Pinso: Precise Isolation of Concurrency Bugs via Delta Triaging

Bo Liu†, Zhengwei Qi§, Bin Wang†, Ruhui Ma†
Shanghai Key Laboratory of Scalable Computing and Systems
†Department of Computer Science and Engineering, §School of Software
Shanghai Jiao Tong University, China
{boliu, qizhwei, binqbu2002, ruhuima}@sjtu.edu.cn

Abstract—Concurrent programs are known to be difficult to test and maintain. These programs often fail because of concurrency bugs caused by non-deterministic interleavings among shared memory accesses. Even though a concurrency bug can be detected, it is still hard to isolate the root cause of the bug, due to the challenge in understanding the complex thread interleavings or schedules. In this paper, we propose a practical and precise isolation technique for concurrent bugs called Pinso that seeks to exploit the non-deterministic nature of concurrency bugs and accurately find the root causes of program error, to further help developers maintain concurrent programs. Pinso profiles runtime inter-thread interleavings based on a set of summarized memory access patterns, and then, isolates suspicious interleaving patterns in the triaging phase. Using a filtration-oriented scheduler, Pinso effectively eliminates false positives that are irrelevant to the bug. We evaluate Pinso with 11 real-world concurrency bugs, including single- and multi-variable violation, from sever/desktop concurrent applications (MySQL, Apache, and several others). Experiments indicate that our tool accurately isolates the root causes of all the bugs.

Keywords—Software testing and debugging; Concurrency bug;

I. INTRODUCTION

The prevalence of multi-core processors has led to an increased use of concurrent programs. In the real world, a large number of high-end critical softwares are designed with multi-thread or multi-process. However, these programs are error prone because of concurrency bugs. It is surveyed that two-thirds of developers handle concurrency issues and bugs at least monthly [1]. Concurrency bugs have caused some of the most serious computer-related accidents in history [2][3][4].

Testing concurrency bugs is a hard problem because they occur in non-deterministically executed code interleavings. In concurrent programs, several threads of execution share a single address space and coordinate their accesses to shared variables via synchronization operations, such as locks and condition variables. Therefore, it is a challenge for programmers to reproduce a failed execution and understand the root cause of the bug.

One common practice for isolating concurrency bugs is record-and-replay solution, represented in [5][6][7]. In particular, a programmer is able to record multi-threaded program executions at an acceptable performance cost, so that she can deterministically replay them if they were to fail. However, even if developers reproduce a concurrency bug with a test, they often spend significant time understanding it. Furthermore, developers often misunderstand a bug and create an incorrect patch.

When trying to understand the root cause for the bug that caused the failure, a programmer is still confronted with a complex graph of inter-thread memory dependencies to reason about. Therefore, another possible solution is to replay the failed execution and analyze it, using generic dynamic bug finding tools such as data-race detectors [8] and atomicity violation checkers [9]. These tools could help programmers find specific classes of bugs, and hence do not target root causes of the concurrency bugs.

In recent years, research points gradually focused on code coverage, such as prior work on Tarantula for sequential programs [10] and others [11][12][13]. These methods work by associating the number of occurrences of a target coverage criterion with passing and failing executions, and use these data to compute suspiciousness scores. For example, Maple [13] defined coverage for multithreaded programs based on a set of thread interleaving idioms. However, coverage detection has not been applied to concurrency interleaving patterns. Besides, false positives are a big concern in these tools. Many previous detectors have high false positive rates [14][15]. Narayanasamy et al. [16] show that only about 10% of real data races are harmful and could cause software failure.

In this paper, we present Pinso†, a precise concurrency bug isolation tool. The goal of Pinso is to help programmers triage a concurrency bug by analyzing a failed execution and automatically isolate its root causes. It takes advantage of the fact that concurrency bugs are non-deterministic. For a given bug triggering input, there correspond a huge number of legal thread interleavings, many of which tend to produce correct executions. Our observation shows that, correct interleavings tend to manifest much more frequently than the incorrect interleavings for a given bug triggering input, especially in applications that adhere to strict software quality assurance process. Pinso exploits the above observation to isolate the root causes of a failed execution. Using the same bug triggering input that caused the failure, Pinso produces several alternative correct executions. Any thread interleaving that manifests only in the incorrect execution but does not manifest in any of the correct executions is reported to be the root cause of the failure.

We define a root cause of a concurrency bug to be a

†Pinso is available to download at https://github.com/poeliu/Pinso
set of necessary inter-thread dependency conditions (patterns) that need to be satisfied in an execution before a bug is triggered. During testing, Pinso detects access patterns from actual program executions, which either pass or fail. Pattern-based interleaving manifested only in the failed execution but never observed in correct executions might be the root cause of the bug. Furthermore, based on these summarized interleaving patterns and context information, a programmer can easily understand the bugs at a high level.

Another challenge that Pinso overcomes lies in reducing the false positives (interleaving patterns which unrelated to a concurrency bug) from being reported to the programmer. While it is relatively easier to produce correct executions given a bug triggering input, they need to be diverse enough to eliminate all the suspected patterns from the incorrect execution. Naively running the program again and again for a bug triggering input is not sufficient to expose a diverse set of schedules. One option could be using a systematic testing tool such as CHESS [17] to expose all possible executions for a given bug triggering input, but unfortunately it does not scale well for long running programs. To address the above challenge, we use a filtration-oriented scheduler that takes a suspected interleaving pattern and tries to produce an execution where the suspect is exposed. If the resulting execution produces a correct answer, then the suspect can be discarded. But if it ends up producing an incorrect execution, even then the execution could also be valuable, because any remaining suspects that did not manifest in the new incorrect execution can be discarded. In Section VI, we discuss in detail the three stages of filtration-oriented scheduler for each suspect to determine if it is a true positive or not.

