# Localization of Concurrency Bugs Using Shared Memory Access Pairs

Wenwen Wang[1,2], Zhenjiang Wang[1], Chenggang Wu[1]*, Pen-Chung Yew[3], Xipeng Shen[4], Xiang Yuan[1,2], Jianjun Li[1], Xiaobing Feng[1], Yong Guan[5]
[1]SKL Computer Architecture, ICT, CAS, [2]University of Chinese Academy of Sciences, [3]University of Minnesota at Twin-Cities, [4]College of William and Mary, [5]Capital Normal University
[1,2]{wangwenwen|wangzhenjiang|wucg|yuanxiang|lijianjun|fxb}@ict.ac.cn,
[3]yew@cs.umn.edu, [4]xshen@cs.wm.edu, [5]guanyong@mail.cnu.edu.cn

## ABSTRACT

We propose an effective approach to automatically localize buggy shared memory accesses that trigger concurrency bugs. Compared to existing approaches, our approach has two advantages. First, as long as enough successful runs of a concurrent program are collected, our approach can localize buggy shared memory accesses even with only one single failed run captured, as opposed to the requirement of capturing multiple failed runs in existing approaches. This is a significant advantage because it is more difficult to capture the elusive failed runs than the successful runs in practice. Second, our approach exhibits more precise bug localization results because it also captures buggy shared memory accesses in those failed runs that terminate prematurely, which are often neglected in existing approaches. Based on this proposed approach, we also implement a prototype, named LOCON. Evaluation results on 16 common concurrency bugs show that all buggy shared memory accesses that trigger these bugs can be precisely localized by LOCON with only one failed run captured.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

## Keywords

Concurrency Bug; Localization; Shared Memory Access Pair

## 1. INTRODUCTION

It is very difficult to debug concurrent programs due to their congenital non-determinism [36, 30, 23]. First, concurrency bugs are triggered only under particular thread interleavings. Second, even after a concurrency bug has been exposed in a failed program run, it still takes a tremendous amount of time and effort to localize and fix the bug [25]. Previous work [19] shows that it takes nearly 73 days on average to fix a concurrency bug, and a survey in Microsoft [9] shows that over half of the respondents suffer from concurrency bugs at least once a month.

Concurrency bugs exhibit most frequently as *order violations* (OV) and *atomicity violations* (AV) (including single-variable (SAV) and multi-variable (MAV)), accounting for 97% of common concurrency bugs [19]. Some of them are caused by data races in programs, while others occur even in data-race-free programs. OV is a violation of the desired happen-before order [19] between two accesses to the same shared variable. AV is a violation of a code region's *atomicity* (also known as *serializability*) [19, 28], which occurs when the code region is unintentionally interleaved with memory accesses from another thread.

A large amount of work [7, 32, 33, 3, 27, 20, 28] has been proposed to detect concurrency bugs, e.g. data races and AV bugs, by leveraging static and/or dynamic program analysis techniques. Unfortunately, due to the large amount of false positives, programmers still have to sift through a large volume of results to localize the bugs. Prior work [26] finds that 90% of the detected data races are actually benign.

Meanwhile, some other work [23, 37, 6] tries to expose concurrency bugs by triggering them via systematic or randomly-orchestrated thread interleavings. But, even after strenuous testing, unexposed bugs could still be hidden in programs, and their manifestation could be catastrophic [15, 8, 31].

During testing concurrent programs, developers have to manually localize a concurrency bug after it incidentally shows up in a failed run. However, it is very time and effort consuming to pinpoint the exact cause or source of a concurrency bug [19, 30, 12], i.e. identifying the exact memory accesses that cause the bug, due to their large volume.

To relieve the debugging burden from programmers, some approaches [30, 29, 12, 5] try to analyze the logged data and automatically localize the exact causes of the bugs. The aim of these approaches is *not to trigger or detect the bugs*, but to *localize the causes of the bugs* after they show up in failed runs, a so-called *fault localization problem*. They collect program predicates at runtime in multiple runs, which may complete successfully or fail. Such predicates can be branch outcomes [12], bug patterns [30, 29], or cache coherence events [5], which exhibit different behavior in successful and failed runs. Statistical models such as the *Jaccard index* [30, 29], the harmonic mean of the occurrence frequency, or well-defined scores [12, 5] are then used to calculate the scores for those predicates. Predicates with higher values are more likely to be the true causes.

*To whom correspondence should be addressed.


*ASE'14,* September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642972.

The score of each predicate is calculated based on the relative numbers of the failed and successful runs in which this predicate shows up. To allow the true causes to have distinctly higher scores, a sufficient number of successful and failed runs are required. However, it is not easy to capture a large number of failed runs, e.g. it took about 22 hours (tens of thousands of successful runs) for a real-world SAV bug in the Apache Server to manifest in a failed run [28]. More likely, a programmer will capture one failed run after many more successful runs in reality.

In this paper, we propose a more practical approach to address the above limitation. Our approach, through a systematic examination of the common buggy shared memory accesses, identifies a list of shared memory access pairs that behave distinctively in failed and successful runs. Three scenarios that summarize the manifestation differences of the buggy memory access pairs triggering OV or AV (also SAV or MAV) bugs in successful and failed runs are thus identified. This inspires us to develop three test procedures, each of which checks the distinctive memory access pairs in one of the three scenarios. Experiments on 16 common concurrency bugs show that our approach can localize these bugs even with only one failed run captured and has significantly fewer false positives than the state of art methods.

This paper makes the following contributions:

▶ First, we identify an exhaustive list of scenarios that indicate the manifestation differences of the buggy access pairs triggering OV or AV bugs in successful and failed runs. To the best of our knowledge, this is the first attempt to compile all possible scenarios that cover OV and AV bugs.

▶ Second, we propose 3 comprehensive test procedures to automatically localize buggy memory access pairs by matching them to all identified scenarios. The test procedures are designed elaborately to eliminate potential false positives.

▶ Third, we implement a prototype, named LOCON, based on the proposed test procedures, and evaluate it on 16 common concurrency bugs. The results show that LOCON is capable of localizing these bugs precisely with only one failed run captured, i.e. it is not required to capture multiple failed runs to pinpoint a bug as in existing approaches.

The rest of this paper is organized as follows. Section 2 describes the main idea of this paper. Section 3 and 4 respectively illustrate scenarios and test procedures. Section 5 and 6 present and evaluate LOCON. Section 7 discusses related work. Section 8 concludes the paper.

# 2. OVERVIEW

We firstly define some notations and then leverage an example to motivate the main idea of our approach.

## 2.1 Notations

A thread is denoted as $T_i$, where $i = 1, 2, ...$, indicates a unique thread id. $M_x^a$ represents an access to a shared variable $x$, where $M$ indicates the type of the access, either $R$ ($R$ead) or $W$ ($W$rite), $a$ is the instruction or statement in the source text that issues the access.

