Active Statistical Debugging: An Intervention-based Framework for Explaining Incidents in Concurrent Programs

Anna Fariha

Abstract

The broad availability and relative affordability of public cloud resources have revolutionized application deployment in distributed settings. But aside from the ease of deployment, application development in such distributed settings brings forth important challenges. Particularly, failures in distributed applications are often triggered by concurrency bugs, such as interference and coordination issues. Such concurrency bugs are difficult to reproduce; therefore, the root causes of concurrent program failures are hard to identify, and even harder to explain. A number of prior work focused on identifying the causes of concurrent application failure by exploiting the statistical correlation between certain runtime events and failure. However, correlation does not always imply causality. Without succinct explanation of *how* the root cause eventually triggers failure, the developer might fail to draw the causal connection from the root cause to the failure. The goal of this work is to provide a succinct *explanation path* consisting of multiple causally related events that explains how a root cause leads to program failure. In this work, we present *Active Statistical Debugging*: a program intervention-based framework, to discover the causal relationship among runtime events and program failure. The framework applies causality-based intervention techniques to *control* or *treat* certain runtime events on top of the statistical debugging framework, and learns causal relationship from the intervention outcomes. Active statistical debugging discovers the correct explanation path, while keeping the number of interventions required low. At the core of this framework is an efficient algorithm built on *group-testing*, a field of applied mathematics. Our experiments show that program intervention is a promising first step towards explaining causes of program failures, motivating further research.

1 Introduction

Computer programs fail due to a variety of reasons. One of the most common causes of program failure is bug in source code, which makes a sequential program more likely to fail for certain set of inputs. Another cause of program failure is certain runtime conditions which mostly apply to concurrent programs. For the same input, a concurrent program might behave differently in different executions and only fail when certain runtime conditions, such as data race, are met. To understand such failures, we need to identify the fault, i.e., *why* the program failed; and the corresponding context, i.e., *how* the fault triggered failure.

While stack trace provides some context for understanding the root cause of program failures, it is limited to the sequence of program statements that led to the failure. However, program statements are not expressive enough to capture complex runtime scenarios such as concurrent access to the same memory address by multiple threads in an interleaved manner. Moreover, stack trace analysis tools report failure trace of a single run at a time and do not leverage discriminatory statistics within a set of failed and passed runs.

To overcome the shortcomings of stack-trace based root cause analysis, dynamic analysis based fault localization techniques, such as statistical debugging approaches [18, 24, 26, 16], use *predicates* to capture certain program behaviors or runtime events (e.g., a variable taking a particular value). These techniques identify the program faults by contrasting the evaluation statistics of the predicates between failed and passed program executions and produce a ranked list of discriminating predicates. The underlying assumption of such techniques is: developers have a perfect bug understanding, i.e., it is enough for them to examine a faulty statement in isolation, without any context, to reason about program failure. However, without any context, a ranked list of predicates is often overwhelming for the developer since often a large number of predicates show similar evaluation statistics and are equally discriminative. To understand the root cause of program failure, the developer needs succinct information regarding how the predicates are *causally* related to each other and the failure. A user study [28] on Tarantula [19], a debugging tool that produces rank list of suspicious statements, reports: *"simply giving the statement was not enough for the participants to understand the problem*

and that more context was needed, which made us conclude that perfect bug understanding is generally not a realistic assumption."

To discover concurrent program bugs, there exists work [16] that defines compound predicates to capture complex interactions among multiple threads (e.g., two threads modifying the same memory location concurrently). However, a shortcoming of this approach is that it does not provide any context (intermediate predicates) of a root cause towards failure. Another approach [8] looks for discriminating subgraphs within the set of program control flow graphs towards identifying bug signature. Although it provides a better context; however, the unit of bug signature is program statement which fails to model compound predicates such as concurrent access to an object. While there exists work [15] that provides context of program failure in the form of *high coverage faulty control flow paths* that link many bug predictors; however, this approach suffers from several shortcomings: (1) it cannot be trivially adopted for concurrent programs, (2) it does not incorporate the notion of *causality* among bug predictors, and (3) it does not guarantee to generate the optimal path in presence of multiple candidate paths.

Instead of reporting the highest scoring predicate [24, 26, 16] or context limited to high coverage control flow path [15], a sequence of causally connected predicates provide much richer explanation to program failures. However, to the best of our knowledge, none of the existing approaches model causality among predicates, nor they provide a sequence of predicates to explain the program failure. A good explanation of program failure is a sequence of predicates, where the first predicate is the root cause, each predicate causes the next predicate in the sequence, and the last predicate is the failure indicating symptom. In Section 2, we present motivating examples to show how both statistical debugging and control-flow-path-based approaches fail to provide succinct causal explanation of concurrent program failures.

In this work, we aim to identify and explain causes of failures observed in concurrent programs that behave non-deterministically. We start with a set of program executions that operated on the same set of inputs and each execution is labeled as either "passed" or "failed". Observed predicates within these program executions constitute potential root causes, failure indicating symptoms, and units of explanation. Beyond answering the question *why* the program fails, we provide answer to the question *how* it fails. Specifically, our goal is to (1) identify the faulty predicates in the programs, (2) capture the causal connection between the fault inducing root cause predicates, subsequent causal predicates, and the failure indicating symptom predicates, and (3) summarize the findings as *causal explanation paths*, a sequence of causally related predicates that connect the root causes to the failures.

To this end, we present *active statistical debugging*, which is an automatic program intervention framework built on top of statistical debugging. Through the interventions, we eliminate correlated but non-causal predicates and construct a causal path from the root cause to the failure indicating symptom. Inspired from the adaptive group-testing literature [14], the key idea is to optimally select a set of predicates, control or treat them through interventions, and observe how it affects the statistical behavior of the intervened executions. Beyond classical adaptive group-testing approach, we also apply two pruning techniques by exploiting the observed temporal precedence and other causal potentiality hints among predicates during runtime. We summarize our contributions below:

- We motivate and define the problem of discovering the *causal explanation path* of unexpected incidents in concurrent programs which provides a better interpretability during root cause analysis (Section 2). To that end, we provide a novel *active statistical debugging* framework, based on program intervention techniques, to pin point a set of predicates that both (1) discriminate failed runs from the passed ones, and (2) causally connect the root cause to the program failure.

- We use the *causal capability* relationship among predicates (Section 4), learned from the program execution logs, to develop a *causality assumption model* (Section 5). This model encodes information regarding the potential causal relationship among predicates, which serves as an additional source of information during program intervention.

- We provide an efficient and effective intervention algorithm in the *adaptive group-testing* paradigm to extract the correct causal path (Section 6). Using the causality assumption model, we provide a strategy to optimally choose the best set of interventions during adaptive group-testing. Compared to the classical adaptive group-testing algorithm, the strategy significantly reduces the number of required interventions using two *predicate pruning* mechanisms which we design. The pruning mechanisms exploit the causality assumption model to discard predicates based on prior observation only, and without any additional intervention.

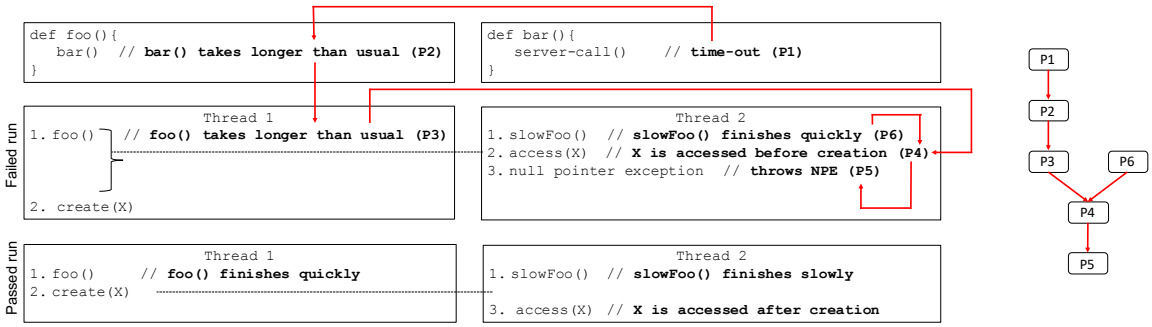

Figure 1: The failed run throws null pointer exception due to an attempt to access `X` which was not created by then. One potential root cause of such incident is a timed-out `server-call()` inside `bar()`. The other potential root cause is unexpected quick execution of `slowFoo()`. The causal dependencies among predicates is shown using red arrows in the right.

- Under the presence of the causality assumption model, we provide a *theoretical analysis* which quantifies the reduction of required information during intervention, in terms of the minimum number of interventions required (Section 7). The analysis shows that the information theoretic lower bound in our case is significantly lower than that of the uninformative group-testing.
- We empirically show that active statistical debugging is effective and efficient on a large set of synthetically generated concurrent programs (Section 8).

The rest of the report is organized as follows: Section 2 provides two motivating examples that contrast existing work with active statistical debugging. We state the problem settings and our assumptions in Section 3. Section 4 discusses predicates that encode the runtime events during program executions and serve as the basic units of active statistical debugging. Section 5 elaborates on reasoning about causality among the predicates. We describe the intervention algorithm that discovers the causal explanation path in Section 6, followed by its complexity analysis in Section 7. We provide results of the experimental evaluation of active statistical debugging in Section 8. We contrast active statistical debugging with the existing literature in Section 9 and finally conclude in Section 10.

2 Motivating Examples

In this section, we provide two motivating examples to demonstrate the shortcomings of the existing approaches in explaining the root causes of concurrent program failures and the advantage of active statistical debugging over those approaches. The example scenarios inspire us towards designing our solution for the causal explanation path discovery problem.

Example 2.1. We provide an example scenario where a program with two concurrent threads fails due to attempt to access a null pointer.

Scenario settings. In Figure 1, two threads, Thread 1 and Thread 2, are running concurrently. Six predicates are mined from the instrumented program: P1 – P6. When the program fails, it throws a null-pointer exception (P5). This was caused due to an attempt to access X which was not created by then (P4). Note that, P4 does not refer to the event "attempt to access X", rather it encodes the compound event "X is being accessed before creation". This particular event is never observed in the passed runs. P4 has two potential causes:

1. Slower execution of `foo()` (P3), which in turn was caused by slower execution of `bar()` (P2). The actual root cause here is P1: a timed-out `server-call()` inside `bar()`. The corresponding causal explanation path is: $P1 \rightarrow P2 \rightarrow P3 \rightarrow P4 \rightarrow P5$.
2. Faster execution of `slowFoo()` (P6). The corresponding causal explanation path is: $P6 \rightarrow P4 \rightarrow P5$.