Most of existing isolation tool for concurrency bug are lack of precision, since they tend to isolate the bug using ranking strategy and then report the failure-induce interleaving with a suspiciousness score [11][18][19]. Programmers need to check the report manually. This observation implies both opportunities and challenges. If we could reduce the redundant false positives which is root cause suspects but is not related to the concurrency bug actually, fault-localization precision could improve significantly. To address this challenge and exploit the opportunity, we propose a filter in this paper to identify which suspects are the root causes and exclude the irrelevant ones. The filter plays an important role in precisely finding the root causes of the bugs.

Overall, this paper makes the following contributions:

- A bug isolation technique that handles a wide variety of concurrency bugs, including data races, order violation and atomic violation involving single or multiple variables, while requiring no changes to software and hardware that is suitable for production-running.
- A novel way of using a three-stage filtration-oriented scheduler to filter false positives, which is the main reason of why our localization technique is more accurate than existing technique based on ranking strategy. To our best knowledge, this is one of the first techniques that exploit existing fault localization technique to isolate the root causes of the concurrency bugs without false positives.

- An evaluation of 11 real-world concurrency bugs from open-source server/desktop programs, which shows that our tool can effectively pinpoint the root causes of these bugs.
- Our isolation framework for concurrent bugs and all the testing tools we developed are made available to the public under the Apache 2.0 license. They can be downloaded from online open source community.

The rest of the paper is organized as follows. Section II presents the background on interleaving pattern definition. Section III provides an overview of the Pinso framework. Sections IV, V and VI respectively describe our three phase strategies in detail. We present experimental results in Section VII and related work in Section VIII. Section IX concludes.

II. BACKGROUND: INTERLEAVING PATTERNS

<table>
<thead>
<tr>
<th>Pattern Definitions</th>
<th>Real-world Interleaving Bug Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Thread 1</td>
</tr>
<tr>
<td>A</td>
<td>Bx</td>
</tr>
<tr>
<td>Dy</td>
<td>Dy</td>
</tr>
<tr>
<td>Pattern 1</td>
<td>Pattern 2</td>
</tr>
<tr>
<td>Ac</td>
<td>Ca</td>
</tr>
<tr>
<td>Bx</td>
<td>Dx</td>
</tr>
<tr>
<td>Pattern 2</td>
<td>Pattern 3</td>
</tr>
<tr>
<td>Ac</td>
<td>Bx</td>
</tr>
<tr>
<td>Ca</td>
<td>Dx</td>
</tr>
<tr>
<td>Pattern 3</td>
<td>Pattern 4</td>
</tr>
<tr>
<td>Ac</td>
<td>Bx</td>
</tr>
<tr>
<td>Ca</td>
<td>Dx</td>
</tr>
<tr>
<td>Pattern 4</td>
<td>Pattern 5</td>
</tr>
<tr>
<td>Ac</td>
<td>Bx</td>
</tr>
<tr>
<td>Ca</td>
<td>Dx</td>
</tr>
<tr>
<td>Pattern 5</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1: Definition of Interleaving patterns for two threads and two inter-thread dependencies

Interleaving root cause is the set of necessary inter-thread dependency conditions, which need to be satisfied in an
execution before a concurrency bug is triggered. A programmer could use root cause to determine whether a thread interleaving is illegal, so that he can fix a concurrency bug.

A practical tool should limit the number of necessary conditions that are represented in an interleaving root cause. Otherwise, programmers would potentially be dealing with an unbounded set of interleaving conditions to analyze. To handle such property, previous studies on small scope hypothesis [13][17][20] showed that a majority of the concurrency bugs can be triggered using a small set of thread interleavings (two threads and two inter-thread dependencies). Such observation is leveraged when defining the set of the interleaving patterns in Pinso.

Figs. 1 shows the five interleaving patterns for consideration. They are canonical and sufficient to isolate the root causes so that they capture concurrency bugs involving no more than two threads and two inter-thread dependencies. For example, pattern1 is a simple interleaving pattern, while others are complex patterns. Complex patterns are used to reduce the set of possible root causes without significantly compromising the ability to isolate a concurrency bug. In particular, the five patterns scale 3 typical root causes: atomicity violations, including both single variable (pattern1, pattern2, pattern3) and multi-variable (pattern4, pattern5); typical deadlock bugs (pattern4, pattern5), and generic order related concurrency bugs (pattern1).

In the pattern definition, \( A_X \) is a data access, \( A \) denotes the static address of the memory instruction and \( X \) refers to the memory location accessed. Two data accesses are conflicting if both access the same memory location and at least one of them is a write. If \( A_X \) is a lock/unlock synchronization access, \( A \) denotes the address of the instruction that invoked the lock/unlock operation and \( X \) refers to the lock variable. Two synchronizations accesses conflict if one of them is a lock and the other is an unlock operation. Symbol \( \rightarrow \) denotes an inter-thread dependency between two accesses. Column 2 of Figs. 1 provides real-world bug examples that have the root causes with type pattern1 to pattern5 respectively.

To verify our hypothesis that achieving high coverage for our simple set of interleaving patterns could cover a significant fraction of concurrency bugs, we conducted an empirical study using 31 real world concurrency bugs from various programs including Apache, MySQL, psfscan, Aget, HTTrack, Pbzip2, Mozilla and OpenLDAP. All of the 31 concurrency bugs can be characterized using one of our 5 interleaving patterns.

