AC-DAG has J junctions, and B branches at each junction, where each branch is a simple chain of n predicates. Therefore, the total number of predicates in the symmetric AC-DAG is $N = JBn$, and the search space of GT is $W^{GT} = 2^{JBn}$. For CPD, based on horizontal expansion, the subgraph in-between two subsequent junctions has a total of $1 + \sum_i^B (2^n - 1) = 1 + B(2^n - 1)$ candidate solutions. Then, based on vertical expansion, the overall search space of CPD is $W^{CPD} = (B(2^n - 1) + 1)^J$.

6.2 Bound on Number of Interventions

6.2.1 Lower bound. We now show that, due to the predicate pruning mechanisms, and the strategy of picking predicates according to topological order, the lower bound[3] on the required number of interventions in CPD is significantly reduced. For the sake of simplicity, we drop the deterministic effect assumption in this analysis. In GT, after each group test, we get at least 1 bit of information. Since after retrieving all information, the remaining information should be ≤ 0, therefore, the number of required interventions in GT is bounded below by $\log \binom{N}{D}$. In contrast, for CPD, we have the following theorem. (Proofs are in our technical report [18].)

[3]Lower bound is a theoretical bound which states that, it might be possible to design an algorithm that can solve the problem which requires number of steps equal to the lower bound. Note that, this does not imply that there exists one such algorithm.

	Search space	#Interventions	
		Lower bound	Upper bound (AID/TAGT)
CPD	$(B(2^n-1)+1)^J$	$\frac{JBn}{JBn+DS_1} \log \binom{JBn}{D}$	$J \log B + D \log (Jn) - \frac{D(D-1)S_2}{2Jn}$
GT	2^{JBn}	$\log \binom{JBn}{D}$	$D \log B + D \log (Jn) - \frac{D(D-1)}{2JBn}$

Figure 6: Theoretical comparison between CPD and GT for the symmetric AC-DAG of Figure 5(c).

Theorem 1. *The number of required group interventions in CPD is bounded below by $\frac{N}{N+DS_1} \log \binom{N}{D}$, where at least S_1 predicates are discarded (either pruned using the pruning rule or marked as causal) during each group intervention.*

Since $\frac{DS_1}{N} > 0$, we obtain a reduced lower bound for the number of required interventions in CPD than GT. In general, as S_1 increases, the lower bound in CPD decreases. Note that we are not claiming that AID achieves this lower bound for CPD; but this sets the possibility that improved algorithms can be designed in the setting of CPD than GT.

Symmetric AC-DAG. Figure 6 shows the lower bound on the number of required interventions in CPD and GT for the symmetric AC-DAG of Figure 5(c), assuming that each intervention discards at least S_1 predicates in CPD.

6.2.2 Upper bound. We now analyze the upper bound on the number of interventions for AID under (1) branch pruning, which exploits the deterministic effect assumption, and (2) predicate pruning.

Branch Pruning. Whenever AID encounters a junction, it has the option to apply branch pruning. In CPD, at most one branch can be causal at each junction; hence, we can find the causal branch using $\log B$ interventions at each junction, where B is the number of branches at that junction. Also, B is upper-bounded by the number of threads T in the program. This holds since we assume that the program inputs are fixed and there is no different conditional branching due to input variation in different failed executions within the same thread. If there are J junctions and at most T branches at each junction, the number of interventions required to reduce the AC-DAG to a chain is at most $J \log T$. Now let us assume that the maximum number of predicates in any path in the AC-DAG is N_M. Therefore, the chain found after branch pruning can contain at most N_M predicates. If D of them are causal predicates, we need at most $D \log N_M$ interventions to find them. Therefore, the total number of required interventions for AID is $\leq J \log T + D \log N_M$. In contrast, the number

of required interventions for TAGT, which does not prune branches, is $\leq D\log(TN_M) = D\log T + D\log N_M$. Therefore, whenever $J < D$, the upper bound on the number of interventions for AID is smaller than the upper bound for TAGT.