In contrast, in the passed runs, `foo()` finishes quickly and `slowFoo()` finishes slowly. Hence X is created in Thread 1 before Thread 2 attempts to access it, which leads to normal execution.

```
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
```

```
A(){                  // main()
    B = getValue()    // getValue() returning 1 is the
                      //    root cause
    if B == 1:
        C()           // new thread 1
        D()           // new thread 2
        E
    if E happens:
        F
    if B == 1:
        G             // waits for C, D to finish
    if F happens:
        H             // throws exception
}
```

```
15.
16.
17.
```

```
C(){        // thread 1
    I
    J
}
```

```
18.
19.
20.
```

```
D(){        // thread 2
    K
    L
}
```

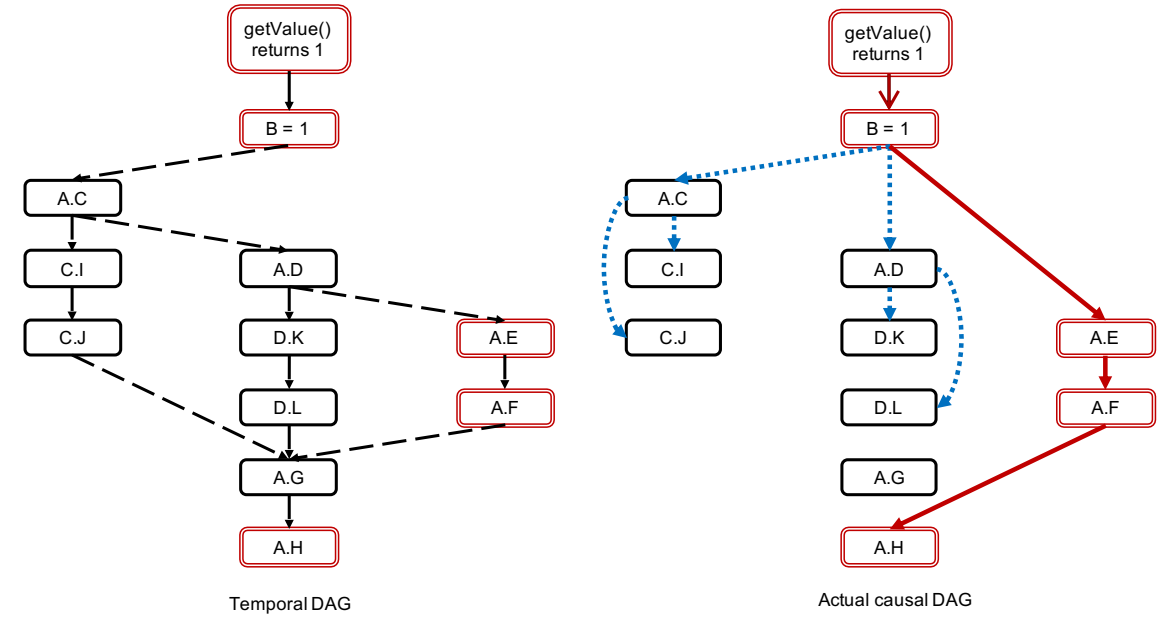

Figure 2: The three boxes on top show code segments. The program fails whenever `getValue()` returns `1`. The bottom-left DAG is a temporal precedence DAG, with transitive edges removed for the ease of exposition. The bottom-right DAG is the actual causal DAG. The right-most solid red path is the actual causal path.

Shortcomings of statistical debugging. Existing statistical debugging approaches will report P4 and P5 as root causes since they are the highest scoring predicates[1]. However, P5 is not useful at all in identifying the root cause of failure, since it is just a symptom of the failure. While P4 explains why P5 happens, it does not explain why P4 happens at the first place in the failed runs while it never happens in the passed ones. In fact, the root cause lies deep in the program flow, in this case, either P1 or P6 (or both). Even when P1 and P6 are reported as the root causes, without the intermediate predicates, the true scenario is obscured. For example, it is often hard for a developer to relate the timed-out `server-call()` (P1) with the program failure due to null pointer exception (P5).

In general, neither individual predicates nor rank-list of predicates are sufficient for deep understanding of program failures. The relationship between a root cause predicate and a failure indicating symptom predicate is often unclear without the context, i.e., the intermediate predicates that causally connect the root cause to the failure. One can sort all discriminating predicates by temporal order of observance to obtain the causal explanation paths, but without the causal connection, the two paths can be arbitrarily interleaved: $P6$ and $P3$ have no particular temporal order. Generally, in concurrent programs, there might not exist any temporal order of predicates that is consistent across all failed runs. Hence, temporal order-based technique results in reduced interpretability for the developer while analyzing root causes of program failures. Moreover, it does not provide any guarantee about the causal connection among predicates within the temporal order. We provide such a scenario in Example 2.2.

Example 2.2. We now provide another example where both statistical debugging approaches and control-flow-path-based approaches fail.

[1]Predicate score is computed using different metrics in the existing literature. However, they generally assign highest score to the predicates that always happen in the failed runs and never happen in the passed runs.

Scenario settings. In Figure 2, the program fails whenever `getValue()` returns 1, hence it is the root cause. Whenever the root cause triggers, two threads (`C` and `D`), unrelated to the failure indicating symptom (`H` at line 11), are also created. These two threads are unrelated to the bug, but only spawn during failed executions.

Shortcomings of statistical debugging. In this program, all predicates, including predicates within threads `C` and `D`, will show very strong correlation with failure and there is no way to rank them based on discriminating power. If we use the notion of *increase*[2] (defined in [24]), all predicates except the top two ("`getValue()` returning 1" and "`B` = 1") will be discarded. This is due to the fact that, [24] treats the two types of predicates equally — (1) predicates that are always observed and always true, and (2) predicates that are only observed in failed runs and turns out to be true. However, the latter type of predicates play a vital role in constructing the explanation path from the root cause to the failure indicating symptom.

Shortcomings of control flow path-based approach. Control flow path-based approach [15] identifies the control-flow path, that covers the most number of predicates, as the faulty control-flow path. While the technique works well for non-concurrent programs, it fails for concurrent programs. In this example, due to multithreading, three different control flow paths exist in the temporal DAG. Since [15] decides on the correct path based on high predicate coverage, i.e., linking more discriminating predicates, it will pick the sub-optimal middle branch involving predicates in `D`. Moreover, depending on the condition of line 10 (e.g., instead of `F`, it could be either `L` or `J`), the corresponding correct path would be different.

Example 2.2 shows that in cases where a lot of unrelated events happen during a program failure, both statistical and control-flow-path based approaches fail to report the correct set of predicates or control-flow path that are causally related to the failure. This inspires us to design active statistical debugging which decides on the causal explanation path based on the true causal connection among program predicates.

3 Problem Settings

In this section, we discuss the problem settings of the proposed active statistical debugging framework, provide our assumptions, and discuss the applicability of the framework. This work aims at identifying and explaining causes of failures observed in programs that behave non-deterministically for the same set of inputs. Therefore, the proposed framework is best suited for concurrent programs. However, the framework would also work for sequential programs when our assumptions hold, and produce correct result.

The input to our framework is a set of multiple program execution logs where each execution is labeled as either "passed" or "failed". From these execution logs, we mine discriminating predicates (discussed in Section 4) which constitute potential root causes, failure indicating symptoms, and units of explanation. The output of our system is a causal explanation path that explains why and how the program failed. We obtain this causal explanation path through active statistical debugging involving intervention on the discriminating predicates.

In this work, we make two assumptions:

- **Identical input**. We assume *identical input* to all program executions; i.e., there is no variance in program behavior due to input variability. This implies that in all passed (or failed) runs, a particular method instance, that depends only on the input data, returns a fixed value.
- **Exactly-one-causal-path**. We assume that there is only a *single type of failure*, *exactly one root cause*, and *exactly one causal path* that connects the root cause to the failure. The *exactly-one-causal-path* assumption implies that there is exactly one root cause which triggers other intermediate causes which eventually trigger failure. Hence, there is no disjunction among predicates that lead to multiple causal paths and there is no conjunctive set of predicates that jointly trigger failure.

These two assumptions enable us to design an efficient intervention algorithm for causal explanation path discovery. In Section 6.6, we discuss few possible extensions that relax some of the assumptions mentioned above, and how active statistical debugging can be modified to handle those extensions.

[2]For a predicate p: $increase(p) = \frac{\#\text{failed runs where p is observed to be True}}{\#\text{runs where p is observed to be True}} - \frac{\#\text{failed runs where p is observed}}{\#\text{runs where p is observed}}$

4 Predicates

Predicates are the building block of the causal explanation path, and units of intervention in the proposed active statistical debugging framework. Dynamic fault localization techniques instrument programs to mine predicates that encode the runtime behavior of program executions. Unlike static analysis, these techniques do not assume knowledge of program source. Active statistical debugging framework is built on the same principle of predicate-based dynamic analysis; the causal explanation path consists of a set of predicates that discriminate failed runs from the passed ones.

In this work, we extract predicates by post-processing the execution logs of instrumented programs. However, our framework is agnostic of the predicate extraction framework. It works with predicates extracted directly from the instrumented program executions [24, 31, 26] as well. In this section, we first discuss the predicates and proceed to two predicate design requirements required for active statistical debugging. Then we discuss the types of predicates we consider in this work, how we extract them from program execution logs, and the mechanism to intervene them. Finally, we discuss predicate scoring to narrow down on the discriminating predicates to start intervention with.

4.1 Design Requirement of Predicates

Predicates encode specific events that describe certain runtime conditions during program execution. Since the notion of increase [24] is not useful in our problem settings (explained in Example 2.2), we simply consider each predicate to be either true (observed and evaluated to be true) or false (unobserved, or observed and evaluated to be false). Prior statistical debugging approaches considered different types of predicates, but here we focus only on the predicate types that we can intervene on. Therefore, active statistical debugging is agnostic of the predicate design provided that the design fulfills the following two requirements:

- **Observation.** Given a predicate P and an execution log E, a mechanism to answer whether P is observed in E.
- **Intervention.** Given a predicate P and a program source S, a mechanism to *treat* P in S' such that P will never be observed in any execution of S'.

We discuss the impact of lack of intervenability in the discussion on extensions in Section 6.6.