III. SYSTEM OVERVIEW

As shown in Figure 2, Pinso bases on dynamic analysis technique. It create an analyzer using the following modules: Profiler, Triager and Filter.

Profiler: In the first step of delta triaging, profiler analyzes the failed execution to determine the set of the interleaving patterns observed in that execution. Then it uses the bug triggering input and produces alternative thread schedules using a variant of the PCT scheduler [21], which is very efficient in term of performance. In our experience, these alternative executions mostly produce correct results as buggy thread interleavings tend to manifest fairly infrequently, even while executing for a bug triggering input. These alternative executions are also profiled to determine the interleaving exposed in those executions. Any interleaving pattern that is exposed in the incorrect executions but does not appear in any of the correct execution is flagged as a "suspect".

Triager: In the second phase of delta triaging, we use triager in order to find ourselves with a fairly large number of suspects. We continue to produce and analyze alternative schedules using the random scheduler as long as we continue to reduce the number of suspects. When we reach a point in our analysis where random scheduler is no longer able to prune any suspects, Pinso will move on to the third step.

Filter: The last step of delta triaging uses our filtration-oriented scheduler for a more targeted scheduling to filter a specific suspect. Our scheduler has the ability to orchestrate a thread schedule to either avoid or preserve (keep) a suspected pattern in an execution. If it manages to produce a correct schedule with a suspect, the set of interleaving can be safely discarded. If the scheduler produces an incorrect execution while trying to expose a suspected pattern, any suspect that did not manifest during the new incorrect execution could be discarded. For such patterns that resulted in an incorrect execution during deterministic scheduling, we will again use the scheduler to preserve that interleaving pattern, but this time we would also try to avoid other remaining suspects. This approach allows us to quickly filter the remaining suspects once we have identified the root interleaving pattern that causes the program to fail during scheduling.
Pinso relies on the assumption that a programmer would be able to classify an execution as either correct or incorrect. This process is clearly easier if the failed execution has visible symptoms such as exceptions. Fortunately, many concurrency bugs lead to such visible conditions. Future work could potentially learn invariants and help automate this process [22].

IV. PROFILING INTERLEAVING PATTERNS

In this section, we discuss our approach to isolate suspects. We take the program and the corresponding bug triggering input, profile it using a few number of runs, observe the patterns exposed in the succeeded runs as well as in the failed runs, and compare them to isolate the suspects.

As we discussed in Section II, an interleaving pattern is comprised of a set of inter-thread dependencies between memory accesses. A memory access could be either a data access or a synchronization access. For synchronization access, we only consider lock and unlock operations in this paper. A lock or an unlock operation is treated as a single access when we construct a pattern, and all memory accesses executed within the lock and unlock functions are ignored. For readability, a memory access is usually referred to as an access. We observed that non-mutex happens-before relations (e.g. signal/wait, fork-join) mostly remain the same across different executions for a given input. Therefore, the profiler predicts an dependence only if it does not violate the non-mutex happens-before relations in at least one of the profiled executions. Our profiler uses an online algorithm to observe exposed interleaving patterns in both succeeded and failed runs. In each profile run, the profiler observes all types of patterns exposed.

A. Algorithm for Base Pattern

Observing exposed interleaving pattern of pattern1 type is fairly straightforward. The profiler monitors every access. For each object X, the profiler maintains a meta-data which contains the information about the last write access to X and the set of read accesses to X that happen after the last write access. When an access AX is being executed by a thread Ti, the profiler checks the meta-data maintained for object X. If it finds a conflicting access BX from Tj (i ≠ j), a pattern1 interleaving B → A will be reported, shown in Algorithm 1.

B. Supporting Complex Patterns

In order to observe exposed interleaving for complex patterns, the profiler needs to maintain additional meta-data. For each thread Ti, the profiler keeps track of the recent accesses from Ti in an ordered list, called recent access history. We use RH(Ti) to denote the recent access history for thread Ti. Accesses in RH(Ti) are ordered according to the execution order of Ti. Every time an access is being executed by a thread Ti, the profiler pushes it to the end of RH(Ti) and pops any stale accesses from the front. The size of the recent access history is limited by the size of the vulnerability window in the pattern definitions. In other words, the number of dynamic instructions executed between the first access and the last access in the recent access history should not exceed the size of the vulnerability window in the interleaving pattern definitions.

Algorithm 1 Algorithm for Base Pattern Identification

1: Initially: X: shared memory access location
2: A: memory access instruction
3: type: memory access type
4: Ti: thread id
5: Output: PT: patterns are detected and updated online
6: W: store meta-data for each X
7: W.p: predecessor instruction
8: W.c: current instruction (from other threads)
9: while A ∈ Shared memory instruction do
10: if the X accessed by A does not have W then
11: W = CreateWindow(p, c);
12: W.p.insert(A, type, X);
13: RegisterWindow(W, X);
14: else
15: W = GetWindow(X);
16: W.p == W.GetLastAccess();
17: if Ti != W.p.Ti & & (type == p.type == read) then
18: W.c.insert(A, type, Ti);
19: PT = W.CreatePattern(p, c);
20: W = SlideWindow(p=c, c=null);
21: else
22: W.p.update(A, type, Ti);
23: end if
24: end if
25: end while
26:

For each access AX in the recent access history RH(Ti), the profiler also maintains a set of accesses called successors (denoted by Successors(AX)) such that for any access BX ∈ Successors(AX), AX and BX form a dynamic inter-thread dependency (i.e. AX → BX) according to the algorithm for pattern1. Let us consider the example in Fig. 3. When CX is being executed by T2, the pattern1 algorithm finds an inter-thread dependency between AX and CX. Therefore, CX is added to Successors(AX). If an access is removed from the recent access history, its corresponding successors will be removed as well.