Predicate Pruning. For an AC-DAG with N predicates, D of which are causal, we now focus on the upper bound on the number of interventions in AID using only predicate pruning. In the worst case, when no pruning is possible, the number of required interventions would be the same as that of TAGT without pruning, i.e., $O(D\log N)$.

Theorem 2. *If at least S_2 predicates are discarded (pruned or marked as causal) from the candidate predicate pool during each causal predicate discovery, then the number of required interventions for AID is $\leq D\log N - \frac{D(D-1)S_2}{2N}$.*

Hence, the reduction depends on S_2. When $S_2 = 1$, we are referring to TAGT, in absence of pruning, because once TAGT finds a causal predicate, it removes that predicate from the candidate predicate pool.

Symmetric AC-DAG. Figure 6 shows the upper bound on the number of required interventions using AID and TAGT for the symmetric AC-DAG of Figure 5(c), assuming that at least S_2 predicates are discarded during each causal predicate discovery by AID.

7 EXPERIMENTAL EVALUATION

We now empirically evaluate AID. We first use AID on six real-world applications to demonstrate its effectiveness in identifying root cause and generating explanation on how the root cause causes the failure. Then we use a synthetic benchmark to compare AID and its variants against traditional adaptive group testing approach to do a sensitivity analysis of AID on various parameters of the benchmark.

7.1 Real-world Case Studies

We now use 3 real-world open-source applications and 3 proprietary applications to demonstrate AID's effectiveness in identifying root causes of transient failures. Figure 7 summarizes the results and highlights the key benefits of AID:

- AID is able to identify the true root cause and generate an explanation that is consistent with the explanation provided by the developers in corresponding GitHub issues.
- AID requires significantly fewer interventions than traditional adaptive group testing (TAGT), which does not utilize causality among predicates (columns 5 and 6).
- In contrast, SD generates a large number of discriminative predicates (column 3), only a small number of which is actually causally related to the failures (column 4).

7.1.1 Data race in Npgsql. As a case study, we first consider a recently discovered concurrency bug in Npgsql [53], an open-source ADO.NET Data Provider for PostgreSQL. The bug

(1) Application	(2) GitHub Issue #	(3) #Discrim. preds (SD)	(4) #Preds in causal path	#Interventions (5) AID	(6) TAGT
Npgsql [53]	2485 [54]	14	3	5	11
Kafka [33]	279 [34]	72	5	17	33
Azure Cosmos DB [14]	713 [15]	64	7	15	42
`Network`	N/A	24	1	2	5
`BuildAndTest`	N/A	25	3	10	15
`HealthTelemetry`	N/A	93	10	40	70

Figure 7: Results from real-world case studies. SD produces way too many spurious predicates beyond the correct causal predicates (columns 3 & 4). SD actually produces even more predicates, but here we only report the number of fully-discriminative predicates. AID and TAGT both pin-point the correct causal predicates using interventions, but AID does so with significantly fewer interventions (columns 5 & 6).

(GitHub issue #2485) causes an Npgsql-baked application to intermittently crash when it tries to create a new PostgreSQL connection. We use AID to check if it can identify the root cause and generate an explanation of how the root cause triggers the failure.

We used one of the existing unit tests in Npgsql that causes the issue, and generated logs from 50 successful executions and 50 failed executions of the test. By applying SD, we found a total of 14 discriminative predicates. However, SD did not pinpoint the root cause or generate any explanation.

We then applied AID on the discriminative predicates. In the branch pruning step, it used 3 rounds of interventions to prune 8 of the 14 predicates. In the next step, it required 2 more rounds of interventions. Overall, AID required a total of 5 intervention rounds; in contrast, TAGT would require 11 interventions in the worst case. After all the interventions, AID identified a data race as the root cause of the failure and produced the following explanation: (1) two threads race on an index variable: one increments it while the other reads it (2) The second thread accesses an array at the incremented index location, which is outside the array size. (3) This access throws `IndexOutOfRange` exception (4) Application fails to handle the exception and crashes. This explanation matches the root cause provided by the developer who reported the bug to Npgsql GitHub repository.