4.2 Predicates Types

In this work, we consider four types of predicates. Our framework is not limited to these predicates, rather it can support any predicate that satisfy the two design requirements mentioned above. As an example, our framework can support predicates that represent runtime faults, used in the fault injection literature [13, 20]. We provide the description of the predicates that we consider in this work along with how they satisfy the two predicate design requirements below. Note that, in all cases, the goal of intervention is to treat the predicates so that it mimics the ideal behavior, i.e., the behavior observed in the passed runs.

1. **Method TTC anomaly**: Time to completion (TTC) of a method is the amount of time required for it to complete a task. We use two sub-types of predicates to express TTC anomalies — (1) slower TTC, and (2) faster TTC. This type of predicates indicates whether TTC of a method instance, is shorter/longer than ideal.

   • **Observation**: We learn the ideal TTC of method instances from the passed runs and use that to detect whether the predicate is observed in the failed runs.

   • **Intervention**: We use delay parameter to intervene. For slower TTC, if the method contains a delay construct (e.g., `Thread.Sleep(sleepTime)`), we intervene on the parameter `sleepTime` to reduce method latency. To intervene predicates with faster TTC, we inject delay within the method to ensure slower execution.

2. **Order violation**: This predicate encodes violation of particular order of accessing the same object by two methods from two different threads. We learn the ideal access order from the passed runs and mark the violation of the ideal order in the failed runs. Note that, concurrent access can also be a form of access order violation.

   • **Observation**: We observe the time signature of the access events which we extract from the execution logs to observe this type of predicates.

• **Intervention**: We use a delay parameter to inject delay before the violating access. As an example, if the ideal behavior is "$A()$ accesses X before $B()$", and the order violation is "$B()$ accesses X before $A()$", we inject delay before $B()$ to ensure the ideal access order. Similar technique can be used to intervene concurrent accesses as well.

3. **Return value anomaly**: This predicate encodes unexpected return value of a method instance. According to our problem settings and assumptions, a particular method instance that depends on the input data only, returns a fixed value in all passed (or failed) runs. For simplicity, we assume that the method return values are boolean, i.e., either True or False. However, this assumption can be relaxed to accommodate return value that follows a particular distribution, but this is orthogonal to our problem and we do not consider it in this work.

   • **Observation**: We observe the return values of method instances from the execution log.

   • **Intervention**: We force a function to return the "correct" value, i.e., the value that is consistent in the passed runs.

4. **Exception**: This predicate encodes whether a method instance throws an exception.

   • **Observation**: We observe this predicate by simply looking at the execution log.

   • **Intervention**: We include an exception handling mechanism to prevent the thrown exception from propagating further in the program flow.

4.3 Discriminating Predicates

The goal of finding discriminating predicates is to identify program behavior that deviates from the ideal behavior which is observed during the passed executions. We use two metrics, *precision* and *recall*, to capture the discriminatory power of predicates. We provide how we compute them below:

$$precision(P) = \frac{\#\text{failed runs where } P \text{ was observed}}{\#\text{runs where } P \text{ was observed}}$$

$$recall(P) = \frac{\#\text{failed runs where } P \text{ was observed}}{\#\text{failed runs}}$$

Both of these metrics capture the correlation of a predicate with failure but from different aspects. Since we assume identical inputs to all program executions, any predicate with high precision must be correlated with program failure. Furthermore, single-failure and exactly-one-causal-path assumptions imply that the predicates in the causal path should have high recall. Note that, precision alone is not a good metric since it will put high significance on rarely observed predicates in very few failed runs, which might be just coincidental. On the other hand, recall alone is not a good metric since any program invariant that happens regardless of program failure will have very high recall. Therefore, we restrict intervention on predicates that have both high precision and high recall and such predicates best discriminate the failed runs from the passed ones.

5 Modeling Causality

In this section, we discuss the *causality assumption model*. We start by discussing how we reason about *causal capability* relationships among predicates and proceed to describe the data structure that represents the potential causalities. We conclude with the discussion on *counterfactual causality* which we use to reason about actual causality in the intervention step during causal path discovery.

5.1 Causal Capability

A causal capability query is in the form: given two predicates P_1 and P_2, whether P_1 is capable to cause P_2. We need to first understand temporal precedence to reason about causal capability.

5.1.1 Temporal Precedence

For an event to be capable to cause another event, the cause must happen *temporally* before the effect. We use the term *temporal precedence* to express this basic prerequisite for causal capability. Note that, all causal capability relationships must obey the temporal precedence, but the opposite is not necessarily true. Specifically, temporal precedence is a necessary condition for causal capability, but not sufficient.

For program predicates, it is not always straightforward how to define temporal precedence. If P_1 ends before P_2 starts, we can be certain that P_1 temporally precedes P_2 and P_2 does not temporally precede P_1. But for any other case where the predicates overlap temporally, it remains uncertain. In general, reasoning about temporal precedence of predicates is a challenging task due to several reasons. First, we do not know the exact timestamps of the predicates, rather a time-window associated with each predicate. For example, when two methods are related in a parent-child relationship, i.e., `A()` calls `B()`, we can say (1) `B()` temporally precedes `A()` since `B()` finishes its job before `A()` does, or (2) `A()` temporally precedes `B()` since `A()` starts its job before `B()` does. Second, due to coarser granularity of instrumentation, we might observe two events occurring at the same time where they might happen a fractions of nano-seconds apart from each other. These two challenges led us to make some simplified assumption regarding temporal precedence.

Assumption 1 (Temporal Precedence). *We assume P_1 temporally precedes P_2 if P_1 started before P_2 in* all *executions where both of them were observed.*

We also note that, no definition of temporal precedence guarantees to capture all causal relationships. As an example, the predicate "a child method (callee) is running slow" might cause the predicate "parent (caller) method is running slow". The definition of temporal precedence that assigns precedence to the event that finishes first is well suited in this case. On the other hand, consider two events when the child method execution is delayed due to delayed execution of the parent method. In this case, the definition of temporal precedence that assigns precedence to the event that starts first is well suited.

5.1.2 Reasoning About Causal Capability

Given two predicates P_1 and P_2, we use the following five scenarios to reason about whether P_1 is capable to cause P_2. The conditions in the scenarios are much simpler than ideal, but we consider this orthogonal to our work and hence do not focus on fine tuning the conditions. Ideally, we assume an oracle that can answer the causal capability queries among predicates. In our work, A predicate P_1 is capable to cause another predicate P_2 if all conditions of any of the following scenarios holds:

1. **Causality involving simple sequential predicates**: We call a predicate *simple* if it does not involve events from multiple threads. In a program control-flow, within the same thread, a simple predicate is always capable to cause another predicate that follows it temporally. This is also true when the corresponding threads of the predicates are in a parent-child relationship. the conditions below formalize such a scenario to capture causal capability.

   - P_1 and P_2 do not involve multiple threads.
   - Both of them are from either (1) same thread or (2) different threads, but one of their threads stems from the other's thread.
   - P_1 temporally precedes P_2.

2. **Causality involving exception predicates**: A method does not execute further after throwing an exception. Hence, any other event within that method (e.g., running slow) is capable to cause the event related to that method throwing exception. the conditions below formalize such a scenario to capture causal capability.

   - P_1 and P_2 do not involve multiple threads.
   - P_1 and P_2 involve the same method instance, hence there is no temporal precedence among them.
   - P_2 is a predicate that encodes an exception throwing event.

3. **Causality involving compound-simple predicate pairs**: We call a predicate *compound* when it encodes interaction among events from multiple threads. A compound predicate is capable to cause another simple predicate, when a part of the compound predicate is capable to cause (based on the prior two scenarios) the simple predicate. The conditions below formalizes such a scenario to capture causal capability. To avoid any cycle, we impose strict conditions.

   - P_1 involves two events P_1^1 and P_1^2 from two different threads.
   - P_2 does not involve multiple threads.
   - P_1^1 or P_1^2 is capable to cause P_2.
   - P_2 is capable to cause neither P_1^1 nor P_1^2.

4. **Causality involving simple-compound predicate pairs**: A simple predicate is capable to cause another compound predicate, when the simple predicate is capable to cause a part of the compound predicate (based on the first two scenarios). the conditions below formalize such a scenario to capture causal capability. To avoid any cycle, we impose strict conditions.

   - P_1 does not involve multiple threads.
   - P_2 involves two events P_2^1 and P_2^2 from two different threads.
   - P_1 is capable to cause either P_2^1 or P_2^2.
   - Neither P_2^1 nor P_2^2 is capable to cause P_1.

5. **Causality involving compound-compound predicate pairs**: When both predicates are compound, one can cause the other when a part of the former can cause the latter and the latter can cause no part of the former. the conditions below formalize such a scenario to capture causal capability.

   - P_1 involves two events P_1^1 and P_1^2 from two different threads.
   - P_2 involves two events P_2^1 and P_2^2 from two different threads.
   - P_1 is capable to cause either P_2^1 or P_2^2.
   - P_2 is capable to cause neither P_1^1 nor P_1^2.

In all other cases, P_1 is not capable to cause P_2. If none of the two cases — (1) P_1 is capable to cause P_2 and (2) P_2 is capable to cause P_1 — are true, then we conclude that none of them is capable to cause each other. Note that, this is just a set of rules we follow in this work. In practice, one can design arbitrarily complex rule set to resolve the causal capability query. We only require that the definition of causal capability does not introduce any cyclic causal relationship involving multiple predicates.

5.2 Causal-Capability DAG, Causal DAG, and Causality Assumption

Causal-Capability DAG. Based on the causal capability rules, we organize the predicates using a directed acyclic graph (DAG) which we call the causal-capability DAG. We put a directed edge from P_1 to P_2 in the causal-capability DAG if P_1 is capable to cause P_2 based on the rules mentioned above. Note that, the causal-capability DAG contains both causal (P_1 causes P_2) and sequential relationships (P_1 precedes P_2 in the same thread, but does not cause).

Causal DAG. A causal DAG contains the actual causal relationships, i.e., for any edge (u, v) in the causal DAG, u must cause v. Informally, our causality assumption states that: "the causal-capability DAG contains the actual causal DAG within it". We provide it more formally below:

Assumption 2 (Causality Assumption). *If an edge (u, v) exists in the causal DAG, there must be a path from u to v in the causal-capability DAG.*

Our goal is to derive the causal DAG from the causal-capability DAG under the causality assumption. However, we are not interested in the entire causal DAG, but only the one that connects the root cause and the failure indicating symptom. Since we assume that there exists only one such path, the causal DAG is basically a chain of predicates.