We now illustrate our algorithm for complex patterns using the example shown in Fig. 3. When BY is being executed by T1, the profiler first applies the algorithm for pattern1 to find those accesses that can form inter-thread dependencies with

![Fig. 3: Observing exposed interleaving patterns for complex pattern type](image)

---

2For lock variables, we treat all accesses to them to be write accesses.
By. Those accesses are put into a set called predecessors, denoted by Predecessors(By). Next, the profiler checks the recent access history RH(T1) and finds that \( AX \) is in RH(T1). As we just discussed, the profiler should have Successors(AX) at this moment. Then, the profiler scans every access in Successors(AX) and Predecessors(By), and find that a pattern interleaving \( A \rightarrow C \rightarrow D \rightarrow B \) is exposed. Finally, the profiler updates the meta-data by adding \( By \) to Successors(Dy) if Dy is still in RH(T2).

V. ISOLATING SUSPECTS THROUGH TRIAGING

In this section, for trouble shooting, we use a mature practice that has been used by many of the existing bug isolation techniques [11][23][24]. For example, each profile run is labeled with a boolean flag indicating whether the execution is succeeded or failed. Currently, we ask programmers to provide hints for the labeling (e.g. scripts) since they are program and input specific.

To isolate suspects, we collect two sets of interleaving patterns, Succ (succeeded) and Fail (failed), during profile runs. Succ contains those patterns that are exposed in one of the succeeded runs. Fail contains those patterns that are exposed in failed runs. We consider an interleaving pattern to be a suspect if it is in Fail but not in Succ:

\[
\text{Suspects} = \text{Fail} - \bigcup_{i=1}^{N} \text{Succ}(i)
\] (1)

For example, the root cause suspects that appeared in all of the failed runs, but never appeared in succeeded runs are highly likely to be related to the bug. Investigating those suspected patterns is beneficial, and helps programmers to quickly understand the bug.

There are two ways to reduce false positives (suspects that are irrelevant to the bug). One way is to increase the size of Succ. That is because we want to get more correct interleavings so that a correct interleaving will less likely to be falsely treated as a suspect even if it appears in all failed runs. Simply running the program frequently tends to exercise the same interleavings. Therefore, to increase the size of Succ, we apply the best random testing technique during profile runs. The best random testing technique is a variation of the PCT algorithm [21]. The algorithm is similar to PCT but it uses non-strict priorities instead of strict priorities. It achieves higher coverage faster than PCT for most of the benchmarks. We choose to stop the profiling once Succ does not increase in consecutive runs (In our experiments, we use an empirical value 50).

Another way is to reduce the size of Fail. To achieve that, programmers may want to produce more distinct failed runs. However, even using the best random testing technique, the probability to expose a failed run is still very low. Therefore, we need more advanced techniques. Our filtration-oriented scheduler is one of them. We will discuss it in detail in Section VI.

Notice that our technique requires at least one failed run. To ensure that, we need support from a record and replay tool [13][25]. Unlike the poor detection from the previous techniques [11][26], our approach works well even with only one failed run.

VI. FILTER FALSE POSITIVES USING SCHEDULER

After the two previous phases, the programmer receives a set of suspicious interleaving patterns (suspects) which might be relevant to the bug. If the size of the suspects is small, the programmer can just manually examine each of them. But more often the size of suspects at this moment is usually quite large. For example, in our experiments, about half of the bugs to be analyzed have about 300 suspects after the triaging phase. Manually checking each of them will be very tedious.

One practical approach is to use a ranking mechanism with the hope that the real root cause has a higher rank than other suspects. However, most of the existing techniques [11][23][26] are based on either heuristics or statistical reasons. In the worst case, the programmer still has to manually examine each of the suspects.

In this paper, we propose a new approach to solve this problem. Our motivation is still the same as that described in Section V, which is increasing the size of Succ and reducing the size of Fail. Our novelty, however, stems from the way we use to achieve the above goal. The main idea is to leverage a thread scheduler to exercise more targeted and relevant interleavings to the suspects so that we have a higher chance of increasing the size of Succ and reducing the size of Fail than a random technique.

A. Filtration-Oriented Scheduler

Pinso profiler and triager have predicted a set of root cause suspects. The goal of the scheduler is to validate the prediction by orchestrating the thread schedule to realize the predicted patterns in an actual execution for the test input.

For a given program and input, scheduler takes one of the suspects as the input, orchestrates the thread schedules trying to expose the given interleaving pattern in an actual execution. It leverages the non-preemptive and strict priorities provided by the underlying operating system to control the thread schedules. Under this scenario, a lower priority thread never gets executed if there exists a non-blocked higher priority thread. All threads are forced to run on a single processor. Each thread is assigned with a non-preemptive strict priority. By changing the priorities at the specific program points according

![Fig. 4: Filtration-Oriented Scheduler](image-url)
to the given interleaving dependence, the filtration-oriented scheduler is able to produce a patterns specific execution that has a higher chance of exposing it.

Besides preserving suspected patterns to expose the interleaving deterministically, called KEEP, also introduce another mode called AVOID. In AVOID, instead of trying to expose the given pattern, the scheduler will try to avoid exposing it. AVOID will be useful when we are filtering suspects. Interestingly, avoiding an interleaving pattern in an execution is much easier than exposing it, since we just need to prevent one of the inter-thread dependencies in the pattern from manifesting during the execution.