A *shared memory access pair*, or an access pair for short, represents an immediate dependency ($R$-$W$, $W$-$R$, or $W$-$W$) between two memory accesses in two threads: $M_x^a \rightarrow N_x^b$, where $x$ is the shared variable accessed and *at least* one of $M$ and $N$ is a write. The access pair indicates that $M_x^a$ happens before $N_x^b$. $M_x^a$ refers to the *head*, and $N_x^b$ refers to the *tail*, of the access pair. Besides, $M_x^a \rightarrow N_x^b$ and $N_x^b \rightarrow M_x^a$ are respectively called the *reversed* access pair of each other.

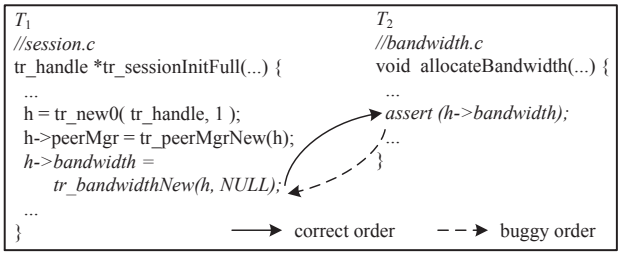



**Figure 1:** An OV bug in Transmission-1.42.

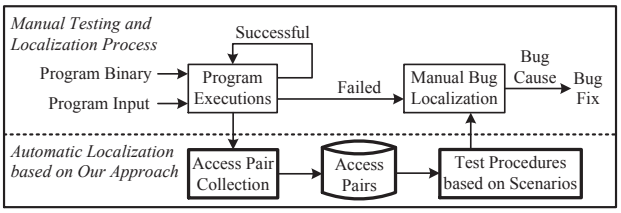



**Figure 2:** Automatic bug localization based on our approach.

A *successful* run is a program execution that produces the expected output of the programmer. Otherwise, the execution is considered as a *failed* run. In this paper, we only handle with failed runs caused by OV and AV bugs. Also, failed and successful runs have the *same* program input.

Typically, the manifestation of concurrency bugs shows up in the form of wrong values being read from the violated shared variables. These *reads* are referred to as $ETPs$ ($E$rror $T$riggering $P$oint$s$) of the concurrency bugs in this paper. After the occurrence of an $ETP$, the program may continue to run but produce incorrect results, or may be terminated prematurely due to faults. Therefore, the tail of the buggy access pair may not show up after the $ETP$. Such buggy access pair is often not included in the final report of most existing approaches because its tail is not actually executed. Our approach pays special attention to these buggy access pairs and results in more precise localization results.

## 2.2 A Motivating Example

We use the example illustrated in Figure 1 to describe the main idea of our approach. This is a real-world OV bug from Transmission-1.42 [34], which is a multi-threaded BitTorrent download client. In this example, *h->bandwidth* is a shared variable initialized by the statement *h->bandwidth = tr_bandwidthNew(h, NULL)* in *session.c*, i.e. $W_{h->bandwidth}$ (the statement is omitted for simplicity). The statement *assert(h->bandwidth)* in *bandwidth.c* reads *h->bandwidth*, i.e. $R_{h->bandwidth}$, which ought to occur after $W_{h->bandwidth}$. Unfortunately, due to the lack of proper synchronization, $R_{h->bandwidth}$ is likely to be executed before $W_{h->bandwidth}$ and reads an uninitialized value, causing an assertion failure. Thus, $R_{h->bandwidth}$ is the $ETP$ of this bug.