5.3 Counterfactual Causality

There exists different definitions for causality in the existing in causality literature [12, 29]. While the actual notion of causality is hard to define and sometimes is a philosophical question, for practical purposes we often consider counterfactuals as causes. This is realistic in our case since we do not consider any conjunction or disjunction in the causal explanation path.

Assumption 3 (Counterfactual Causality). *We assume that the following two statements are equivalent: (1) if C had not occurred, E would not have occurred, and (2) C causes E.*

One might wonder that, from this assumption, running a program is a cause of its failure, because if the program was not run at the first place, no failure would have occurred. However, since we limit the predicates within the discriminating ones (with high precision and high recall), we get rid of such predicates that are program invariants.

6 Causal Path Discovery: An Intervention-based Approach

There are two steps in active statistical debugging. The first step is to build the causal-capability DAG from the discriminating predicates (discussed in Sections 4 and 5). The second step involves refining the causal-capability DAG through interventions and reveal the true causal DAG. In this section, we describe the second step. We first formalize the problem of *causal path discovery* under the problem settings and assumptions discussed in Section 3. After providing some background on group-testing, we proceed to elaborate on the intervention approach based on the group-testing paradigm, which we apply to solve the problem of causal path discovery. We conclude by providing directions towards possible extensions of active statistical debugging on some relaxed assumptions about the problem settings.

6.1 Problem Definition

Definition 6.1 (Causal path discovery)**.** Given a set of predicates $\mathcal{P}$, a failure indicating predicate F, and a causal-capability DAG $\mathcal{T} = (\mathcal{P}, E)$ that encodes the causal capability relationship among the predicates in $\mathcal{P}$, the problem of causal path discovery is to find a path $\langle C_0, C_1, \ldots, C_n \rangle$ such that (1) $C_i \in \mathcal{P}\ \forall\ 0 \leq i \leq n$, (2) $\forall\ 0 \leq i < n$ there is a path from C_i to C_{i+1} using edges in E, (3) C_0 is the root cause, (4) $C_n = F$, and (5) $\forall\ 0 \leq i < j \leq n$, C_i is a counterfactual cause of C_j.

Under the counterfactual causality assumption, a naïve approach to discover the causal path is to intervene on each discriminative predicate and learn its causal relationship with other predicates and the failure. However, this requires linear number of interventions in number of discriminative predicates. A better way is to use *group-testing*. We now proceed to discuss group-testing in general and then elaborate on our intervention scheme based on the group-testing paradigm.

6.2 Group-testing

Group-testing refers to the procedure that identifies certain objects (e.g., defective) among a set of objects while minimizing the number of *group-tests* required. Result of a group-test on a group of items is positive if at least one item in that group is defective, and negative otherwise. More formally, given a set $\mathcal{P}$ of N elements where D of them are defective, we perform M group-tests, each on group $P_i \subseteq \mathcal{P}$. Result of test on group P_i is positive if $\exists p \in P_i$ s.t. p is defective, and negative otherwise. The objective is to minimize M, i.e., the number of group-tests required.

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-test is performed after we observe the results of all $1 \leq j < i$ previous group-tests. In contrast, in the non-adaptive setting, each test is conducted independently. A trivial upper bound for adaptive group-testing [14] is $D \log N$. 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 $D < \frac{N}{\log N}$.

6.3 Intervention Preliminaries

Notations. We use F to denote the failure indicating predicate. We use the terms *symptom* or *failure symptom* interchangeably to refer to F. We use $\neg A$ to denote the event that predicate A is not observed. We use $\mathcal{P}$ to denote the predicate space, and $\mathcal{C}$ to denote the set of predicates that are in the target causal path connecting the root cause to F. We use R_{test} to denote the set of intervened runs. With slight abuse of notation, we use $A(r)$ to denote the fact that predicate A is observed in run r, and $\neg A(r)$ to denote that A is not observed in run r. We use $A \rightsquigarrow B$ to denote that there is a path from A to B in the causal-capability DAG. We summarize the notations in Figure 3.

Intervention terminologies. We use two terms *treatment* and *control* to denote whether intervention is performed on a predicate or not. In our case, a treatment is an intervention that *deactivates* a predicate. By deactivation, we mean that we make sure that the predicate is behaving the way it does in the passed runs. As an example, if a method `foo()` returns `True` in all passed runs and `False` in all failed runs, treating the predicate "`foo()` returns `False`" would be to force `foo()` to return `True`. We use the word deactivation because we are deactivating the potential cause of program failure through the treatment.

Notation	Description
$\mathcal{P}$	Set of predicates
F	Failure indicating symptom predicate
$\mathcal{T} = (\mathcal{P}, E)$	Causal-capability DAG
$\mathcal{C} = \langle C_0, C_1, \ldots, C_n \rangle$	Causal path
C	A potential cause under intervention
$\mathcal{X}$	Set of spurious predicates
X	A predicate to be discarded via pruning
$\mathcal{B}$	Set of mega predicates representing branches
R_{test}	Set of intervened runs
$A(r)$	Predicate A is observed in run r
$\neg A(r)$	Predicate A is not observed in run r
$A \rightsquigarrow B$	There is a path from A to B in $\mathcal{T}$
$\mathcal{N}$ or N	Total number of predicates (or objects)
M	Number of group-interventions (or group-tests)
D	Number of faulty predicates (or defective objects)
J	Number of junctions in $\mathcal{T}$
B	Maximum number of branches at any junction
T_j	Number of branches at j-th junction
N_t^j	Number of predicates in the t-th branch at j-th junction
d_j	Number of faulty predicates between junction j and $j+1$
$\mathbf{d} = [d_1, \ldots, d_J]$	Vector containing all d_js
k	Information theoretic lower bound for the group-testing problem
S	Number of discarded predicates during each faulty predicate discovery
N_C	Maximum number of predicates in any path in $\mathcal{T}$

Figure 3: Summary of notations

6.4 Predicate Pruning

The classical group-testing approaches do not assume any dependency among objects. Specifically, when a group-test is performed on a group P_i, no additional information is obtained or used about the other groups $P_j, j \neq i$. In our case, when a particular predicate is deactivated, we will often observe few other causally related predicates to become deactivated as an effect. We can leverage this additional information while using an adaptive group-testing strategy in our problem settings.

We use the causal-capability DAG and this additional observation during intervention to prune predicates aggressively. We provide two cases where we can prune the predicate space beyond adaptive group-testing approach below. We focus on intervening on a single predicate while describing the pruning rules. Later we discuss how these rules can be extended for group-intervention.

- Case 1: We intervene C where $C \rightsquigarrow F$ and observe $\nexists r \in R_{test} \ \neg C(r) \wedge F(r)$. We conclude that C causes F by the counterfactual assumption, i.e., whenever C does not happen, F does not happen. We update $\mathcal{C} = \mathcal{C} \cup \{C\}$.

  (Pruning). Suppose that $\mathcal{X} = \{X \in \mathcal{P} : X \not\rightsquigarrow C \wedge (\exists r \in R_{test} \ \neg C(r) \wedge \neg F(r) \wedge X(r))\}$. We update $\mathcal{P} = \mathcal{P} - \mathcal{X}$. Informally, we find out all predicates X such that X is not capable to cause C and X is observed, although neither C nor F is observed. This ensures that X is neither caused by C nor causes F. Note that, we need the additional constraint $X \not\rightsquigarrow C$ to ensure that X is not the root cause that causes C which in turn causes F. Because, in that case, treating C will not result in deactivation of X as X is independent of C.

- Case 2: We intervene C and observe $\exists r \in R_{test} \ \neg C(r) \wedge F(r)$. We conclude that C does not cause F. We update $\mathcal{P} = \mathcal{P} - \{C\}$.

  (Pruning). Now suppose that $\mathcal{X} = \{X \in \mathcal{P} : C \rightsquigarrow X \rightsquigarrow F \wedge (\nexists r \in R_{test} \ \neg C(r) \wedge X(r))\}$. This means that predicates in $\mathcal{X}$ are caused by C, and has nothing to do with F. Hence, we update $\mathcal{P} = \mathcal{P} - \mathcal{X}$.

Extending pruning for group-intervention. The aforementioned pruning rules extend when C is a set of predicates. By the counterfactual assumption: when we deactivate all predicates within a set of predicates $\mathcal{A}$ and observe all predicates in another set of predicates $\mathcal{B}$ to be deactivated as an effect, we can conclude that $\mathcal{A}$ is a counterfactual cause of $\mathcal{B}$.

Algorithm 1: Causal-Path-Discovery $(\mathcal{T}, F)$

Input : Causal-capability DAG, $\mathcal{T}$
Failure indicating predicate, F
Output : A causal path $\mathcal{C} = \langle C_0, C_1, \ldots, C_n \rangle$

```
begin
    T_chain = Branch-Prune(T, F)
    C, X = Group-Intervention-With-Pruning(V_{T_chain} − {F}, T_chain, F, [0, |V_{T_chain}| − 1])
    C = Vertex-Induced-Subgraph(C, T_chain)
return C
```

In general, whenever we observe failure despite deactivating some predicates, we prune them and all other predicates that are deactivated as an effect to the deactivation of the predicates under treatment. The only difference is when we add a predicate to $\mathcal{C}$. We only add a predicate group to $\mathcal{C}$ when it contains a single predicate. This is because when a group of predicates turn out to be a counterfactual cause of failure, we only know that at least one of the predicates within that group is the counterfactual cause, but nothing beyond that. Therefore, we perform repeated group-intervention on that group in a divide and conquer fashion until we narrow down to one single predicate.

6.5 Intervention Algorithm

Now we describe the intervention algorithm. We intervene in a top-down manner since this gives us more chance to prune spurious predicates based on the two pruning rules. Specifically, we follow the topological order of predicates in the causal-capability DAG to decide on which predicates to perform group-intervention first. We execute the intervention algorithm in a two stage approach — branch pruning (Algorithm 2) and group intervention with pruning (Algorithm 3). During the branch pruning stage, we do not intervene as long as we are following a single path while traversing the causal-capability DAG from top to bottom. We perform intervention only when we hit a *junction* with multiple possible branches downwards. We provide our intervention algorithm in Algorithm 1.

Branch pruning. Since we assume that there is exactly one causal path, at each junction, we use adaptive group-testing approach to select the correct branch. To do so, we group all predicates within a branch and find the correct branch in a divide and conquer approach. In this case, we perform *branch-group-intervention*, where we intervene all predicates within a set of branches.