The algorithm of AVOID is easy to understand. Suppose that we want to avoid a pattern interleaving \( A \rightarrow B \). If the candidate instruction \( A \) is reached by a thread \( T_i \), we change the priority of \( T_i \) to be lower than all the other threads. And if the candidate instruction \( B \) is reached by a thread \( T_j \), we change the priority of \( T_j \) to be higher than all other threads. For a complex pattern, we always choose to avoid the first inter-thread dependency in the pattern using the same algorithm as we use for pattern interleaving. Like KEEP, the algorithm for AVOID is a best effort algorithm. In other words, it is still likely that a suspected pattern is exposed even if the scheduler tries to avoid exposing it.

Assume that we want to use our filtration-oriented scheduler to filter the pattern suspect \( A \rightarrow B \) in the example of Fig. 4, where \( A_X \) is in thread 1 and \( B_X \) in thread 2 respectively. The idea is to use two test runs on each root cause candidate (suspect). Each newly created thread \( T_i \), including the main thread, is assigned with an initial priority \( P_{init}(T_i) \) which assigned normal priority. In KEEP mode, when thread 1 reaches \( A_X \), scheduler changes the priority of thread 1 to higher priority, in order to guarantee \( A_X \) would be executed first. Therefore, when thread 2 reaches \( B_X \) ahead of thread 1, the scheduler changes the priority of thread 2 to lower. Then, once \( A_X \) was executed successfully, scheduler would switch its priority from higher to lowest value, so that context would switch to thread 2 to execute \( B_X \). Finally, the suspect \( A \rightarrow B \) is exposed. The second test run is in AVOID mode. As mentioned in the last paragraph, only one change point is needed in AVOID mode, as shown in Fig. 4.

**B. Filtering Suspects**

The filtering process is composed of three stages. Each stage takes a set of suspects as input, applies the filtering, and produces a reduced set of suspects which will then be fed into the next stage. Fig. 5 shows the workflow of the filtering process. The size of the suspects becomes smaller after each stage. Notice that we will not move to the next stage until all the suspects are examined by the previous stage.

![Fig. 5: Overview of the suspects filtering process](image)

All stages share the same rules when filtering suspects: 1) if a suspect is exposed in a succeeded run, it will be filtered as it is now in Succ; 2) if a suspect is NOT exposed in a failed run, it will be filtered as well as it is no longer in Fail. The stages only differ in the way they use the scheduler. We will discuss each of them in detail. In fact, this is really the key of our filtering technique: how to produce an interleaving that is more relevant and more targeted to a specific suspect such that we have a high chance to filter it.

1) **Stage1:** Fig. 6a depicts the workflow of stage1. In each run, we first pick a suspect (called target suspect), and then profile the execution under the scheduler KEEP which tries to expose the target suspect. The profiler is the same with the description in Section IV. If the resultant run succeeded, we apply the filter F1 to filter those suspects that are exposed in the succeeded run. If the resultant run failed, we apply the filter F2 to filter those suspects that are NOT exposed in the failed run. Our rationale behind F1 is that: if the target suspect is irrelevant to the bug and it is exposed during the execution, the probability that the execution fails should be comparable to the probability that the bug is exposed in a random execution, which is very low. Therefore, in such a case, the target suspect is very likely to be filtered. For F2, we expect it to be effective as well because distinct failed runs are precious in reducing the size of Fail. The effectiveness of each filter will be discussed in Section VII-C.

2) **Stage2:** Fig. 6b describes the workflow of stage2. Besides the suspects, stage2 also requires some information from
stage1. In stage1, we record all the failure inducing patterns. A failure inducing pattern is the root cause that causes a failed run. Using keep on a failure inducing pattern is very likely to cause another failed run. Leveraging this nice property, in each run, we feed keep with a failure inducing pattern and slightly perturb the execution by avoiding a suspect using avoid in the same run. With such approach, it is likely that we still get a failed run and the suspect to avoid is not exposed in the failed run, in which case we can use avoid to filter this suspect. In fact, avoid is a more sophisticated version of keep in the sense that it perturbs the failed executions in a more targeted way.

3) Stage3: Fig. 6c depicts the work flow of stage3. This stage is similar to stage1 but it uses avoid instead of keep, for each suspect. If the run succeeded, we apply filter avoid to filter those suspects that are exposed. If the run failed, we apply filter avoid to filter those suspects that are not exposed.

VII. EXPERIMENTAL RESULTS

A. Methodology

Pinso is implemented using the PIN binary instrumentation framework [27] and runs on a real machine with multiple processors. It is able to handle unmodified x86 binaries. All experiments are conducted on a Linux workstation with a quad core Intel Xeon processor running at 2.4GHz and 8GB of main memory. We use 6 open source multi-threaded applications throughout our experiments shown in Table I. Two of them are server applications (MySQL and Apache), while other 4 are desktop applications (Pbzip2, Pfscan, Aget and HTTrack). In order to evaluate the bug isolation capability, we choose 11 documented bugs along with their bug triggering inputs from the 6 applications, as shown as Table II. Because there is no standard benchmark suite for use in fault-localization of concurrency bugs, which is common to all existing work in the area, we used widely-used open-source programs with known bugs, to reduce this threat.