The bug is not often triggered even with the bug-triggering input. Actually, we cannot come across this bug until tens of thousands of successful runs have been tested. After studied the program, we found that the function *allocateBandwidth* is called from an event callback function that is often called about 500 milliseconds after *h->peerMgr* is created. This is why the bug occurs rarely in practice. But, if the execution environment is changed, e.g. the underlying schedule policy, the bug may manifest and cause the program to crash.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| | ID | Illegal Interleavings | Failed Runs | Scenario Type | Successful Runs Run-1 | Successful Runs Run-2 | Description |
| OV | 1 | $W_x^a \rightarrow \boldsymbol{R_x^b}$ | $W_x^a \rightarrow \boldsymbol{R_x^b}$ | I | $\boldsymbol{R_x^b} \rightarrow W_x^a$ | | Unexpected value is read due to the premature write. |
| | | | | | | | |
| | 2 | $\boldsymbol{R_x^a} \rightarrow W_x^b$ | $\boldsymbol{R_x^a} \rightarrow W_x^b$ | I | $W_x^b \rightarrow \boldsymbol{R_x^a}$ | | Unexpected value is read due to the hysteretic write. |
| | | | $\varnothing$ | II | | | Buggy access pair is missing due to the $ETP$. |
| | 3 | $W_x^a \rightarrow W_x^b$ | $W_x^a \rightarrow W_x^b$ | I | $W_x^b \rightarrow W_x^a$ | | The first write is covered by the second. |
| | | | | | | | |
| SAV | 4 | $W_x^a \rightarrow W_x^b$, $\boldsymbol{R_x^c} \rightarrow W_x^b$ | $W_x^a \rightarrow W_x^b$, $W_x^b \rightarrow \boldsymbol{R_x^c}$ | I | $\boldsymbol{R_x^c} \rightarrow W_x^b$ | $W_x^b \rightarrow W_x^a$ | The first write is covered by the second. |
| | | | | | | | |
| | 5 | $R_x^a \rightarrow W_x^b$, $W_x^c \rightarrow W_x^b$ | $R_x^a \rightarrow W_x^b$, $W_x^b \rightarrow W_x^c$ | I | $W_x^c \rightarrow W_x^b$ | $W_x^b \rightarrow R_x^a$ | The first write is covered by the second. |
| | | | | | | | |
| | 6 | $R_x^a \rightarrow W_x^b$, $\boldsymbol{R_x^c} \rightarrow W_x^b$ | $R_x^a \rightarrow W_x^b$, $W_x^b \rightarrow \boldsymbol{R_x^c}$ | I | $\boldsymbol{R_x^c} \rightarrow W_x^b$ | $W_x^b \rightarrow R_x^a$ | Inconsistent values are read due to improper write. |
| | | | | | | | |
| | 7 | $W_x^a \rightarrow \boldsymbol{R_x^b}$, $W_x^c \rightarrow \boldsymbol{R_x^b}$ | $W_x^a \rightarrow \boldsymbol{R_x^b}$, $\boldsymbol{R_x^b} \rightarrow W_x^c$ | I | $W_x^c \rightarrow \boldsymbol{R_x^b}$ | $\boldsymbol{R_x^b} \rightarrow W_x^a$ | Unexpected intermediate value is read between the writes. |
| | | | $W_x^a \rightarrow \boldsymbol{R_x^b}$ | I | | | Unexpected intermediate value is read after the first write. |
| MAV | 8 | $R_x^a \rightarrow W_x^b$, $W_y^d \rightarrow \boldsymbol{R_y^c}$ | $R_x^a \rightarrow W_x^b$, $\boldsymbol{R_y^c} \rightarrow W_y^d$ | III | $R_x^a \rightarrow W_x^b$ | $W_x^b \rightarrow R_x^a$ | Inconsistent results of multiple variables due to improper accesses. |
| | | | $W_x^b \rightarrow R_x^a$, $W_y^d \rightarrow \boldsymbol{R_y^c}$ | III | $W_y^d \rightarrow \boldsymbol{R_y^c}$ | $\boldsymbol{R_y^c} \rightarrow W_y^d$ | Inconsistent results of multiple variables due to improper accesses. |
| | | | $R_x^a \rightarrow W_x^b$ | II | | | Buggy access pairs are partially missing due to the $ETP$. |
| | 9 | $R_x^a \rightarrow W_y^d$, $R_y^b \rightarrow W_x^c$ | $R_x^a \rightarrow W_x^c$, $R_y^b \rightarrow W_y^d$ | III | $R_x^a \rightarrow W_x^c$, $W_y^d \rightarrow R_y^b$ | $R_y^b \rightarrow W_y^d$, $W_x^c \rightarrow R_x^a$ | Inconsistent results of multiple variables due to improper accesses. |
| | | | | | | | |
| | | | | | | | |
| | 10 | $W_x^a \rightarrow W_y^d$, $R_y^b \rightarrow \boldsymbol{R_y^c}$ | $W_x^a \rightarrow R_x^b$, $\boldsymbol{R_y^c} \rightarrow W_y^d$ | III | $W_x^a \rightarrow R_x^b$ | $R_x^b \rightarrow W_x^a$ | Inconsistent values of multiple variables are read. |
| | | | $R_x^b \rightarrow W_x^a$, $W_y^d \rightarrow \boldsymbol{R_y^c}$ | III | $W_y^d \rightarrow \boldsymbol{R_y^c}$ | $\boldsymbol{R_y^c} \rightarrow W_y^d$ | Inconsistent values of multiple variables are read. |
| | | | $W_x^a \rightarrow R_x^b$ | II | | | Buggy access pairs are partially missing due to the $ETP$. |
| | 11 | $W_x^a \rightarrow W_y^d$, $R_y^b \rightarrow \boldsymbol{R_y^c}$ | $W_x^a \rightarrow W_x^b$, $\boldsymbol{R_y^c} \rightarrow W_y^d$ | III | $W_x^a \rightarrow W_x^b$ | $W_x^b \rightarrow W_x^a$ | Inconsistent results of multiple variables due to improper accesses. |
| | | | $W_x^b \rightarrow W_x^a$, $W_y^d \rightarrow \boldsymbol{R_y^c}$ | III | $W_y^d \rightarrow \boldsymbol{R_y^c}$ | $\boldsymbol{R_y^c} \rightarrow W_y^d$ | Inconsistent results of multiple variables due to improper writes. |
| | | | $W_x^a \rightarrow W_x^b$ | II | | | Buggy access pairs are partially missing due to the $ETP$. |
| | 12 | $W_x^a \rightarrow W_y^b$, $W_y^d \rightarrow W_y^c$ | $W_x^a \rightarrow W_x^b$, $W_y^c \rightarrow W_y^d$ | III | $W_x^a \rightarrow W_x^b$ | $W_x^b \rightarrow W_x^a$ | Inconsistent results of multiple variables due to improper writes. |
| | | | $W_x^b \rightarrow W_x^a$, $W_y^d \rightarrow W_y^c$ | III | $W_y^d \rightarrow W_y^c$ | $W_y^c \rightarrow W_y^d$ | Inconsistent results of multiple variables due to improper writes. |
| | | | | | | | |
| | 13 | $W_x^a \rightarrow R_y^b$, $W_y^d \rightarrow \boldsymbol{R_x^c}$ | $W_x^a \rightarrow \boldsymbol{R_x^c}$, $R_y^b \rightarrow W_y^d$ | III | $W_x^a \rightarrow \boldsymbol{R_x^c}$ | $R_y^b \rightarrow W_y^d$ | Inconsistent results of multiple variables due to improper reads. |
| | | | $W_x^a \rightarrow \boldsymbol{R_x^c}$ | II | $W_y^d \rightarrow R_y^b$ | $\boldsymbol{R_x^c} \rightarrow W_x^a$ | Buggy access pairs are partially missing due to the $ETP$. |
| | | | | | | | |
| | 14 | $W_x^a \rightarrow R_y^b$, $W_y^d \rightarrow W_x^c$ | $W_x^a \rightarrow W_x^c$, $R_y^b \rightarrow W_y^d$ | III | $W_x^a \rightarrow W_x^c$, $W_y^d \rightarrow R_y^b$ | $R_y^b \rightarrow W_y^d$, $W_x^c \rightarrow W_x^a$ | Inconsistent results of multiple variables due to improper accesses. |
| | | | | | | | |
| | | | | | | | |
| | 15 | $W_x^a \rightarrow W_y^b$, $W_y^d \rightarrow W_x^c$ | $W_x^a \rightarrow W_x^c$, $W_y^b \rightarrow W_y^d$ | III | $W_x^a \rightarrow W_x^c$, $W_y^d \rightarrow W_y^b$ | $W_y^b \rightarrow W_y^d$, $W_x^c \rightarrow W_x^a$ | Inconsistent results of multiple variables due to improper writes. |
| | | | | | | | |
| | | | | | | | |

**Table 1:** Different scenarios for OV and AV bugs. "$\boldsymbol{R}$": possible $ETP$ triggered by a read, "$\varnothing$": there is no access pair in the failed run.

Figure 2 shows a common process of testing and debugging a concurrent program, which iteratively executes the program binary with a given input. If the program run is successful, the testing process iterates again. Otherwise, if failed, programmers have to manually find the bug, which is often time consuming. Our approach collects access pairs in program runs, and when a failed run is encountered, *test procedures* based on the manifestation *scenarios* analyze the collected access pairs and report buggy access pairs that trigger the bug. For the bug in Figure 1, if the assertion happens to fail after some successful runs, our approach will report $R_{h->bandwidth} \rightarrow W_{h->bandwidth}$ as the buggy access pair after applying Test Procedure II (see Section 4.2).

Note, the program crashes after the execution of the $ETP$ of the bug, i.e. $R_{h->bandwidth}$, and $W_{h->bandwidth}$ is not actually executed in failed runs. Most existing pattern-based approaches fail to localize this bug, since they only consider access patterns gathered in failed runs [21, 30, 29]. For instance, Falcon [30] only ranks access patterns collected in failed runs and therefore may fail to localize this bug. Instead, our approach can successfully report the buggy access pair as the bug cause. Our advantage comes from the exhaustive list of manifestation scenarios presented in Section 3, which is the basis of our approach.

# 3. MANIFESTATION SCENARIOS

We present the scenarios based on two hypotheses [37]. First, most of concurrency bugs can be triggered with a small number of thread preemptions, i.e. the well-known small scope hypothesis [10, 23]. We leverage this observation to bound the number of access pairs in scenarios to two. Second, a majority of concurrency bugs can be triggered when the buggy access pairs show up in program runs, regardless of the data value of the shared variables involved in the access pairs, i.e. the value-independence hypothesis [37]. Besides, we also assume that properly synchronized concurrent programs adhere to the data-race-free 0 model [2]. With this model, the underlying hardware appears to be *sequential consistent* even though it may be implemented using a weaker consistency model. Based on these assumptions, Table 1 demonstrates a exhaustive list of scenarios for OV and AV bugs based on two threads $T_1$ and $T_2$ (omitted due to space limitation) and at most two shared variables $x$ and $y$. We explain each column as follows.