7.1.2 Use-after-free in Kafka. Next, we use AID on an application built on Kafka [33], a distributed message queue. On Kafka's GitHub repository, a user reported an issue [34] that causes a Kafka application to intermittently crash or hang. The user also provided a sample code to reproduce the issue; we use a similar code for this case study.

As before, we collected predicate logs from 50 successful and 50 failed executions. Using SD, we identified 72 discriminative predicates. The AC-DAG identified 30 predicates with no causal path to the failure indicating predicate, and hence were discarded. AID then used the intervention algorithm on

the remaining 42 predicates. After a sequence of 7 interventions, AID could identify the root-cause predicate. It took an additional 10 rounds (total 17) of interventions to discover a causal path of 5 predicates that connects the root cause and the failure. The causal path gives the following explanation: (1) The main thread that creates a Kafka `consumer` C starts a child thread (2) the child thread runs too slow before calling a method on C (3) main thread disposes C (4) child thread calls a commit method on C (5) since C has already been disposed by the main thread, the previous step causes an exception, causing the failure. The explanation matches well with the description provided in GitHub.

Overall, AID required 17 interventions to discover the root cause and explanation. In contrast, SD generates 72 predicates, without pinpointing the true root cause or explanation. TAGT could identify all predicates in the explanation, but it takes 33 interventions in the worst case.

7.1.3 Timing bug in Azure Cosmos DB application. Next, we use AID on an application built on Azure Cosmos DB [14], Microsoft's globally distributed database service for operational and analytics workloads. The application has an intermittent timing bug similar to the one mentioned in a Cosmos DB's pull request on GitHub [15]. In summary, the application populates a cache with several entries that would expire after 1 second, performs a few tasks, and then accesses one of the cached entries. During successful executions, the tasks run fast and end before the cached entries expire. However, a transient fault triggers expensive fault handling code that makes a task run longer than the cache expiry time. This makes the application fail as it cannot find the entry in the cache (i.e., it has already expired).

Using SD, we identified 64 discriminative predicates from successful and failed executions of the application. Applying AID on them required 15 interventions and it generated an explanation consisting of 7 predicates that are consistent with the aforementioned informal explanation. In contrast, SD would generate 64 predicates and TAGT would take 42 interventions in the worst case.

7.1.4 Bugs in proprietary software. We applied AID for finding root causes of intermittent failures of several proprietary applications inside Microsoft. We here report our experience with three of the applications that we name as follows (Figure 7): (1) `Network`: the control plane of a data center network, (2) `BuildAndTest`: a large-scale software build and test platform, and (3) `HealthTelemetry`: a module used by various services to report their runtime health. Parts of these applications (and associated tests) had been intermittently failing for several months and their developers could not identify the exact root causes. This highlights that the root causes of these failures were non-trivial. AID identified the root causes and generated explanations for how they lead to

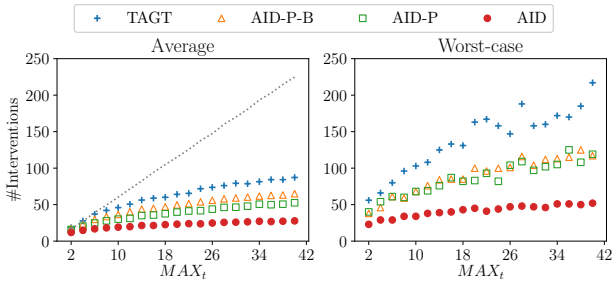

Figure 8: Number of interventions required in the average and worst case by TAGT and different variations of AID with varying MAX_t. For average case analysis, total number of predicates is shown using a grey dotted line. Total number of predicates is not shown for the worst-case analysis, because the worst cases vary across approaches.

failures: for `Network`, the root cause was a random number collision, for `BuildAndTest`, it was an order violation of two events, and for `HealthTelemetry`, it was a race condition. Developers of the applications confirmed that the root causes identified by AID are indeed the correct ones and that the explanations given by AID correctly showed how the root causes lead to the (intermittent) failures. Figure 7 shows the performance of AID with these applications. As before, SD produces many discriminative predicates, only a subset of which are causally related to the failures. Moreover, for all applications, AID requires significantly fewer interventions than what TAGT would require in the worse case.