Based on the intervention outcome, we update the causal-capability DAG and remove any predicate that is no longer reachable from the correct branch (Figure 4). After picking the correct branch (or none if that is the case), we treat all other predicates in the irrelevant branches in one single intervention and prune the predicates downstream by applying the second pruning rule (line 27 of Algorithm 3).

Group intervention with pruning. Once we are left with a single chain, we use adaptive group-testing approach to discard the irrelevant predicates. We pick predicates in the topological order so that the chance to prune more predicates down in the chain is maximized. We expect this to perform more efficiently due to the two pruning rules which traditional adaptive group-testing approaches cannot leverage on. During each intervention round, we pick half of the remaining predicates that are at the highest topological level, and intervene on them. We keep applying the two pruning rules during each intervention step until we either (1) obtain the maximum number of causally related predicates or (2) all predicates are marked as either causal or pruned. If the former happens, we discard all other predicates since none of them can be causal predicates.

6.6 Extensions

We now discuss how our algorithm needs modification to accommodate few possible extensions that relax/extend some of our assumptions mentioned earlier in Section 3.

Probability distribution. If we can incorporate the causal probability distribution, we can apply that to optimally pick branches and predicates during group intervention. In that case, our goal would be to pick a set of predicates during each intervention that maximizes the expected number of predicates that will be pruned. In this work, we consider the simplest case where we quantify the effect of each intervention on other predicates with boolean states (deactivated or not). This can be seen as if we assign weights to the edges of the causal-capability DAG to either 0 (not causal) or 1 (certainly causal). While this works well under the conservative assumptions that we mentioned

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

Input : Causal-capability DAG, $\mathcal{T}$
Failure indicating predicate, F
Output : A chain of predicates $\mathcal{T}_{chain}$

begin
  $\mathcal{T}_{init} = \mathcal{T}$
  $potentialCauses = \emptyset$
  $\mathcal{P} = \{v \in V_{\mathcal{T}} : \nexists u \in V_{\mathcal{T}} \ \text{Level}(u) < \text{Level}(v)\}$ `// Level: topological level`
  while $|\mathcal{P}| > 0$ **do**
    if $|\mathcal{P}| == 1$ **then**
      $\mathcal{C} = \mathcal{P}$
    else
      $\mathcal{B} \leftarrow \emptyset$
      foreach $p \in \mathcal{P}$ **do**
        $desc_p$ =UniqueDescendants$(p, \mathcal{T})$ `//` $desc_p \cap desc_q = \emptyset \ \forall p, q \in \mathcal{P}, p \neq q$
        $\mathcal{B} = \mathcal{B} \cup \{\text{Mega-Predicate}(desc_p)\ \}$
      $\mathcal{C}, \mathcal{X} =$ **Group-Intervention-With-Pruning**$(\mathcal{B}, \mathcal{T}, F, [0, 1])$ `// at most one branch can be causal`
      $\mathcal{C} =$ Extract-Predicates $(\mathcal{C})$
      $\mathcal{X} =$ Extract-Predicates $(\mathcal{X})$
      if $\mathcal{C} \neq \emptyset$ **then** `// one causal branch is found in` $\mathcal{B}$
        $\mathcal{X} = \mathcal{X} \cup \{u \in V_{\mathcal{T}} : \nexists v \in \mathcal{C} \ \ v \rightsquigarrow u \vee u \rightsquigarrow v\}$ `// unreachable predicates`
      else `// none of the branches in` $\mathcal{B}$ `is causal`
        $\mathcal{X} = \mathcal{X} \cup$ Extract-Predicates $(\mathcal{B})$
      Remove$(\mathcal{X}, \mathcal{T})$ `// remove predicates in` $\mathcal{X}$ `and their incident edges from` $\mathcal{T}$
    $potentialCauses = potentialCauses \cup \mathcal{C}$
    Remove$(\mathcal{C}, \mathcal{T})$ `// remove predicates in` $\mathcal{C}$ `and their incident edges from` $\mathcal{T}$
    $\mathcal{P} = \{v \in V_{\mathcal{T}} : \nexists u \in V_{\mathcal{T}} \ \text{Level}(u) < \text{Level}(v)\}$
  $\mathcal{T}_{chain} =$ Vertex-Induced-Subgraph$(potentialCauses, \mathcal{T}_{init})$ `//` $\mathcal{T}_{chain}$ `must be a chain`
  return $\mathcal{T}_{chain}$

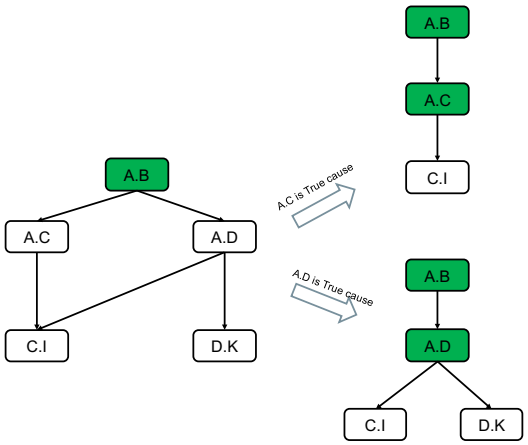

Figure 4: Updating causal-capability DAG after determining the correct branch. We remove any predicate that is no longer reachable from the correct branch.

in Section 3, in practice, many of those assumptions do not hold. In such cases, we can update the edge weights using real values in [0, 1] to signify the causal probability. More specifically, when we intervene on certain predicates, we might observe a deviation in distribution of other predicate states (not necessarily the two extreme cases of full deactivation or no deactivation). Any significant deviation implies causal connection between the predicates under intervention and the predicates that deviates. We can use this deviation to update the edge weights which signify causal probabilities. Hence, after each round of intervention, we will update the causal DAG and compute the next set of interventions such a way that maximizes the expected number pruning of spurious predicates.

Conjunctive root causes. If we assume that root causes can be conjunctive, i.e., multiple predicates, in conjunction, trigger failure, then we will not be able to apply pruning rule 1.

Algorithm 3: Group-Intervention-With-Pruning $(\mathcal{P}, \mathcal{T}, F, D)$

Input : A set of candidate predicates, $\mathcal{P}$
Causal-capability DAG, $\mathcal{T}$
Failure indicating predicate, F
Lower and upper limit of number of causes to find, $D = [D_{low}, D_{high}]$
Output : A set of predicates that are counterfactual causes of F, $\mathcal{C}$
A set of spurious predicates that are not causally related to F, $\mathcal{X}$

begin
  $\mathcal{C} = \emptyset$
  $\mathcal{X} = \emptyset$
  while $\mathcal{P} \neq \emptyset \wedge |\mathcal{C}| \leq D_{high}$ **do**
    if $D_{low} = 0$ **then** `// there might be no predicate in` $\mathcal{P}$ `that is causal`
      $R_{test} = \text{Intervene}(\mathcal{P})$ `// deactivate all` $p \in \mathcal{P}$
      if $\exists r \in R_{test}\ F(r) = True$ **then** `// failure did not stop`
        $\mathcal{U} = \{u \in \bigcup_{p \in \mathcal{P}} \text{Descendants}(p, \mathcal{T})\text{: } \forall r \in R_{test}\ u(r) = False\}$
        $\mathcal{X} = \mathcal{X} \cup \mathcal{P} \cup \mathcal{U}$ `// pruning rule 2`
        break
    Pick $P_i \subseteq \mathcal{P}$ s.t. $|P_i| = \lceil \frac{|\mathcal{P}|}{2} \rceil \wedge \nexists u, v : u \in P_i \wedge v \in (\mathcal{P} - P_i) \wedge Level(v) < Level(u)$
    $R_{test} = \text{Intervene}(P_i)$ `// deactivate all` $p \in P_i$
    if $\forall r \in R_{test}\ F(r) = False$ **then** `// failure stopped`
      if $|P_i| == 1$ **then**
        $\mathcal{C} = \mathcal{C} \cup P_i$
      else
        $\mathcal{C}', \mathcal{X}' =$ **Group-Intervention-With-Pruning**$(P_i, \mathcal{T}, F, [1, \min(D_{high}, |P_i|)])$
        $\mathcal{C} = \mathcal{C} \cup \mathcal{C}'$
        $\mathcal{X} = \mathcal{X} \cup \mathcal{X}'$
      $\mathcal{U} = \{u \in \mathcal{P} : \nexists\ p \in P_i\ p \in \text{Descendants}(u, \mathcal{T}) \wedge \exists r \in R_{test}\ u(r) = True\}$
      $\mathcal{X} = \mathcal{X} \cup \mathcal{U}$ `// pruning rule 1`
    else `// failure did not stop`
      $\mathcal{X} = \mathcal{X} \cup P_i$ `// no predicate in` P_i `is a counterfactual cause of` F
      $\mathcal{U} = \{u \in \bigcup_{p \in P_i} \text{Descendants}(p, \mathcal{T})\text{: } \forall r \in R_{test}\ u(r) = False\}$
      $\mathcal{X} = \mathcal{X} \cup \mathcal{U}$ `// pruning rule 2`
    $\mathcal{P} = \mathcal{P} - (\mathcal{C} \cup \mathcal{X})$
  if $\mathcal{P} \neq \emptyset$ **then** `// we found all causal predicates, the rest are spurious`
    $\mathcal{X} = \mathcal{X} \cup (\mathcal{P} - \mathcal{C})$
  return $\mathcal{C}$, $\mathcal{X}$

Disjunctive root causes. If we assume that there could be multiple disjunctive causal paths, i.e., many causal paths are *sufficient* to trigger failure, then we will not be able to apply pruning rule 2. Moreover, we will rarely observe a situation where pruning rule 1 can be applied. However, we can still make some deduction in this case by observing the reduction of failure rate. As an example, suppose that there exists 3 disjunctive causal paths, each equally likely to cause failure, and we observe failure in 60% executions. Deactivating one of these paths will reduce failure rate to 40%.

Lack of intervenability. One practical challenge arises when a predicate is not intervenable. In this case, we rely on other predicates to guide us in pruning spurious non-intervenable predicates. However, there is no concrete way to guarantee whether such non-intervenable predicates are causally related to the failure. Hence, we give them a benefit of doubt and do not prune them unless we are certain. If the root cause is non-intervenable or not captured by any predicate, active statistical debugging still works by producing a causal path that is close to the actual one. In general, our intervention algorithm might report false positive predicates in the causal path, but the case of false negative never happens.