Firstly, for a given bug and its bug triggering input, we produce a single failed execution (which can be achieved using a record and replay tool such as [25]). And then we start profiling the program using a random testing technique described in [21]. After each profile run, we update either succ or fail, depending on a success or a failure of the execution. The set of suspects will be updated accordingly. In fact, in all of our experiments, we never observe a single failed execution during profiling. The profiling will stop when the size of the suspects does not change in consecutive 50 runs. Once the profiling is done, we move to the filtering part. The set of suspects goes through each stage one by one. In our experiment, each stage will give each suspect a second chance if it is not filtered in the first attempt. Finally, after the filtering, we report the set of suspects to the programmer.

B. Bug Isolation Capability

The first question is how effective the Pinso framework is in isolating concurrency bugs. We consider a bug isolation technique to be effective if it can always isolate the cause of the bug (no false negatives) and does not report too many irrelevant warnings (low false positives). To evaluate that, we apply our Pinso framework to all the 11 bugs.

Overall, as shown in Table II, Pinso can precisely isolate all the tested concurrency bugs efficiently. Further investigation reveals that our filtration-oriented scheduler plays a very important role in reducing the false positives. Column 7 lists the number suspects before we apply the filters. As can be seen from the table, 5 out of 11 bugs still have about 300 suspects before the filters are applied. If we do not apply the filtering, the programmer might end up examining all the 300 suspects manually, which is time consuming. Column 9 presents the number of root cause suspects after filtering, which implies that a programmer need only look at the # after filtering patterns reported by Pinso to find an actual bug. It is noticed that some bugs have more than one patterns in suspect list after filtering. For example in #4 bug Pbzip2, there are 3 suspects eventually. The reason is that it contains a pattern 3 root causes and two pattern 1 subparts of it, such as $A_X \rightarrow B_X \rightarrow C_X$, $A_X \rightarrow B_X$ and $B_X \rightarrow C_X$.

C. The Filters Effectiveness

As already shown in Table II, our filters are effective in pruning suspects. In what follows, we want to know how effective each filter is. This will give us some insights about an effective filter.

Table III shows the results. We apply the filters according to the order described in Section VI-B. Column 6, 8 and 11 show the number of scheduler runs used by stage1, stage2 and stage3 respectively. Filters F1 and F2 are the two most effective filters. On average, F1 filters about 40% of the total false positives and F2 filters about 55%. We notice that none of the false positives are filtered by F4 and F5. This is not because these two filters are not effective. The real reason is that none of the false positives actually escape from the first 3 filters.

D. Profiling Based on Random Testing Technique

After that, we evaluate the benefits of using a random testing technique during profile runs. We compare the following two approaches: 1) apply the random testing technique for each profile run (denoted by random); 2) do not apply any testing technique (denoted by base). Both approaches will stop the profiling when no suspect is pruned in 50 consecutive runs.

Figure 7 plots the results. In each sub-figure, x-axis is the number of profile runs and y-axis is the number of suspects remaining. From the results, we observe two benefits of using random during profiling. First, the number of suspects reduces faster than base, which potentially reduces false positives. Second, the profiling saturates earlier than base, which reduces triaging time.
TABLE II: The bug isolation capability

<table>
<thead>
<tr>
<th>Bug #</th>
<th>App.</th>
<th>Version</th>
<th>Pattern Type</th>
<th># Profile Runs</th>
<th># Total patterns</th>
<th># Suspects</th>
<th># Schedules</th>
<th># After Filtering</th>
<th># False Positives</th>
<th>Isolated?</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Pfscan</td>
<td>1.0</td>
<td>Pattern1</td>
<td>76</td>
<td>273</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#2</td>
<td>Aget</td>
<td>0.4-dev</td>
<td>Pattern1</td>
<td>107</td>
<td>459</td>
<td>31</td>
<td>37</td>
<td>1</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#3</td>
<td>HTTrack</td>
<td>3.43.9</td>
<td>Pattern1</td>
<td>105</td>
<td>1102</td>
<td>26</td>
<td>29</td>
<td>1</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#4</td>
<td>Pbzip2</td>
<td>0.9.4</td>
<td>Pattern2</td>
<td>99</td>
<td>292</td>
<td>9</td>
<td>12</td>
<td>3</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#5</td>
<td>Apache</td>
<td>2.0.48</td>
<td>Pattern2</td>
<td>214</td>
<td>7011</td>
<td>342</td>
<td>290</td>
<td>3</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#6</td>
<td>MySQL</td>
<td>4.0.12</td>
<td>Pattern2</td>
<td>235</td>
<td>7974</td>
<td>298</td>
<td>121</td>
<td>3</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#7</td>
<td>Aget</td>
<td>0.4-dev</td>
<td>Pattern3</td>
<td>185</td>
<td>574</td>
<td>6</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#9</td>
<td>Apache</td>
<td>2.0.46</td>
<td>Pattern3</td>
<td>309</td>
<td>7347</td>
<td>156</td>
<td>130</td>
<td>3</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#10</td>
<td>MySQL</td>
<td>3.2.35b</td>
<td>Pattern4</td>
<td>866</td>
<td>9549</td>
<td>301</td>
<td>332</td>
<td>1</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#11</td>
<td>Apache</td>
<td>2.2.9</td>
<td>Pattern5</td>
<td>250</td>
<td>8244</td>
<td>273</td>
<td>262</td>
<td>2</td>
<td>0</td>
<td>✓</td>
</tr>
</tbody>
</table>