7.2 Sensitivity Analysis

We further evaluate AID on a benchmark of synthetically-generated applications, designed to fail intermittently and with known root causes. We generate multi-threaded applications ranging the maximum number of threads MAX_t from 2 to 40. For each parameter setting, we generate 500 applications. In these applications, the total number of predicates N ranges from 4 to 284, and we randomly choose the number of causal predicates in the range $[1, \frac{N}{\log N}]$.

In this experiment, we compare TAGT, AID, AID without predicate pruning (AID-P), and AID without predicate or branch pruning (AID-P-B). All four approaches derive the correct causal paths but differ in the number of required interventions. Figure 8 shows the average (left) and the maximum (right) number of interventions required by each approach. The grey dotted line in the average case shows the average number of predicates over the 500 instances for that setting. This experiment provides two key observations:

Interventions in topological order converge faster. Causally-related predicates are likely to be topologically close to each other in the AC-DAG. AID discards all predicates in an intervened group only when none are causal. This is unlikely to occur when predicates are grouped randomly.

For this reason, AID-P-B, which uses topological ordering, requires fewer interventions than TAGT.

Pruning reduces the required number of interventions. We observe that both predicate and branch pruning reduce the number of interventions. Pruning is a key differentiating factor of AID from TAGT. In the worst-case setting in particular, the margin between AID and TAGT is significant: TAGT requires up to 217 interventions in one case, while the highest number of interventions for AID is 52.

8 RELATED WORK

Causal inference has been long applied for root-cause analysis of program failures. Attariyan et al. [3, 4] observe causality within application components through runtime control and data flow; but only report a list of root causes ordered by the likelihood of being faulty, without providing further causal connection between root causes and performance anomalies. Beyond statistical association (e.g., correlation) between root cause and failure, few techniques [5, 6, 19, 60] apply statistical causal inference on observational data towards software fault localization. However, observational data collected from program execution logs is often limited in capturing certain scenarios, and hence, observational study is ill-equipped to identify the intermediate explanation predicates. This is because observational data is not generated by randomized controlled experiments, and therefore, may not satisfy *conditional exchangeability* (data can be treated as if they came from a randomized experiment [28]) and *positivity* (all possible combinations of values for the variables are observed in the data)—two key requirements for applying causal inference on observational data [60]. While observational studies are extremely useful in many settings, AID's problem setting permits interventional studies, which offer increased reliability and accuracy.

Explanation-centric approaches are relevant to AID as they also aim at generating informative, yet minimal, explanations of certain incidents, such as data errors [66] and binary outcomes [20]; however they do not intervene. Viska [22] allows the users to perform intervention on system parameters to understand the underlying causes for performance differences across different systems. None of these systems are applicable for finding causally connected paths that explain intermittent failures due to concurrency bugs.

Statistical debugging approaches [13, 30, 37, 42, 44, 64, 72] employ statistical diagnosis to rank program predicates, extracted from execution traces of the program [10], based on their likelihood of being the root causes of program failures. However, all statistical debugging approaches suffer from the issue of not separating correlated predicates from the causal ones, and fail to provide contextual information regarding *how* the root causes lead to program failures.

Fault injection techniques [2, 24, 35, 48] intervene application runtime behavior with the goal to test if an application can handle the injected faults. In fault injection techniques, faults to be injected are chosen based on whether they can occur in practice. In contrast, AID intervenes with the goal of verifying (presence or absence of) causal relationship among runtime predicates, and faults are chosen based on if they can alter selected predicates.