7 Algorithm Complexity Analysis

In this section, we first prove that the information theoretic lower bound is reduced under the presence of pruning. Then we compare the upper bound of number of required interventions under branch pruning with that of the classical group-testing without branch pruning. We also prove that

the upper bound of the number of required interventions is reduced assuming a lower bound on the number of predicates pruned during each intervention round. For the upper bound analysis, we use the naïve approach as the underlying adaptive group-testing mechanism, that requires at most $D \log N$ group-tests to retrieve D defective items from N items. While we could use any other adaptive group-testing algorithm, such as Hwang's Generalized Binary Splitting Algorithm [14], we chose the naïve approach for the sake of keeping the analysis simple. Specifically, we want to show that the upper bound of any group-testing algorithm is reduced when pruning mechanism is available. Recall that, we assume $D < \frac{N}{\log N}$, since otherwise, a linear approach that intervenes only one predicate at a time would be preferable.

7.1 Information Theoretic Lower Bound

In the classical group-testing problem, we are given $\mathcal{N}$ objects, and out of them D are defective. Since there are $\binom{\mathcal{N}}{D}$ possible outcomes for such a case, the information theoretic lower bound of required computation by any group-testing algorithm is $\log_2 \binom{\mathcal{N}}{D}$. Our aim is to show that the required amount of information is reduced in our case. We can gain such reduction due to two properties: (1) *exactly-one-causal-path* assumption, and (2) structure of the causal-capability DAG and causal dependency among predicates. We now analyze how we can get a lower information theoretic lower bound exploiting these properties.

7.1.1 Exactly-one-causal-path Assumption

Due to concurrency in the programs, the temporal precedence DAG obtained from the run-time behavior of the programs can be viewed as a collection of *junctions*. Junctions are start or sync point of multiple threads. The *exactly-one-causal-path* assumption states that, when there are multiple branches at any junction, at most one of them can contain faulty predicates. Because, otherwise, we will end up constructing a causal *DAG*, instead of a causal *path*.

Suppose that there are J junctions in the causal-capability DAG, there are T_j branches at the j-th junction, and the t-th branch contains N_t^j predicates. Without loss of generality, we can consider linear parts (there is no branching) of the causal-capability DAG as if there is only one branch there. Also assume that there are D faulty predicates which are causally related to the failure. We use $\mathcal{N}$ to denote the total number of predicates in the causal-capability DAG.

We use $\mathbf{d}$, a vector of size J, where d_j is the number of faulty predicates between junction j and $j+1$ (or the end point containing only the failure indicating predicate), for $1 \leq j \leq J$. Also note that, $\sum_{j=1}^{J} d_j = D$. This leaves us limited number of ways we can choose the value of $\mathbf{d}$. We use $Sol(D, J)$ to denote the set of vectors $\mathbf{d}$ of size J such that $\sum_{j=1}^{J} d_j = D$. Note that, $|Sol(D, J)| = \binom{D+J-1}{D}$ since there are $\binom{D+J-1}{D}$ possible ways to assign the values of d_j for $1 \leq j \leq J$. Now we can compute the number of possible ways to pick D faulty predicates under the *exactly-one-causal-path* assumption W_A below:

$$W_A = \sum_{\mathbf{d} \in Sol(D,J)} \prod_{j=1}^{J} \sum_{t=1}^{T_j} \binom{N_t^j}{d_j}$$

To contrast the above quantity with the case where the *exactly-one-causal-path* assumption does not hold, we compute the number of possible outcomes W_G. We partition the set $Sol(D, J)$ into two disjoint subsets. We use the notation $Sol'(D, J) \subseteq Sol(D, J)$ to denote the set of vectors where exactly one element is non-zero. We use the notation $\overline{Sol'}(D, J) \subseteq Sol(D, J)$ to denote the set containing all other elements. i.e., $\overline{Sol'}(D, J) = Sol(D, J) - Sol'(D, J)$.

Example: $Sol'(D, J) = \{\begin{bmatrix} D & 0 & 0 & \cdots & 0 \end{bmatrix}, \begin{bmatrix} 0 & D & 0 & \cdots & 0 \end{bmatrix}, \cdots, \begin{bmatrix} 0 & 0 & 0 & \cdots & D \end{bmatrix}\}$.

Theorem 1. *Under exactly-one-causal-path assumption, the information theoretic lower bound of adaptive group-testing is reduced by* $\sum_{\mathbf{d} \in Sol(D,J)} \prod_{j=1}^{J} \sum_{\mathbf{d^j} \in \overline{Sol'}(d_j, T_j)} \prod_{t=1}^{T_j} \binom{N_t^j}{d_t^j}$.

Proof. We compute the number of possible outcomes without exactly-one-causal-path assumption W_G below:

$$W_G = \binom{\mathcal{N}}{D} = \binom{\sum_{j=1}^{J} \sum_{t=1}^{T_j} N_t^j}{D} = \sum_{\mathbf{d} \in Sol(D,J)} \prod_{j=1}^{J} \binom{\sum_{t=1}^{T_j} N_t^j}{d_j}$$

$$
\begin{aligned}
&= \sum_{\mathbf{d}\in Sol(D,J)} \prod_{j=1}^{J} \sum_{\mathbf{d^j}\in Sol(d_j,T_j)} \prod_{t=1}^{T_j} \binom{N_t^j}{d_t^j} \\
&= \sum_{\mathbf{d}\in Sol(D,J)} \prod_{j=1}^{J} \Big(\sum_{\mathbf{d^j}\in Sol'(d_j,T_j)} \prod_{t=1}^{T_j} \binom{N_t^j}{d_t^j} + \sum_{\mathbf{d^j}\in \overline{Sol'}(d_j,T_j)} \prod_{t=1}^{T_j} \binom{N_t^j}{d_t^j} \Big) \\
&= \sum_{\mathbf{d}\in Sol(D,J)} \prod_{j=1}^{J} \Big(\sum_{t=1}^{T_j} \binom{N_t^j}{d_j} + \sum_{\mathbf{d^j}\in \overline{Sol'}(d_j,T_j)} \prod_{t=1}^{T_j} \binom{N_t^j}{d_t^j} \Big) \\
&= W_A + \underbrace{\sum_{\mathbf{d}\in Sol(D,J)} \prod_{j=1}^{J} \sum_{\mathbf{d^j}\in \overline{Sol'}(d_j,T_j)} \prod_{t=1}^{T_j} \binom{N_t^j}{d_t^j}}_{Reduction}
\end{aligned}
$$

□

As shown above, we get a much lower information theoretic lower bound with the exactly-one-causal-path assumption. To get a sense of how much reduction we are achieving, suppose that there are J junctions, T branches at each junction, and N predicates at each branch. Although there are way too many different possible ways to pick the values of d_t^j, when $D \geq TJ$, one such possibility is $d_t^j = 1 \ \ \forall 1 \leq t \leq T$ and $\forall 1 \leq j \leq J$. For just this case, the reduction is $(N^T)^J$. The key take-away from this is that when we have exactly-one-causal-path assumption, the number of possible ways to pick faulty predicates at each branch becomes *additive* to the other branches at the same junction, where it is *multiplicative* when the assumption does not hold.

7.1.2 Causal Dependency

Now we move on to analyze the information theoretic lower bound under the presence of two pruning rules along with the strategy of picking predicates from lowest topological level first. Suppose that we are given N objects and D of them are defective. In the group-testing approach without any pruning, after each query, we get 1 bit of information. Now suppose that a minimum number of k queries are required, i.e., k is the information theoretic lower bound. After retrieving all information, the remaining information should be ≤ 0. Hence,

$$
\begin{aligned}
&\log_2 \binom{N}{D} - \sum_{i=1}^{k} 1 \leq 0 \\
\implies &\log_2 \binom{N}{D} - k \leq 0 \\
\implies &k \geq \log_2 \binom{N}{D}
\end{aligned}
$$

Theorem 2. *The information theoretic lower bound for adaptive group-testing is* $\frac{\log_2 \binom{N}{D}}{1+\frac{DS}{N}}$ *when* S *predicates are discarded using the two pruning rules during each group-intervention.*

Proof. After the first intervention, we get $\Big(\log_2 \binom{N}{D} - \log_2 \binom{N-S}{D} + 1 \Big)$ bits of information. Since after retrieving all information, the remaining information should be ≤ 0:

$$
\begin{aligned}
&\log_2 \binom{N}{D} - \sum_{i=1}^{k} \Big(\log_2 \binom{N-(i-1)S}{D} - \log_2 \binom{N-iS}{D} + 1 \Big) \leq 0 \\
\implies &\log_2 \binom{N-kS}{D} - k \leq 0 \\
\implies &k \geq \log_2 \frac{(N-kS)!}{D!(N-ks-D)!} \\
\implies &k \geq \log_2 \frac{(N-kS)^D}{D!} \quad [\frac{(N-kS)!}{(N-ks-D)!} \approx (N-kS)^D] \\
\implies &k \geq D \log_2(N-kS) - \log_2(D!)
\end{aligned}
$$

$$\implies k \geq D \log_2 N(1 - \frac{kS}{N}) - \log_2(D!)$$
$$\implies k \geq D \log_2 N + D \log_2(1 - \frac{kS}{N}) - \log_2(D!)$$
$$\implies k \geq D \log_2 N - \frac{kDS}{N} - \log_2(D!) \qquad [\log(1-x) \approx -x \text{ for small } x\text{; we assume } \frac{kS}{N} \text{ to be small}]$$
$$\implies k\Big(1 + \frac{DS}{N}\Big) \geq D \log_2 N - \log_2(D!)$$
$$\implies k\Big(1 + \frac{DS}{N}\Big) \geq \log_2 \frac{N^D}{D!}$$
$$\implies k\Big(1 + \frac{DS}{N}\Big) \geq \log_2 \frac{N!}{D!(N-D)!} \qquad [N^D \approx \frac{N!}{(N-D)!}]$$
$$\implies k \geq \frac{\log_2 \binom{N}{D}}{1 + \frac{DS}{N}}$$

□

Since $\frac{DS}{N} > 0$, we obtain a reduced lower bound of number of required interventions. In general, the larger value S has, the lower the information theoretic lower bound will be.

7.2 Upper Bound of Number of Interventions

Similar to the lower bound analysis, we analyze the upper bound of number of interventions under exactly-one-path-assumption (using branch pruning), and causal dependency.