The column 1 shows bug types. The column 2 gives each type of bug an ID number. For OV bugs, there are only three possible violations of shared memory access orders: $R$-$W$, $W$-$R$, and $W$-$W$. Thus, there are three types of OV bugs (ID-1 $\sim$ ID-3). For SAV bugs, each one involves three

shared memory accesses [20, 30]. The combinatorial number of $R$ and $W$ for the three accesses is $2^3$, but only four of them are unserializable, i.e. AV bugs. Thus there are four types of SAV bugs (ID-4 $\sim$ ID-7). Similarly, we can conclude that there are eight types of MAV bugs (ID-8 $\sim$ ID-15).

The column 3 lists one possible illegal interleaving for each type of bug, where the accesses in the left and right sides are respectively executed by $T_1$ and $T_2$, the accesses in the same thread are executed from top to bottom, and the arrow between two accesses denotes their happen-before order, i.e. the source happens before the target. The bug is triggered under the illegal interleaving, leading to a failed run in the column 4. For example, ID-1 is an OV bug, where the intended happen-before order between the two accesses is $R_x^b$ happens before $W_x^a$. However, the illegal interleaving $W_x^a$-$R_x^b$ violates this order and causes the failed run in the column 4, where $W_x^a \rightarrow R_x^b$ shows up. Some types of bugs may have more than one possible illegal interleaving. For example, ID-10 is an MAV bug, where shared variables $x$ and $y$ should be written ($W_x^a$ and $W_y^d$) and read ($R_x^b$ and $R_y^c$) atomically. There are two possible illegal interleavings, i.e. $W_x^a$-$R_x^b$-$R_y^c$-$W_y^d$ and $R_x^b$-$W_x^a$-$W_y^d$-$R_y^c$, which respectively violate the atomicity between ($W_x^a$ and $W_y^d$) and ($R_x^b$ and $R_y^c$). We only show the first one in the column 3 due to space limitation, but list all possible buggy access pairs that show up under each illegal interleaving in the column 4, i.e. the first two cases in the column 4. The third case $W_x^a \rightarrow R_x^b$ is listed with a description of the cause in the column 8, i.e. one buggy access pair is missing due to the $ETP$ caused by $R_y^c$. Note, $W_x^a$-$R_x^b$-$W_y^d$-$R_y^c$ and $R_x^b$-$W_x^a$-$R_y^c$-$W_y^d$ are not illegal interleavings for ID-10, because the execution results of these two interleavings are respectively equivalent to $W_x^a$-$W_y^d$-$R_x^b$-$R_y^c$ and $R_x^b$-$R_y^c$-$W_x^a$-$W_y^d$, which means they are serializable.

The illegal interleavings listed in the column 3 represents all possible anomalous interleavings triggering OV and AV (including SAV and MAV) bugs. These interleaving patterns are more comprehensive and general than the problematic access patterns used in prior work for specific classes of concurrency bugs [20, 30, 29, 28, 35].

The columns 6 and 7 illustrate access pairs manifested in possible successful runs. For each OV bug, there is only one possible case in successful run, where the desired happen-before order of the involved accesses is followed. For each SAV or MAV bug, there are two possible cases, and each of them is a serialized execution of the involved atomic regions and the atomicity semantics are satisfied.

As shown in the column 5, there are three possible manifestation scenarios, denoted as Scenario I, II, and III. The intuition behind these scenarios is to summarize the manifestation differences of the buggy access pairs in the failed and successful runs. These differences serve as the basis for our test procedures for concurrency bug localization. Actually, developers also pay more attention to abnormal program behaviors in failed runs compared with successful runs when debugging a program. We discuss each scenario as follows.

## 3.1 Scenario I

In this scenario, buggy access pairs only occur in failed runs, but *not* in successful runs. Take ID-4, an SAV bug, as an example. In the two successful runs (the columns 6 and 7), the atomicity between $W_x^a$ and $R_x^c$ is enforced, and two access pairs respectively show up in each successful run: $R_x^c \rightarrow W_x^b$ and $W_x^b \rightarrow W_x^a$. But, in the failed run (the column 4), the atomicity is violated under the illegal interleaving: $W_x^a$-$W_x^b$-$R_x^c$. Comparing with the former two access pairs, we can easily see that buggy access pairs, i.e. $W_x^a \rightarrow W_x^b$ and $W_x^b \rightarrow R_x^c$, only occur in the failed run.

## 3.2 Scenario II

Buggy access pairs manifest in *neither* successful *nor* failed runs in this scenario. Take ID-2, an OV bug, as an example. The intended happen-before order between $W_x^b$ and $R_x^a$ is enforced by $W_x^b \rightarrow R_x^a$ in the successful run, shown in the column 6. Because $R_x^a$ is an $ETP$ that may terminate program execution due to the fault triggered by this violation, there are two possible cases in the failed runs. One is to have the buggy access pair $R_x^a \rightarrow W_x^b$, which does not present in the successful runs, but another does not form any access pair because $W_x^b$ is not executed after $R_x^a$ triggers an $ETP$. We use an $\varnothing$ to represent this case in the column 4.

Another example of this scenario is ID-10, an MAV bug, in which $R_y^c$ is a possible $ETP$. There are three possible cases in failed runs. The first two will be discussed in Scenario III. In the third one, there exists only one buggy access pair $W_x^a \rightarrow R_x^b$, and another buggy access pair $R_y^c \rightarrow W_y^d$ is missing because $W_y^d$ is not executed after the $ETP$ triggered by $R_y^c$. In fact, this bug can only be triggered when the two buggy access pairs show up together in the same run (i.e. it is an MAV bug involving two shared variables, $x$ and $y$). We call such two buggy access pairs *coupled buggy access pairs*. If one of them is missing in a failed run, we call them *partially-missing* coupled buggy access pairs.

Note, in the second case of the failed runs in ID-7, the bug can be triggered by $W_x^a \rightarrow R_x^b$ alone without $R_x^b \rightarrow W_x^c$ as in the first case. However, in the third case of the failed runs in ID-10, the bug cannot be triggered by $W_x^a \rightarrow R_x^b$ alone. It needs an illegal interleaving with $y$. This is why the former is in Scenario I while the latter in Scenario II.

## 3.3 Scenario III

Buggy access pairs show up in *both* failed and successful runs in this scenario. Take ID-10 again as an example. As mentioned above, there are three possible cases in its failed runs. In the first one (the second one is similar), there is a coupled buggy access pairs: $W_x^a \rightarrow R_x^b$ and $R_y^c \rightarrow W_y^d$, each of which also shows up in successful runs (the columns 6 and 7). But, they do not show up together in any successful run.