TABLE III: The effectiveness of each filter in pruning suspects

<table>
<thead>
<tr>
<th>Bug #</th>
<th>App.</th>
<th># Suspects</th>
<th>Stage1</th>
<th>Stage2</th>
<th>Stage3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># Filtered by F1</td>
<td># Filtered by F2</td>
<td># Runs</td>
<td># Filtered by F3</td>
</tr>
<tr>
<td>#1</td>
<td>Pfscan</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>#2</td>
<td>Pfzip2</td>
<td>9</td>
<td>2</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>#3</td>
<td>Aget</td>
<td>31</td>
<td>12</td>
<td>17</td>
<td>36</td>
</tr>
<tr>
<td>#4</td>
<td>HTTrack</td>
<td>26</td>
<td>4</td>
<td>19</td>
<td>26</td>
</tr>
<tr>
<td>#5</td>
<td>Apache</td>
<td>342</td>
<td>127</td>
<td>185</td>
<td>275</td>
</tr>
<tr>
<td>#6</td>
<td>Apache</td>
<td>298</td>
<td>74</td>
<td>206</td>
<td>298</td>
</tr>
<tr>
<td>#7</td>
<td>MySQL</td>
<td>354</td>
<td>161</td>
<td>181</td>
<td>251</td>
</tr>
<tr>
<td>#8</td>
<td>Aget</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>#9</td>
<td>Apache</td>
<td>156</td>
<td>102</td>
<td>48</td>
<td>128</td>
</tr>
<tr>
<td>#10</td>
<td>MySQL</td>
<td>301</td>
<td>124</td>
<td>139</td>
<td>304</td>
</tr>
<tr>
<td>#11</td>
<td>Apache</td>
<td>273</td>
<td>96</td>
<td>147</td>
<td>229</td>
</tr>
</tbody>
</table>

E. Performance

Table IV shows the overhead of each component in our framework. Since both the profiler and the scheduler have to run on a real physical machine, we include the execution time of native (Column 2) for comparison purpose. We also include the execution time of Pin-base which is the execution time under PIN framework without any instrumentation. The overhead (shown via X) is normalized to the execution time of native environment. We find that the average overhead for the profiler is about 23.2X and the average overhead for the scheduler is about 26.2X. We realize that the triaging time is relatively long. However, we believe this is not a serious problem because all the profile runs and scheduler runs are independent to each other, making them easily parallelizable.

TABLE IV: Runtime overhead

<table>
<thead>
<tr>
<th>App.</th>
<th>Native</th>
<th>Pin-base</th>
<th>Profiler</th>
<th>Filler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pfscan</td>
<td>0.003s</td>
<td>0.024s (8.0X)</td>
<td>0.077s (25.6X)</td>
<td>0.077s (25.8X)</td>
</tr>
<tr>
<td>Pbzip2</td>
<td>0.075s</td>
<td>0.0817s (10.8X)</td>
<td>2.355s (31.4X)</td>
<td>2.437s (32.5X)</td>
</tr>
<tr>
<td>Aget</td>
<td>0.069s</td>
<td>0.627s (9.1X)</td>
<td>1.345s (19.5X)</td>
<td>1.807s (26.2X)</td>
</tr>
<tr>
<td>HTTrack</td>
<td>0.104s</td>
<td>1.424s (13.6X)</td>
<td>2.402s (23.1X)</td>
<td>3.016s (29.0X)</td>
</tr>
<tr>
<td>Apache</td>
<td>0.886s</td>
<td>4.075s (4.6X)</td>
<td>10.809s (12.2X)</td>
<td>13.112s (14.8X)</td>
</tr>
<tr>
<td>MySQL</td>
<td>0.405s</td>
<td>2.713s (6.7X)</td>
<td>11.259s (27.8X)</td>
<td>11.785s (29.1X)</td>
</tr>
<tr>
<td>Average</td>
<td>1X</td>
<td>8.8X</td>
<td>23.2X</td>
<td>26.2X</td>
</tr>
</tbody>
</table>

TABLE V: The bug isolation capability

<table>
<thead>
<tr>
<th>Bug #</th>
<th>App.</th>
<th>Version</th>
<th>Pattern Type</th>
<th># Profile Runs</th>
<th># Total patterns</th>
<th># Suspects</th>
<th># Schedules</th>
<th># After Filtering</th>
<th># False Positives</th>
<th>Isolated?</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Pfscan</td>
<td>1.0</td>
<td>Pattern1</td>
<td>76</td>
<td>273</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#2</td>
<td>Aget</td>
<td>0.4-dev</td>
<td>Pattern1</td>
<td>107</td>
<td>459</td>
<td>31</td>
<td>37</td>
<td>1</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#3</td>
<td>HTTrack</td>
<td>3.43.9</td>
<td>Pattern1</td>
<td>105</td>
<td>1102</td>
<td>26</td>
<td>29</td>
<td>1</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#4</td>
<td>Pbzip2</td>
<td>0.9.4</td>
<td>Pattern2</td>
<td>99</td>
<td>292</td>
<td>9</td>
<td>12</td>
<td>3</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#5</td>
<td>Apache</td>
<td>2.0.48</td>
<td>Pattern2</td>
<td>214</td>
<td>7011</td>
<td>342</td>
<td>290</td>
<td>3</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#6</td>
<td>MySQL</td>
<td>4.0.12</td>
<td>Pattern2</td>
<td>235</td>
<td>7974</td>
<td>298</td>
<td>121</td>
<td>3</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#7</td>
<td>Aget</td>
<td>0.4-dev</td>
<td>Pattern3</td>
<td>185</td>
<td>574</td>
<td>6</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#9</td>
<td>Apache</td>
<td>2.0.46</td>
<td>Pattern3</td>
<td>309</td>
<td>7347</td>
<td>156</td>
<td>130</td>
<td>3</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#10</td>
<td>MySQL</td>
<td>3.2.35b</td>
<td>Pattern4</td>
<td>866</td>
<td>9549</td>
<td>301</td>
<td>332</td>
<td>1</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>#11</td>
<td>Apache</td>
<td>2.2.9</td>
<td>Pattern5</td>
<td>250</td>
<td>8244</td>
<td>273</td>
<td>262</td>
<td>2</td>
<td>0</td>
<td>✓</td>
</tr>
</tbody>
</table>