7.2.1 Exactly-one-causal-path Assumption

Theorem 3. *Under exactly-one-causal-path assumption, with J junctions in the causal-capability DAG, at most D predicates in the causal path, and $J < D$, the upper bound of the number of interventions required is reduced when branch pruning is applied.*

Proof. The number of interventions required during the branch pruning process is bounded above by the logarithm of the number of branches. Since we assume that at most one branch is the true causal branch at each junction, we can find that using at most $\log B$ interventions at each junction, where B is the number of branches at that junction. Also, B can be at most 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 runs within the same thread. Note that, although the total number of possible paths can be exponential in the number of branches, the number of interventions required is logarithmic in the number of branches at each junction. So if there are J junctions and there are at most T branches at each junction, the number of interventions required is at most $J \log T$. Now let us assume that the maximum number of predicates in any path in the causal-capability DAG is N_C. Therefore, the chain found after branch pruning can contain at most N_C predicates, and out of them, D are causally related to the failure. We will require at most $D \log N_C$ interventions to find out the true causes. Therefore, the total number of required interventions is at most $J \log T + D \log N_C$. In contrast, a group-testing approach without branch pruning would require at most $D \log(TN_C) = D \log T + D \log N_C$ interventions. Therefore, whenever $J < D$, upper bound of the number of interventions required with branch pruning will be lower. □

7.2.2 Causal Dependency

We now focus on the upper bound of required interventions. As an extreme case, if every edge in the causal-capability DAG is causal, we would not need any intervention. On the other hand, in worst case, any pruning might not happen. In that case, the number of required interventions would be the same as that of the adaptive group-testing without pruning ($\approx D \log N$, assuming naïve group-testing approach).

Theorem 4. *If at least S predicates are pruned during discovery of each faulty predicate, then the upper bound of number of interventions required to isolate D faulty predicates among N total predicates using the naïve group testing approach is $D \log N - \frac{D(D-1)S}{2N}$.*

Proof. Since at least S predicates will be pruned during discovery of one faulty predicate, and there are D of them, at least DS predicates will be pruned in total. Now we compute the upper bound of the number of required interventions using the naïve approach below:

$$
\begin{aligned}
&\sum_{i=1}^{D} \log\Big(N - (i-1)S\Big) \\
=&\sum_{i=1}^{D} \log\Big(N\big(1 - \frac{(i-1)S}{N}\big)\Big) \\
=&\sum_{i=1}^{D} \log N + \sum_{i=1}^{D} \log\Big(1 - \frac{(i-1)S}{N}\Big) \\
\approx&\sum_{i=1}^{D} \log N - \sum_{i=1}^{D} \frac{(i-1)S}{N} \\
=&D \log N - \frac{D(D-1)S}{2N}
\end{aligned}
$$

□

Hence, the reduction depends on S. When $S = 1$, we are referring to the group-testing approach in absence of pruning, because once we fine a faulty predicate, we exclude that predicate from any further intervention.

8 Experimental Evaluation

In this section, we present the empirical result of applying active statistical debugging towards root cause explanation of synthetically generated concurrent programs. We first provide brief discussion of the program instrumentation towards generating the execution logs and then proceed towards discussing the dataset, i.e., how we generate the faulty programs. Then we discuss the baseline approaches and finally present our findings.

8.1 Program Instrumentation

We instrument the programs to generate program execution logs in method-level granularity. For each instance of a method call, we log run-time features associated with that instance, such as start-time, duration, thread that invokes the method instance, data type of the object the method is a member of and the particular object identifier, return value, thrown exception (if any), etc. We use this execution log to extract predicates from each execution. Additionally, each execution is labelled as whether it passed or failed.

Since our instrumentation is of much sparser granularity than existing work [24, 16] that employ sampling based finer granularity instrumentation, we do not use any sampling. Another advantage of our instrumentation is, we can design the predicates after the execution log collection. In contrast, prior work instrument the programs to directly extract the predicates. As an example, to assess whether two methods return the same value, prior work would need to instrument the program using a hard coded conditional statement "`pred = (foo() == bar())`". In contrast, our instrumentation framework simply collects the return values of the two methods and stores them in the execution log. Therefore, we have the flexibility to design the predicates post-execution, often based on knowledge of some domain-expert. For example, in this case, we can design multiple predicates such as whether two values are equal, unequal, or satisfy any custom relation.

8.2 Dataset

We used a template to automatically generate 20 different programs in C#. The generated programs had variable number of threads ranging from 3 to 20. In all programs, the length of the causal path was 5, where three of them were exception related predicates. The total number of predicates ranged from 77 to 243 and most of these predicates were intervenable.

The root cause of failure of all of these programs was a specific value of a variable which was randomly generated within the program. However, this root cause was not captured due to limitation in predicate design and coarse instrumentation granularity. Note that, this program template design

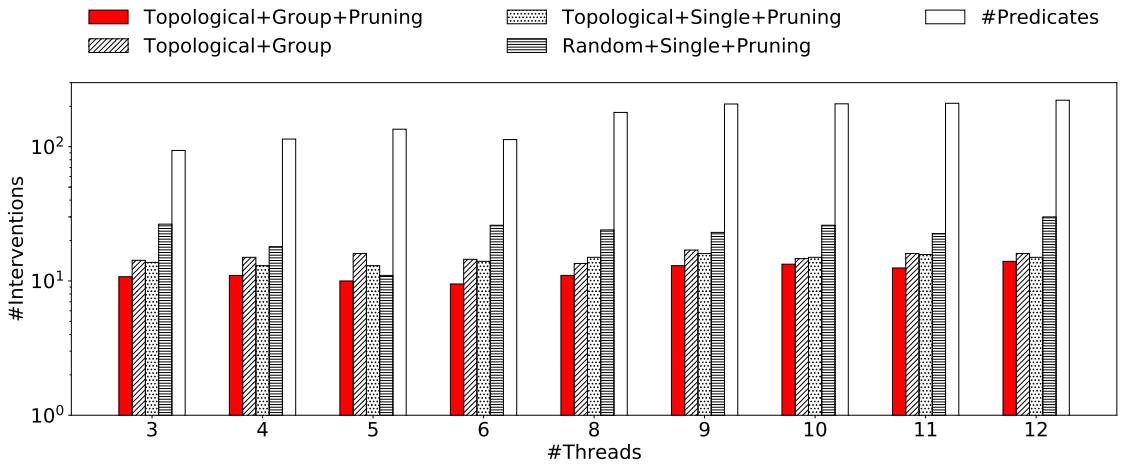

Figure 5: Comparison based on number of interventions required among four strategies of program intervention. Each bar group represent aggregated result over programs having same number of threads. The number of total predicates is shown at the rightmost bar within each bar group. Without group-testing and pruning, one would need to perform as many interventions as the number of total predicates to identify the true causal path.

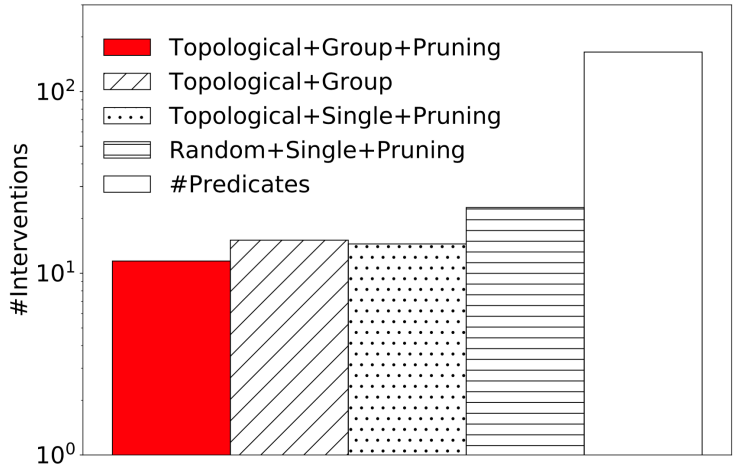

Figure 6: Average number of interventions across all test programs where active statistical debugging outperforms all baselines.

was intentional, because this type of situation is very common during root cause analysis. In such cases, often the root cause invokes a number of events that are causally unrelated to the failure. Without any clever approach, the task of separating the causally related events from spurious events becomes as difficult as finding needle in a haystack. However, our active statistical debugging framework efficiently solves this challenging task.

We designed the template such that the root cause triggers a number of spurious method calls in different threads, including threads containing events that are causally related to failure. Some of these spurious method calls are causally related to each other, while some are not. In our case, the next causally related event was a particular method running slow, which triggered an order violation involving a shared object[3]. These two events were captured by our predicate design and they in turn caused three causally connected exceptions including the failure indicating exception. Our goal is to build the causal chain connecting the two captured causes with the failure indicating exceptions, and filter out all other correlated but spurious events.

[3]We call this order violation incident `WriteAfterDispose`, where a thread attempts to write to a `TextWriter` object which had already been `disposed` by some other thread.

8.3 Baseline Approaches

In Figures 5 and 6, we use "Topological+Group+Pruning" to refer to our active statistical debugging approach. This signifies that active statistical debugging (1) intervenes predicates following the *topological* order of the causal-capability DAG, (2) applies *group* intervention as opposed to single intervention, and (3) applies *pruning* rules to discard predicates. We empirically contrast active statistical debugging with four other baselines that are also based on program intervention and discuss them below:

1. **Naïve Approach:** The simplest intervention approach to find the causal path is to intervene each predicate and observe its causal connection to other predicates and program failure using counterfactual causality. Since we need to inspect each predicate through intervention, the number of required interventions in such an approach would be equal to the total number of predicates. Therefore, we do not implement such inefficient approach, but provide the total number of predicates so that one can compare our approach with the naïve approach.

2. **Topological+Group:** This approach intervenes the predicate as groups, but does not apply any pruning scheme. Note that, since this approach does not prune any predicate, it does not matter whether it picks predicates in topological order or not. Hence, this approach can be thought of as the classical group-testing approach.

3. **Topological+Single+Pruning:** This approach intervenes one predicate at a time, but picks the predicates following the topological order of the causal-capability DAG. Moreover, it also applies the pruning mechanisms. This is an useful setting because often intervening predicates interfere with each other. Hence, in such cases, the only way to apply program intervention technique is to intervene one predicate at a time.

4. **Random+Single+Pruning:** This approach is similar to the previous approach, except it picks the predicates in a random order. This approach serves as a baseline to prove our intuitive hypothesis: following the topological order while picking predicates for intervention results in pruning more predicates.

8.4 Empirical Findings

We now procede to describe our empirical findings. Figures 5 and 6 summarizes our empirical findings. All of the implemented strategies resulted in correct causal path discovery, which makes all of the approaches equally effective. We focus on contrasting the efficiency of different approaches and we measure efficiency by the number of interventions required. We provide three key-observations that we learned from the experiments below:

1. **Intervening predicates by topological order is beneficial under pruning:** This is an intuitive hypothesis and the results supported the hypothesis. Since no pruning can be done to antecedent predicates, this is natural that picking predicates by topological order would favor pruning. The more we can prune early on, the fewer number of interventions we will require later.