Although Table 1 only lists MAV bugs triggered by only two access pairs, other unlisted MAV bugs can also fall into this scenario. Suppose an MAV bug that can be triggered by "at least" $n$ access pairs, where $n \geq 2$: $h_1 \rightarrow t_1$, $h_2 \rightarrow t_2$, ..., $h_n \rightarrow t_n$, which may involve more than two shared variables. If the happen-before order between $h_i$ and $t_i$, where $1 \leq i \leq n$, is reversed in a program run, the bug will not be triggered, and the following access pairs will be collected: $h_1 \rightarrow t_1$, $h_2 \rightarrow t_2$, ..., $t_i \rightarrow h_i$, ..., $h_n \rightarrow t_n$. Therefore, there exist at least $n$ successful runs, each of which only reverses one of $n$ access pairs. Thus, the scenario of the $n$ buggy access pairs satisfies the conditions of Scenario III: each one of them shows up in both failed and successful runs, but they never show up together in any successful run.

As previously discussed in the small scope hypothesis, concurrency bugs triggered by more than two access pairs are very rare in real practice [19, 37]. Hence, even if it is very simple, we will not extend our test procedures to localize such bugs, because it will increase the localization over-

head significantly without much benefit. To the best of our knowledge, none of the existing concurrency bug localization methods considered such bugs due to their complicated trigger conditions and rarity in real practice.

# 4. TEST PROCEDURES

Assume $P$ is a concurrent program and $I$ is a concurrency-bug-triggering input of $P$. $PRun$ and $FRun$ respectively denote a set of access pairs that can be collected in a successful and failed run of $P$ under $I$. $PSet$ is the set of $PRuns$. Given $PSet$ and an $FRun$, the fault localization problem is to identify buggy access pairs triggering the concurrency bug in $FRun$. In practice, it is much easier to come across $PRun$ than $FRun$ due to the special trigger conditions of concurrency bugs. Moreover, after a few number of $PRuns$ encountered, almost all of possible access pairs in $P$ under $I$ can be collected. For simplicity of explanation, the following description first assumes that we have collected all possible access pairs that can appear in $PRuns$ in the $PSet$. In Section 4.4, we will show that the assumption is not required in practice. Besides, in Section 6.3, we will also study the sensitivity of localization results on the number of $PRuns$.

To solve the localization problem, we design 3 test procedures to localize buggy access pairs manifested in scenarios discussed in Section 3. We describe them as follows.

## 4.1 Test Procedure I

The first test procedure aims to uncover buggy access pairs manifested with Scenario I, where buggy access pairs show up in $FRun$, but not in $PRuns$. Algorithm 1 shows the details of this test procedure. In `Step1` (lines $1 \sim 6$), we collect access pairs that show up in $FRun$, but not in $PRuns$, denoted as $apSet$. However, access pairs in $apSet$ are not all buggy. This is one major cause of false positives in other existing approaches. We identify those false positives in the following steps and filter them out.

One type of false positives is caused by the *predictable* access pairs. An access pair is predictable by another, if and only if the presence of the former can be inferred by the presence of the latter, but not vice versa. This means if the latter presents in a program run, the former will be also found in this run. Figure 3 shows an OV bug, which is triggered by $W_y^3 \rightarrow R_y^2$ (i.e. $T_1$ dereferences a null pointer). In this example, $W_x^1 \rightarrow R_x^4$ is predictable by $R_y^2 \rightarrow W_y^3$, and $W_y^3 \rightarrow R_y^2$ is predictable by $R_x^4 \rightarrow W_x^1$. If we process $FRun2$ using Algorithm 1 (Note: we can process $FRun1$ and $FRun2$ in Figure 3 independently as mentioned in Section 1), we can obtain $W_y^3 \rightarrow R_y^2$ and $R_x^4 \rightarrow W_x^1$ in $apSet$ after `Step1`, where $R_x^4 \rightarrow W_x^1$ is a false positive because the bug is not triggered by it, but by $W_y^3 \rightarrow R_y^2$.

We filter out this type of false positives in `Step2` of Algorithm 1 (lines $7 \sim 10$) by identifying predictable access pairs in $apSet$. Suppose two access pairs $AP_1$ and $AP_2$ in $apSet$, where $AP_1$ is predictable by $AP_2$. Then $AP_2$ is identified as a false positive. The reason is as follows. Assume $AP_2$ is not a false positive but the buggy access pair that triggers the bug. Then its *reversed* access pair, denoted as $RAP_2$, will not be buggy according to OV and SAV bugs shown in Table 1. Thus, there exists a $PRun$ that contains both $AP_1$ and $RAP_2$. This violates the precondition that $AP_1$ is in $apSet$, i.e. $AP_1$ does not show up in any $PRun$.

Another type of false positives is caused by the execution of a rarely executed code region after an $ETP$ in the failed run, e.g. an embedded debugging routine in the program. In this case, all access pairs in this code region will appear only in $FRun$ but not in $PRuns$. They are certainly not the causes of the concurrency bug. To eliminate such false positives, access pairs in $apSet$ are sorted according to their occurrence order in `Step3` of Algorithm 1 (line 11) as such false positives always occur *after* buggy access pairs. The reason is that the formation of these false positive access pairs results from the triggered concurrency bugs.

**Algorithm 1:** Test Procedure I

**Input**: failed run: $FRun$; successful run set: $PSet$
**Output**: buggy access pairs manifested with Scenario I

```
   // Step1: get access pairs that show up only in FRun
1  apSet ← FRun;
2  for access pair AP ∈ apSet do
3  |  for successful run PRun ∈ PSet do
4  |  |  if AP ∈ PRun then apSet ← apSet − {AP}; break;
5  |  end
6  end
   // Step2: filter out predictable access pairs
7  for two access pairs APi, APj ∈ apSet do
8  |  if APi is predictable by APj then apSet ← apSet − {APj};
9  |  else if APj is predictable by APi then
   |  apSet ← apSet − {APi};
10 end
   // Step3: sort access pairs according to occurrence order
11 apList ← SortAccessPairs(apSet);
12 return apList;
```

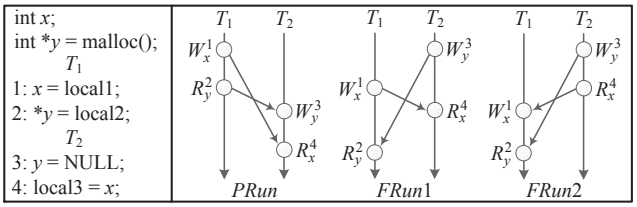



**Figure 3:** False positive introduced by predictable access pairs.

## 4.2 Test Procedure II

Test Procedure II is to localize buggy access pairs manifested with Scenario II, where buggy access pairs show up in neither $PRuns$ nor $FRun$ because they are missing or partially-missing in $FRun$ due to $ETPs$. Thus, they cannot be directly localized as those manifested with Scenario I. We localize these buggy access pairs by observing the manifestation of their *reversed* access pairs in $PRuns$ and $FRun$.

For missing buggy access pairs, their reversed access pairs will always show up in $PRuns$, but not in $FRun$. For example, in the second case of failed runs of ID-2 in Table 1, $R_x^a \rightarrow W_x^b$ is missing in both successful and failed runs. But its reversed access pair, i.e. $W_x^b \rightarrow R_x^a$, always show up in successful runs due to the intended happen-before order between $W_x^b$ and $R_x^a$.