VIII. RELATED WORK

A. Bug Isolation using Delta Traiging

Using delta triaging to isolate bugs is not new. Many bug isolation tools using delta triaging have been proposed in the past [11][18][23][26][28]. These tools usually share the same principle. Each run is labeled with either a success or a failure. To isolate a bug, these tools observe program behaviors in succeeded runs as well as in failed runs, compare their differences, and report the likely cause of the bug. A ranking mechanism is used to prioritize the report. Liblit et al. published a series of papers on cooperative bug isolation (CBI) for sequential programs [26][29]. CBI adopts sampling techniques to collect program predicates (e.g. branches, statements) from succeeded and failed runs, and infers those program points that are related to the failures using a statistical approach which is based on the rate of appearance of these predicates in succeeded and failed runs. Later, the technique used in CBI was extended by a few tools CCI [23][30], Falcon [11], Recon [18] to isolate bugs in concurrent programs.

The way in [11][23][30] they achieve that is essentially by introducing new kinds of program predicates that are inter-leaving and thread aware. False positives are a big concern in these tools. To achieve a reasonably low level of false positives, they assume that a notable number of distinct failed runs will manifest during triaging. However, without the help of some tools, simply re-running the program or using a basic random
technique, just like these tools do, is very unlikely to produce an acceptable number of failed runs because concurrency bugs usually manifest under very rare thread interleavings. Pinso does not have such an assumption. It uses a novel solution to reduce false positives using a filtration-oriented scheduler.

B. Concurrency Bug Detection

Concurrency bug detection tools can be cataloged into two classes: static tools and dynamic tools. Static bug detection tools [31][32][33][34] analyze programs statically and predict potential concurrency bugs. Most of the static bug detection tools produce large volume of false positives, preventing them from being widely used.

As well as static analysis, dynamic program analysis based on code instrumentation serve many important software engineering tasks such as profiling, debugging and testing. Dynamic bug detection tools [8][9][14][15] analyze a dynamic program execution and report bugs or potential bugs in that execution. One difference between our technique and many dynamic bug detection tools is that many of these tools report bugs according to only one execution, thus are more likely to report false warnings. In contrast, Pinso relies on the information retrieved from multiple executions, including both succeeded and failed runs, hence is more accurate and less likely to produce false warnings. Besides, each of these bug detection tools usually target only a specific class of concurrency bugs such as data races or deadlocks. In contrast, Pinso is based on interleaving patterns, thus is more general and covers a variety of types of concurrency bugs. AVIO [12] and DefUse [19], also collect interleaving related invariants from correct training runs, and use these invariants to detect anomalies in production runs. One obvious difference with Pinso is that our technique dramatically reduce false positives by leveraging our filtration-oriented scheduler, which is not mentioned in prior works.

C. Concurrency Testing

Concurrency testing is another way to deal with concurrency bugs. For a given program and input, a concurrency testing tool focuses on finding an execution (interleaving) that can trigger a bug in the program. Systematic testing tools [35] try to explore all possible thread interleavings. Random testing tools [21] pick thread interleavings randomly to avoid exploring the same interleavings in different test runs. Deterministic testing tools [13][36][37] first predict potential concurrency bugs in a program, and then try to actively expose each of them in real executions. Maple [13] is a coverage-driven concurrency testing tool based on interleaving idioms, which seeks to expose untested thread interleavings as much as possible based on idiom definitions. However, we have a very different goal. The goal of Pinso is to isolate and understand the root cause of a bug, while Maple is to find an execution that exposes a bug. Sometimes, the interleavings that Maple uses to expose a bug might be totally irrelevant to the bug. Reporting it will not be helpful to the programmer for understanding and debugging. In contrast, we compare the interleavings exposed in succeeded and failed runs, thus are more likely to isolate the cause.

Unlike those concurrency testing techniques whose goal
is to expose buggy executions, our goal is to isolate and understand the cause of bugs. Even if a buggy execution can be found and reproduced, it might still be difficult for a programmer to find and understand the cause of the bug because concurrency bugs typically involve non-trivial inter-thread communications. Our technique is one way towards solving this issue. Concurrency testing is complementary to our approach as it will help us reproduce a failed execution.

IX. CONCLUSION

Isolating concurrency bugs has been often an onerous task. In this paper, we presented a fault localization technique that could help programmers automatically identify the root causes of an execution failure. By producing correct executions for a bug triggering input and comparing them with the incorrect executions, we were able to precisely pinpoint the root causes of the concurrency bugs. A key contribution of our work is in the application of a filtration-oriented scheduler to produce alternative executions that helps us prune all the false positives. The evaluation showed that Pinso can effectively and efficiently isolate the root causes of the concurrency bugs, which are difficult to achieve using prior debugging tools based on suspiciousness.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful feedback. This research is supported by NSFC (No. 61272101), National Science and Technology Major Project (No. 2013ZX03002004), and National R&D Infrastructure and Facility Development Program (No. 2013FY111900).

REFERENCES