Group testing [1, 7, 9, 17, 27, 36, 41] has been applied for fault diagnosis in prior literature [73]. Specifically, adaptive group testing is related to AID's intervention algorithm. However, none of the existing works considers the scenario where a group test might reveal additional information and thus offers an inefficient solution for causal path discovery.

Control flow graph-based techniques [12, 29] aim at identifying bug signature for sequential programs using discriminative subgraphs within the program's control flow graph; or generating faulty control flow paths that link many bug predictors. But these approaches do not consider causal connection among these bug predictors and program failure.

Differential slicing [31] aims towards discovering causal path of execution differences but requires complete program execution trace generated by execution indexing [70]. Dual slicing [67] discovers statement level causal paths for concurrent program failures. However, this approach does not consider compound predicates that capture certain runtime conditions observed in concurrent programs. Moreover, program slicing-based approaches cannot deal with a set of executions, instead they only consider two executions—one successful and one failed.

9 CONCLUSIONS

In this work, we defined the problem of causal path discovery for explaining failure of concurrent programs. Our key contribution is the novel Adaptive Interventional Debugging (AID) framework, which combines existing statistical debugging, causal analysis, fault injection, and group testing techniques in a novel way to discover root cause of program failure and generate the causal path that explains how the root cause triggers the failure. Such explanation provides better interpretability for understanding and analyzing the root causes of program failures. We showed both theoretically and empirically that AID is both efficient and effective to solve the causal path discovery problem. As a future direction, we plan to incorporate additional information regarding the program behavior to better approximate the causal relationship among predicates, and address the cases of multiple root causes and multiple causal paths. Furthermore, we plan to address the challenge of explaining multiple types of failures.

Acknowledgements: This work was supported by the NSF under grants IIS-1453543 and CCF-1763423, and by Oracle Labs.

REFERENCES

[1] Abhishek Agarwal, Sidharth Jaggi, and Arya Mazumdar. 2018. Novel Impossibility Results for Group-Testing. In *2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018*. 2579–2583.

[2] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 331–346.

[3] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. 307–320.

[4] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. 237–250.

[5] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2010. Causal Inference for Statistical Fault Localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. ACM, New York, NY, USA, 73–84.

[6] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2011. Mitigating the confounding effects of program dependences for effective fault localization. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 146–156.

[7] Yechao Bai, Qingsi Wang, Chun Lo, Mingyan Liu, Jerome P. Lynch, and Xinggan Zhang. 2019. Adaptive Bayesian group testing: Algorithms and performance. *Signal Processing* 156 (2019), 191–207.

[8] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2015. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1327–1342.

[9] Leonardo Baldassini, Oliver Johnson, and Matthew Aldridge. 2013. The capacity of adaptive group testing. In *Proceedings of the 2013 IEEE International Symposium on Information Theory, Istanbul, Turkey, July 7-12, 2013*. 2676–2680.

[10] Thomas Ball and James R. Larus. 1994. Optimally Profiling and Tracing Programs. *ACM Trans. Program. Lang. Syst.* 16, 4 (1994), 1319–1360.

[11] Antonio Bovenzi, Domenico Cotroneo, Roberto Pietrantuono, and Stefano Russo. 2012. On the aging effects due to concurrency bugs: A case study on MySQL. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 211–220.

[12] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. 2009. Identifying bug signatures using discriminative graph mining. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*. 141–152.

[13] Trishul M. Chilimbi, Ben Liblit, Krishna K. Mehra, Aditya V. Nori, and Kapil Vaswani. 2009. HOLMES: Effective statistical debugging via efficient path profiling. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. 34–44.

[14] Azure Cosmos DB. https://docs.microsoft.com/en-us/azure/cosmos-db/.

[15] CosmosDB bug. https://github.com/Azure/azure-cosmos-dotnet-v3/pull/713.