2. **Group-testing is better than single-testing under short causal path assumption:** With all other settings kept same, switching from single to group-intervention requires fewer number of interventions when the number of faulty predicates are small. In our case, only 2 predicates were faulty and hence group-testing approach outperforms single-testing approach.

3. **Predicate pruning is beneficial:** The key contribution of our approach that differentiates it from the classical group-testing approach is presence of predicate pruning strategies. It is also evident from our experiments that applying pruning significantly reduces the number of required interventions. This reduction is more prominent in case of single-test. Due to interference among multiple interventions, single-test strategy might be often preferable over group-testing. In such cases, through predicate pruning, we can significantly reduce the number of required interventions compared to the naïve approach.

The key take-away from the empirical results is that the relationships among predicates serve as a significant source of information which pruning strategies exploit. The most significant amount of reduction in the number of required interventions comes from the application of pruning; and this is most prominent in the case of single-testing.

8.5 A Note on Evaluation Requirement

Due to the intervention design requirement of program predicates, active statistical debugging has limited applicability to the current benchmarks of buggy programs. Writing source codes in an intervenable framework is one way to support active statistical debugging, but this does not apply to past codes. The other way is to convert existing codes to include and enable intervention points. This requires significant engineering effort which is orthogonal to our contribution. Hence, our evaluation is limited to synthetically generated programs. However, we are currently working on bridging the gap between what technology is available at this point and how active statistical debugging can be adopted there.

9 Related Work

Group-testing. Group-testing has been applied for fault diagnosis in prior literature [35]. Particularly, adaptive group-testing is related to our work since our intervention algorithm uses it as its skeleton. While group-testing is a well-researched area in theoretical computer science [1, 6, 14, 7], none of the existing work considers the scenario where a group-test might reveal additional information. The *probabilistic* variation of adaptive group-testing assumes knowledge of the probability distribution of object defectiveness and leverages that information to minimize the expected number of group-tests required [23]. In contrast, the *combinatorial* variation aims at minimizing the number of group-tests required in the worst case-scenario [10]. Our setting is more aligned with the combinatorial variation; however, with the knowledge of prior likelihood of causing failure for the predicates, probabilisitic variation would be better suited. Other variations of adaptive group-testing are studied in the literature such as group-testing with *geometric restriction*: objects can form a group if they are adjacent to each other or forms a valid path within a network. Karbasi and Zadimoghaddam [21] studied the problem of sequential group-testing with graph constraints, where a group-test is admissible only if the group induces a connected subgraph within the given constraint-specification graph. Although we do not consider any restriction on groups that can be tested, constrained group-testing has applicability in our case where we need to consider interference among interventions. In that case, possible interference can be modeled by absence of edges in the constraint graph, and each valid group must induce a clique to ensure that predicates under intervention do not interfere with each other.

Causal inference. Attariyan et al. [2, 3] attribute performance issues to configuration settings and program inputs, and use dynamic information flow tracking to estimate root cause of performance anomalies. While they observe causality within application components through runtime control and data flow, they only report a list of root causes ordered by the likelihood of being faulty without providing further causal paths that connect root causes to performance anomalies. Moreover, the method uses *taint* tracking which requires low level dynamic binary instrumentation. Beyond statistical association (e.g., correlation) between root cause and failure, Baah et al. [4, 5] and Shu et al. [30] apply causal inference methodology for counterfactual inference on observational data towards software fault localization. To provide the context of failure, Inforence [11] applies feature selection and statistical causal inference techniques to identify specific combination of statements that together cause program failure. However, these approaches use observational data collected from program execution logs, which is often limited in capturing rare scenarios. This is due to the fact that observational data is not generated by randomized controlled experiments where treatment selection on variables is random. Particularly, for the task of discovering the intermediate predicates in a complete causal path, observational data is insufficient, as it does not always satisfy *conditional exchangeability*, a key requirement for applying causal inference on observational data.

Discriminating subgraph. Cheng et al. [8] look for discriminating subgraphs within the set of program control flow graphs towards identifying bug signature for sequential programs. However, since they overlook the notion of causality, such discriminative subgraph would contain any correlated but spurious predicate which is triggered by the root cause, but does not cause failure. Moreover, in their work, the unit of a bug signature is program statement which fails to model compound predicates required to capture incidents of concurrent programs. As such, it is not straightforward to apply their method to concurrent programs. However, such an adoption is complementary to our work towards discovering the initial causal-capability DAG.

Statistical debugging. Snorlax [22] and Gist [22] employ statistical diagnosis to rank the program predicates based on their likelihood of being the root causes of concurrency bugs. Spectrum-based fault localization techniques [27] rank program statements or blocks instead of predicates

that capture interaction among statements. However, these and other statistical debugging approaches [9, 16, 24, 26, 31, 34] suffer from the issue of not separating correlated predicates from the causal ones, and fail to provide contextual information regarding *how* root causes lead to program failures.

Control flow path. Work by Jiang et al. [15] aims at generating faulty control flow paths that link many bug predictors, but does not consider causal connection among those bug predictors and program failure. Therefore, the approach produces sub-optimal result in presence of spurious paths that connect many spurious bug predictors than causal paths that connect fewer but real bug predictors. Differential slicing [17] aims towards discovering causal path of differences that led from input differences to the observed difference, but requires complete program execution trace generated by execution indexing [33]. Dual slicing [32] is another program slicing-based technique to discover statement level causal paths for concurrent program failures. However, this approach does not include compound predicates that capture certain runtime conditions during concurrent program execution, and hence fail to draw causal connection among those runtime conditions. In general, program slicing based approaches are limited in many aspects, such as dealing with a set of runs, instead of just two.

Intervention. Perhaps the closest to our work in terms of intervention and predicate pruning technique is Pinso [25]. Pinso applies a filtration-oriented scheduler to eliminate false positive root causes involving certain interleaving patterns in shared memory access. It also applies pruning strategies that share some similarity with ours. However, it does not perform any direct intervention to the program, rather enforces particular thread schedules to *keep* or *avoid* certain memory access patterns.

10 Conclusions

In this work, we defined and motivated the problem of causal path discovery for explaining incidents in concurrent programs. Our key contribution is the novel active statistical debugging framework, which applies program-intervention techniques to pin point the causal path that connects the root cause of program failure to the failure indicating symptom. Such causal path provides better interpretability for understanding and analyzing the root causes of program failure. We showed theoretically and empirically that active statistical debugging 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 model the causal capability relationship among predicates, and address the cases of multiple conjunctive root causes with disjunctive causal paths. Furthermore, we would like to address the challenge of explaining multiple types of failures as well.

References

[1] A. Agarwal, S. Jaggi, and A. Mazumdar. Novel impossibility results for group-testing. In *2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018*, pages 2579–2583, 2018.

[2] M. Attariyan, M. Chow, and J. Flinn. 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*, pages 307–320, 2012.

[3] M. Attariyan and J. Flinn. 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*, pages 237–250, 2010.

[4] G. K. Baah, A. Podgurski, and M. J. Harrold. Causal inference for statistical fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 73–84, New York, NY, USA, 2010. ACM.

[5] G. K. Baah, A. Podgurski, and M. J. Harrold. 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*, pages 146–156, 2011.

[6] Y. Bai, Q. Wang, C. Lo, M. Liu, J. P. Lynch, and X. Zhang. Adaptive bayesian group testing: Algorithms and performance. *Signal Processing*, 156:191–207, 2019.

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

[8] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. 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*, pages 141–152, 2009.

[9] T. M. Chilimbi, B. Liblit, K. K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: effective statistical debugging via efficient path profiling. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 34–44, 2009.

[10] D.-Z. Du and F. K. Hwang. *Combinatorial group testing and its applications*. World Scientific, Singapore River Edge, N.J, 1993.

[11] F. Feyzi and S. Parsa. Inference: Effective fault localization based on information-theoretic analysis and statistical causal inference. *CoRR*, abs/1712.03361, 2017.

[12] J. Y. Halpern and J. Pearl. 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*, pages 194–202, 2001.

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

[14] F. K. Hwang. A method for detecting all defective members in a population by group testing. *Journal of the American Statistical Association*, 67(339):605–608, 1972.

[15] L. Jiang and Z. Su. 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*, pages 184–193, 2007.

[16] G. Jin, A. V. Thakur, B. Liblit, and S. Lu. 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*, pages 241–255, 2010.

[17] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song. Differential slicing: Identifying causal execution differences for security applications. In *IEEE Symposium on Security and Privacy*, pages 347–362. IEEE Computer Society, 2011.

[18] J. A. Jones and M. J. Harrold. 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*, pages 273–282, 2005.

[19] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477. ACM, 2002.

[20] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Trans. Computers*, 44(2):248–260, 1995.

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

[22] B. Kasikci, W. Cui, X. Ge, and B. Niu. Lazy diagnosis of in-production concurrency bugs. In *SOSP*, pages 582–598. ACM, 2017.

[23] T. Li, C. L. Chan, W. Huang, T. Kaced, and S. Jaggi. Group testing with prior statistics. In *2014 IEEE International Symposium on Information Theory, Honolulu, HI, USA, June 29 - July 4, 2014*, pages 2346–2350, 2014.

[24] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. 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*, pages 15–26, 2005.

[25] B. Liu, Z. Qi, B. Wang, and R. Ma. Pinso: Precise isolation of concurrency bugs via delta triaging. In *ICSME*, pages 201–210. IEEE Computer Society, 2014.

[26] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Software Eng.*, 32(10):831–848, 2006.

[27] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, 2011.

[28] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209. ACM, 2011.

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

[30] G. Shu, B. Sun, A. Podgurski, and F. Cao. MFL: method-level fault localization with causal inference. In *ICST*, pages 124–133. IEEE Computer Society, 2013.

[31] A. V. Thakur, R. Sen, B. Liblit, and S. Lu. 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.*, pages 35–41, 2009.

[32] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan. Analyzing concurrency bugs using dual slicing. In *ISSTA*, pages 253–264. ACM, 2010.

[33] B. Xin, W. N. Sumner, and X. Zhang. 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*, pages 238–248, 2008.

[34] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. 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*, pages 1105–1112, 2006.

[35] A. X. Zheng, I. Rish, and A. Beygelzimer. 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*, page 675, 2005.