For partially-missing coupled buggy access pairs, the reversed access pairs of the missing buggy access pairs show up in $PRuns$ together with their counterparts (which are also buggy access pairs) in the coupled buggy access pairs. Take the third case of failed runs of ID-10 in Table 1 as an example. $W_x^a \rightarrow R_x^b$ and $R_y^c \rightarrow W_y^d$ are partially-missing coupled buggy access pairs, where $R_y^c \rightarrow W_y^d$ is missing. The reversed access pair of $R_y^c \rightarrow W_y^d$, i.e. $W_y^d \rightarrow R_y^c$, which is also missing in the failed run, always shows up together with its counterpart, i.e. $W_x^a \rightarrow R_x^b$, in successful runs due to the required atomicity of accessing to both $x$ and $y$.

We leverage above observations on reversed access pairs to localize buggy access pairs manifested with Scenario II.

**Algorithm 2:** Test Procedure II

**Input**: failed run: $FRun$; successful run set: $PSet$
**Output**: buggy access pairs manifested with Scenario II

```
   // Step1.1: get access pairs that show up in all PRuns
1  apSet ← ∅; capSet ← ∅;
2  for successful run PRun ∈ PSet do
3  |  if apSet = ∅ then apSet ← PRun;
4  |  else
5  |  |  for access pair AP ∈ apSet do
6  |  |  |  if AP ∉ PRun then apSet ← apSet − {AP};
7  |  |  end
8  |  end
9  end
   // Step1.2: get coupled access pairs that always show up
      together in PRuns
10 for successful run PRun_m ∈ PSet do
11 |  for AP_i, AP_j ∈ PRun_m and AP_i, AP_j ∉ apSet and
   |  AP_i, AP_j access different shared variables do
12 |  |  flag ← true;
13 |  |  for successful run PRun_n ∈ PSet do
14 |  |  |  if (AP_i ∈ PRun_n and AP_j ∉ PRun_n) or
   |  |  |  (AP_i ∉ PRun_n and AP_j ∈ PRun_n) then flag ←
   |  |  |  false; break;
15 |  |  end
16 |  |  if flag = true then capSet ← capSet + {(AP_i, AP_j)};
17 |  end
18 end
   // Step2.1: check access pairs from Step1.1 in FRun
19 rapSet ← ∅;
20 for access pair AP ∈ apSet do
21 |  if AP ∉ FRun then
   |  rapSet ← rapSet + {getReversedAP(AP)};
22 end
   // Step2.2: check coupled access pairs from Step1.2 in
      FRun
23 for coupled access pairs (AP_i, AP_j) ∈ capSet do
24 |  if AP_i ∈ FRun and AP_j ∉ FRun then
25 |  |  rapSet ← rapSet + {getReversedAP(AP_j)}
26 |  else if AP_i ∉ FRun and AP_j ∈ FRun then
27 |  |  rapSet ← rapSet + {getReversedAP(AP_i)}
28 |  end
29 end
   // Step3: sort access pairs according to occurrence order
30 rapList ← SortOriginalAccessPairs(rapSet);
31 return rapList;
```

Algorithm 2 shows the details. In `Step1.1` (lines 1 $\sim$ 9), we collect access pairs that show up in all $PRuns$. In `Step2.1` (lines 19 $\sim$ 22), we check each access pair collected in `Step1.1` whether it show up in $FRun$. If not, its reversed access pair (returned by `getReversedAP`()) is considered as one of the missing buggy access pairs. Similarly, `Step1.2` (lines 10 $\sim$ 18) and `Step2.2` (lines 23 $\sim$ 29) can localize partially-missing coupled buggy access pairs in $FRun$.

However, non-buggy access pairs may also be missing in $FRun$ due to $ETPs$. For example, code regions always executed in successful runs may not be executed in the failed run after an $ETP$ occurs, so reversed access pairs of the access pairs in these code regions can also be identified as buggy after the above steps. To eliminate such false positives, we also sort the original access pairs of those obtained by above two steps according to their occurrence order (`Step3` in line 30), as the similar reason to that in Test Procedure I.

Now, apply this test procedure to the example in Figure 1. After `Step1.1`, we can find that $W_{h->bandwidth} \rightarrow R_{h->bandwidth}$ shows up in all $PRuns$, which means the read access to *h->bandwidth* should happen after the initialization. In `Step2.1`, we can further find that this access pair do not show up in $FRun$. Thus, its reversed access pair, $R_{h->bandwidth} \rightarrow W_{h->bandwidth}$, is reported as the buggy access pair after sorting in `step3`.

**Algorithm 3:** Test Procedure III

**Input**: failed run: $FRun$; successful run set: $PSet$
**Output**: buggy access pairs manifested with Scenario III

```
   // Step1: get access pairs that show up in PRuns and
      FRun
1  apSet ← ∅;
2  for access pair AP ∈ FRun do
3  |  for successful run PRun ∈ PSet do
4  |  |  if AP ∈ PRun then apSet ← apSet + {AP}; break;
5  |  end
6  end
   // Step2: check for coupled buggy access pairs
7  capSet ← ∅;
8  for AP_i, AP_j ∈ apSet and AP_i, AP_j access different memory
   locations and AP_i, AP_j are from two threads do
9  |  flag ← true;
10 |  for successful run PRun ∈ PSet do
11 |  |  if (pc and CheckPc(AP_i, AP_j, PRun)) or (tid and
   |  |  CheckTid(AP_i, AP_j, PRun)) or (loop and
   |  |  CheckLoop(AP_i, AP_j, PRun)) then flag ← false; break;
12 |  end
13 |  if flag = true then capSet ← capSet + {(AP_i, AP_j)};
14 end
   // Step3: filter out predictable access pairs
15 for two coupled access pairs (AP_i, AP_j), (AP_m, AP_n) ∈ capSet
   do
16 |  if AP_i or AP_j is predictable by AP_m or AP_n then
17 |  |  capSet ← capSet − {(AP_m, AP_n)};
18 |  else if AP_m or AP_n is predictable by AP_i or AP_j then
19 |  |  capSet ← capSet − {(AP_i, AP_j)};
20 |  end
21 end
22 return capSet;
```

## 4.3 Test Procedure III

This test procedure aims to localize buggy access pairs manifested with Scenario III. Algorithm 3 shows the details of this test procedure. In `Step1` (lines 1 $\sim$ 6), we collect access pairs that show up in both $PRuns$ and $FRun$, denoted as *apSet*. In `Step2` (lines 7 $\sim$ 14), for any two access pairs in *apSet* that access different shared variables and are from two threads, we check whether they show up together in $PRuns$ or not. If they do not show up together in any $PRun$, they are considered as coupled buggy access pairs, denoted as *capSet*. As in Test Procedure I, false positives caused by predictable access pairs are filtered out in `Step3` (lines 15 $\sim$ 21).

To find out whether coupled access pairs show up together in an $PRun$, we need to check whether the heads and the tails of the two access pairs also form the same coupled access pairs in this $PRun$. The problem is how to represent the heads and the tails, which are dynamic memory accesses. Generally, they can be represented by the addresses of the instructions issuing them [30]. This is usually enough for Test Procedures I and II as they mainly aim to localize OV and SAV bugs, which only involve a single shared variable. However, it is insufficient for Test Procedure III.