[16] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[17] Ding-Zhu Du and Frank K. Hwang. 1993. *Combinatorial group testing and its applications*. World Scientific, Singapore River Edge, N.J.

[18] Anna Fariha, Suman Nath, and Alexandra Meliou. 2020. Causality-Guided Adaptive Interventional Debugging. *CoRR* abs/2003.09539 (2020). arXiv:2003.09539

[19] Farid Feyzi and Saeed Parsa. 2017. Inference: Effective Fault Localization Based on Information-Theoretic Analysis and Statistical Causal Inference. *CoRR* abs/1712.03361 (2017). arXiv:1712.03361

[20] Kareem El Gebaly, Parag Agrawal, Lukasz Golab, Flip Korn, and Divesh Srivastava. 2014. Interpretable and Informative Explanations of Outcomes. *PVLDB* 8, 1 (2014), 61–72.

[21] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince R. Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen C. Hunt. 2009. Debugging in the (very) large: ten years of implementation and experience. In *SOSP*. 103–116.

[22] Helga Gudmundsdottir, Babak Salimi, Magdalena Balazinska, Dan R. K. Ports, and Dan Suciu. 2017. A Demonstration of Interactive Analysis of Performance Measurements with Viska. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 1707–1710.

[23] Joseph Y. Halpern and Judea Pearl. 2001. Causes and Explanations: A Structural-Model Approach: Part 1: Causes. In *UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, University of Washington, Seattle, Washington, USA, August 2-5, 2001*. 194–202.

[24] Seungjae Han, Kang G Shin, and Harold A Rosenberg. 1995. Doctor: An integrated software fault injection environment for distributed real-time systems. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*. IEEE, 204–213.

[25] Maurice Herlihy. 1992. A Methodology for Implementing Highly Concurrent Data Objects (Abstract). *Operating Systems Review* 26, 2 (1992), 12.

[26] Christopher Hitchcock. 2015. Conditioning, intervening, and decision. *Synthese* 193 (03 2015).

[27] F. K. Hwang. 1972. A Method for Detecting All Defective Members in a Population by Group Testing. *J. Amer. Statist. Assoc.* 67, 339 (1972), 605–608.

[28] David D. Jensen, Javier Burroni, and Matthew J. Rattigan. 2019. Object Conditioning for Causal Inference. In *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI 2019, Tel Aviv, Israel, July 22-25, 2019*. 393.

[29] Lingxiao Jiang and Zhendong Su. 2007. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. 184–193.

[30] Guoliang Jin, Aditya V. Thakur, Ben Liblit, and Shan Lu. 2010. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. 241–255.

[31] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. 2011. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 347–362.

[32] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*. 273–282.

[33] Kafka - A distributed streaming platform. https://kafka.apache.org.

[34] Kafka bug. https://github.com/confluentinc/confluent-kafka-dotnet/issues/279.

[35] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. 1995. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Trans. Computers* 44, 2 (1995), 248–260.

[36] Amin Karbasi and Morteza Zadimoghaddam. 2012. Sequential group testing with graph constraints. In *2012 IEEE Information Theory Workshop, Lausanne, Switzerland, September 3-7, 2012*. 292–296.

[37] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *SOSP*. ACM, 582–598.

[38] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-scale Industrial Setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. ACM, New York, NY, USA, 101–111.

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

[40] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. 2016. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. *ACM SIGPLAN Notices* 51, 4 (2016), 517–530.

[41] Tongxin Li, Chun Lam Chan, Wenhao Huang, Tarik Kaced, and Sidharth Jaggi. 2014. Group testing with prior statistics. In *2014 IEEE International Symposium on Information Theory, Honolulu, HI, USA, June 29 - July 4, 2014*. 2346–2350.

[42] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 15–26.

[43] Bo Liu, Zhengwei Qi, Bin Wang, and Ruhui Ma. 2014. Pinso: Precise Isolation of Concurrency Bugs via Delta Triaging. In *ICSME*. IEEE Computer Society, 201–210.

[44] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P. Midkiff. 2006. Statistical Debugging: A Hypothesis Testing-Based Approach. *IEEE Trans. Software Eng.* 32, 10 (2006), 831–848.

[45] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. 2007. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. 103–116.

[46] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*. 329–339.

[47] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 643–653.

[48] Paul D Marinescu and George Candea. 2009. LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 379–388.

[49] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. 2010. WHY SO? or WHY NO? Functional Causality for Explaining Query Answers. In *Proceedings of the Fourth International VLDB workshop on Management of Uncertain Data (MUD 2010) in conjunction with VLDB 2010, Singapore, September 13, 2010*. 3–17.

[50] Alexandra Meliou, Wolfgang Gatterbauer, Suman Nath, and Dan Suciu. 2011. Tracing data errors with view-conditioned causality. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. 505–516.

[51] Alexandra Meliou, Sudeepa Roy, and Dan Suciu. 2014. Causality and Explanations in Databases. *PVLDB* 7, 13 (2014), 1715–1716.

[52] MySQL. https://www.mysql.com/.

[53] Npgsql - .NET Access to PostgreSQL. https://www.npgsql.org.

[54] Npgsql bug. https://github.com/npgsql/npgsql/issues/2485.

[55] Lennart Oldenburg, Xiangfeng Zhu, Kamala Ramasubramanian, and Peter Alvaro. 2019. Fixed It For You: Protocol Repair Using Lineage Graphs. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*.

[56] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *ISSTA*. ACM, 199–209.

[57] Judea Pearl. 2000. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, New York, NY, USA.

[58] Judea Pearl. 2011. The algorithmization of counterfactuals. *Ann. Math. Artif. Intell.* 61, 1 (2011), 29–39.

[59] Judea Pearl and Thomas Verma. 1991. A Theory of Inferred Causation. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA, April 22-25, 1991*. 441–452.

[60] Gang Shu, Boya Sun, Andy Podgurski, and Feng Cao. 2013. MFL: Method-Level Fault Localization with Causal Inference. In *ICST*. IEEE Computer Society, 124–133.

[61] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*. 1–10.

[62] P. Spirtes, C. Glymour, and R. Scheines. 2000. *Causation, Prediction, and Search* (2nd ed.). MIT press.

[63] William N Sumner and Xiangyu Zhang. 2009. Algorithms for automatically computing the causal paths of failures. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 355–369.

[64] Aditya V. Thakur, Rathijit Sen, Ben Liblit, and Shan Lu. 2009. Cooperative crug isolation. In *Proceedings of the International Workshop on Dynamic Analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009), WODA 2009, Chicago, IL, USA, July, 2009*. 35–41.

[65] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. 2015. An empirical study of bugs in test code. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*. 101–110.

[66] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. 2015. Data X-Ray: A Diagnostic Tool for Data Errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 1231–1245.

[67] Dasarath Weeratunge, Xiangyu Zhang, William N. Sumner, and Suresh Jagannathan. 2010. Analyzing concurrency bugs using dual slicing. In *ISSTA*. ACM, 253–264.

[68] W Eric Wong and Vidroha Debroy. 2009. A survey of software fault localization. *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45* 9 (2009).

[69] James Woodward. 2003. *Making Things Happen: A Theory of Causal Explanation*. Oxford University Press.

[70] Bin Xin, William N. Sumner, and Xiangyu Zhang. 2008. Efficient program execution indexing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 238–248.

[71] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 249–265.

[72] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. 2006. Statistical debugging: simultaneous identification of multiple bugs. In *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*. 1105–1112.

[73] Alice X. Zheng, Irina Rish, and Alina Beygelzimer. 2005. Efficient Test Selection in Active Diagnosis via Entropy Approximation. In *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, Edinburgh, Scotland, July 26-29, 2005*. 675.