Figure 4 explains the reason. This is an MAV bug, where $T_1$ writes shared variables $x$ and $y$ in a loop. In successful runs ($PRun1 \sim PRun3$), $T_2$ can read consistent values of $x$ and $y$ as the atomicity is satisfied. If Test Procedure III processes $FRun$ only using instruction addresses to identify dynamic memory accesses, no buggy access pair can be localized, because any two of the three access pairs in $FRun$ can be found show up together in $PRun2$ with the same instruction addresses of the heads and the tails.

Dynamic instances of the same instruction executed in different threads or calling contexts can also impede our attempt to localize buggy access pairs. But, concurrency bugs

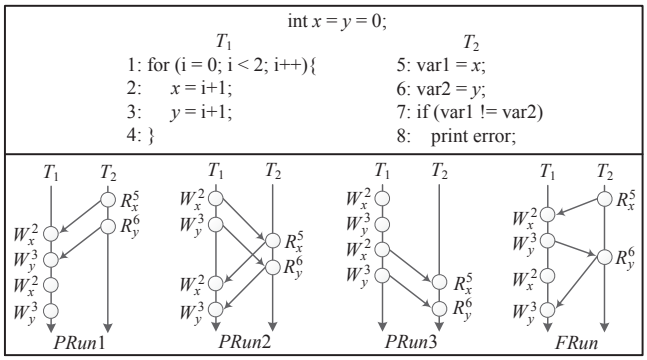



**Figure 4:** An MAV bug example to illustrate the insufficiency of instruction addresses for Test Procedure III.

involving different calling contexts are very rare in real applications [19, 5], so we omit such cases for efficiency reasons.

Besides the instruction address, denoted as *pc*, we append additional information on each shared memory access to eliminate above confusion. The information include the tid of the thread executing the access, denoted as *tid*, and the loop information of the access, denoted as *loop*, which indicates the entry and iteration of the loop if the access is executed in a loop. With these additional information, the check in `Step2` is performed at 3 progressive levels of preciseness: *pc*, *tid*, and *loop*. Each gives more preciseness, but also more offline analysis overhead.

We use the example in Figure 4 again to describe how the check in `Step2` is performed at each preciseness level. After `Step1` in Algorithm 3, $apSet = \{R_x^5 \rightarrow W_x^2, W_y^3 \rightarrow R_y^6, R_y^6 \rightarrow W_y^3\}$. In `Step2`, there are two sets of coupled access pairs to be checked: $(R_x^5 \rightarrow W_x^2, W_y^3 \rightarrow R_y^6)$ and $(R_x^5 \rightarrow W_x^2, R_y^6 \rightarrow W_y^3)$. First, at the pc-level, both of them pass the check as discussed before and no buggy access pair is localized. Second, at the tid-level, the two sets of coupled access pairs respectively have the following forms of tids in $FRun$: $(T_2 \rightarrow T_1, T_1 \rightarrow T_2)$ and $(T_2 \rightarrow T_1, T_2 \rightarrow T_1)$. As shown in the column 3 of Table 1, there is no MAV bug with the second form. Thus, only the first coupled access pairs are checked and the second is discarded. In $PRun2$, we can also find that these two access pairs have the same forms of tids, thus no buggy access pair is localized. Third, at the loop-level, which is based on the observation that programmers generally either make a whole loop atomic or create atomic regions within the loop body but rarely across iteration boundaries [24], we can find that $W_x^2$ and $W_y^3$ in $(R_x^5 \rightarrow W_x^2, W_y^3 \rightarrow R_y^6)$ are in the same loop iteration in $FRun$, but they are in different loop iterations in $PRun2$. Thus this coupled access pairs cannot pass the check in `Step2` and are reported as buggy access pairs.

## 4.4 Is the Complete $PSet$ Necessary?

As stated above, each test procedure requires the complete $PSet$, which is the set that contains *all* feasible $PRuns$. However, it is usually not available in practice. Thus, is it really necessary to have a truly complete $PSet$ when we apply the test procedures to localize buggy access pairs?

To answer this question, we conduct an experiment on 6 very diverse real concurrent applications shown in Table 2. In this experiment, each application is executed many times with the same workload in two environments, *Native* and *Random*. Access pairs showing up in each run are collected. The only difference between Native and Random environments is that a random number of microsecond delays are inserted at the points of shared memory accesses in the Random environment to explore more thread interleavings.

| Applications | KLOC | Description | Workload |
|---|---|---|---|
| FFT | 1 | FFT transformation | default input |
| RADIX | 1 | Integer radix sort | default input |
| Pbzip2 | 2 | File compressor | compress a regular file |
| Transmission | 95 | Bittorrent client | open a torrent file |
| Apache | 340 | Web server | concurrent http requests |
| MySQL | 681 | Database server | mysql test suite |

**Table 2:** Concurrent open-source applications for the experiment. "KLOC": program size in thousands of lines of code.

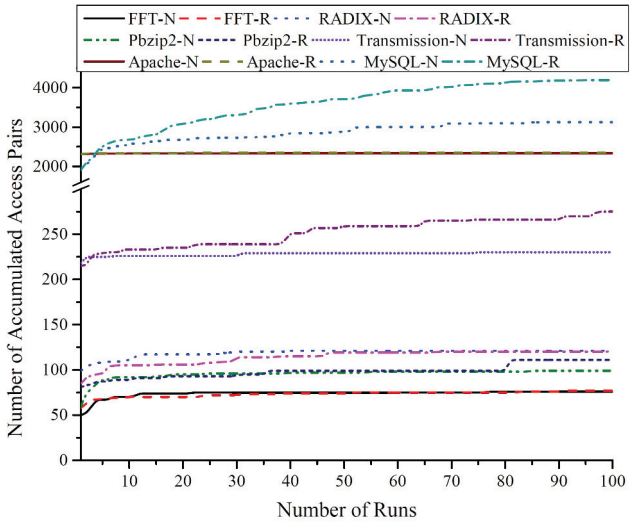



**Figure 5:** Accumulated access pairs in each program run. "*-N": Native, "*-R": Random.

Figure 5 shows the accumulated number of different access pairs in each run. When the number of test runs reaches a certain amount, the increase in the number of different access pairs diminishes quickly or remains the same, which means that the number of different access pairs in concurrent programs is *limited*. In fact, the occurrence of a new access pair depends on the explicit synchronizations used in a program and the *distance*[1] between the head and the tail of the access pair. Statistically, if these two factors are kept unchanged, all feasible access pairs can be collected. If a non-buggy access pair is very difficult to be collected, it usually means this access pair has a very low probability to appear in both successful and failed runs. Thus, although it may introduce false positives if it shows up only in failed runs, it can be ignored due to the very low probability.

Hence, there is really no need to run a large number of $PRuns$ to collect a complete $PSet$ because there is only a limited number of access pairs in $PRuns$ for most programs.

# 5. IMPLEMENTATION

This section describes LOCON, a prototype based on the proposed test procedures. LOCON has two components: 1) an online profiler, which takes a concurrent program binary and its input as inputs, executes the program binary with the given input, and collects access pairs during the executions; and 2) an offline analyzer, which takes collected access pairs as inputs and applies test procedures to localize buggy access pairs when a concurrency bug is triggered.

[1]The detailed definition of the distance can be found in [28].

The online profiler is implemented using PIN [22]. To record the head and the tail of an access pair, we need to collect the following information of the two memory accesses: *memory address*, *instruction addresses*, *thread ids*, and *loop information*. To obtain the loop information, LOCON detects loops via dominators in the control flow graph [4]. The detection is offline, and introduces no additional overhead. LOCON uses a stack to maintain the loop information for each thread. When a thread enters a loop, the entry address and current iteration number of this loop are pushed onto the stack. When this thread exits a loop, the corresponding loop information is popped from the stack. For a nested loop, when the thread exits the outer loop from the inner loop, information of the two loops needs to be popped together from the stack.

When a concurrency bug is triggered, the offline analyzer applies test procedures to localize the bug. Users are free to apply any of the test procedures via arguments of LOCON. By default, test procedures are applied in the order of I, II, and then III (in the order of pc, tid, and then loop) until the bug is localized.

# 6. EMPIRICAL STUDIES

As shown in Table 3, we use 16 commonly used concurrency bug benchmarks in prior work to evaluate LOCON [21, 37]. They include 7 extracted bugs, i.e. Group E, and 9 real-world bugs from real applications, i.e. Group R. These bugs are written in C/C++ and represent different failure symptoms of concurrency bugs. For each bug, we collect 100 successful runs but only 1 failed run. The test procedures and preciseness levels are applied to the failed run in the default order until the buggy access pairs are localized. The evaluation is conducted on an Intel Xeon machine with 1.87GHz 48 cores and Debian 6 operating system.

We also compare LOCON with a state-of-the-art tool based on a statistical approach, Falcon [30], which is most relevant and similar to LOCON. Due to the unavailability of the source code, we have to implement Falcon based on our best knowledge. Falcon uses a fixed-size window to collect access patterns on each shared memory location. Each pattern is assigned a *suspicious score*, and patterns with higher scores are considered more likely to be bugs. Our implementation uses the same window size as that mentioned in the Falcon paper. Other arguments are the same in LOCON.

## 6.1 Study 1: Effectiveness

As shown in Table 4, 10, 2, and 4 bugs are respectively localized by Test Procedure I, II, and III. As expected, the localization ability of LOCON is not limited to bugs involving only one or two shared variables. For example, Bug#3 is an MAV bug involving three shared variables, and it is successfully localized by Test Procedure III at the tid-level.

An interesting example is Bug#7, which is an MAV bug but localized by Test Procedure I. Figure 6 shows the source of this bug. $R^2_{log\_file\_name}$ and $R^3_{cur\_log}$ in $T_1$, $W^8_{log\_file\_name}$ and $W^9_{cur\_log}$ in $T_2$ respectively have atomicity semantics. Even if this bug involves two shared variables *log_file_name* and *cur_log*, it can be triggered by $W^9_{cur\_log} \rightarrow R^3_{cur\_log}$ alone, which means $T_1$ reads the right value of *log_file_name* but the wrong value of *cur_log*. Thus it can be localized by Test Procedure I. This provides a good proof that each test procedure is not limited to localize concurrency bugs manifested in the corresponding scenario.

| | | Concurrent Programs | LOC | Concurrency Bugs | |
|---|---|---|---|---|---|
| | | | | Type | Symptom |
| | 1 | s_counter | 53 | SAV | assertion failure |
| | 2 | b_account | 107 | SAV | wrong result |
| | 3 | c_list | 168 | MAV | inconsistent results |
| E | 4 | s_buffer | 186 | SAV | assertion failure |
| | 5 | l_proc_sweep | 99 | SAV | segmentation fault |
| | 6 | MySQL-1_e | 143 | MAV | inconsistent results |
| | 7 | MySQL-2_e | 101 | MAV | abnormal exit |
| | 8 | Pbzip2 | 2K | OV | segmentation fault |
| | 9 | Transmission | 95K | OV | assertion failure |
| | 10 | FFT | 1K | OV | wrong program output |
| | 11 | Aget | 2.5K | MAV | inconsistent results |
| R | 12 | Apache-1 | 290K | SAV | redundant free crash |
| | 13 | Apache-2 | 271K | SAV | confusion in log file |
| | 14 | Apache-3 | 340K | MAV | assertion failure |
| | 15 | MySQL-3 | 681K | SAV | miss in log file |
| | 16 | MySQL-4 | 696K | SAV | segmentation fault |

**Table 3:** Evaluated bugs. "LOC": program size in lines of code.

```
#define BINLOG_MAGIC   "\xfe\x62\x69\x6e"

log_file_name = "MySQL";
cur_log = BINLOG_MAGIC;

                T1                                  T2
1: log_event *next_event(...) {        7:  int queue_event(...) {
2:   if (log_file_name == "MySQL") {              ...
       ...                             8:    log_file_name = "SQL";
3:     if (cur_log != BINLOG_MAGIC)    9:    cur_log = "\x43";
4:       goto err;                               ...
5:   }                                 10: }
6: }
```

**Figure 6:** Description of Bug#7.

Compared with Falcon, the localization results of LOCON have much fewer false positives. As shown in Table 4, for the extracted bugs (i.e. the first 7 bugs), Falcon can precisely identify buggy patterns. However, for the 9 real-world programs, it is less precise and with more false positives. Although Falcon can identify all buggy patterns with high ranks (the column "rank"), some non-buggy patterns can also be ranked with high scores (the column "#rank1"). For example, in Bug#16, the number of patterns with the highest score is 282. The reason is that Falcon requires to capture a sufficient number of failed runs to gain higher scores for buggy patterns. Furthermore, Falcon failed to localize 7 of 16 bugs mainly due to the following three reasons.

First, concurrency bugs are triggered by buggy access pairs manifested in Scenario II, including Bug#9 and Bug#10. Due to $ETPs$, tails of those buggy access pairs are not actually executed in failed runs, i.e. they terminate prematurely. Hence, Falcon fails to collect those buggy patterns. However, LOCON successfully localizes them with Test Procedure II.

Second, concurrency bugs involve multiple shared variables. This contributes to Bug#3, Bug#6, Bug#11, and Bug#14. Although Falcon also tries to extract patterns from bugs involving two shared variables [29], it is very difficult to extract long patterns from complex bugs involving more than two shared variables, such as Bug#3. In contrast, LOCON is based on *access pairs* with only 2 memory accesses each, rather than long access patterns (involved many memory accesses in each pattern), hence, has no such limitation.

Third, Falcon sometimes fails to gather a buggy access pattern due to its fixed window mechanism for efficiency. This impedes the localization of Bug#4, which is an SAV bug with the buggy pattern of $R$-$W$-$R$, similar to ID-6 in Table 1. If the last access recorded in a window is a $W$ and

| | Concurrency Bugs | #SV | Locon | | | | | | | Falcon | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | #AP | #BAP | TP I | TP II | TP III | | | time(s) | #P | rank | #rank1 | time(s) |
| | | | | | | | pc | tid | loop | | | | | |
| 1 | s_counter | 1 | 9 | 2 | ✓(2/1) | - | - | - | - | 0.01 | 3 | 1 | 2 | 0.01 |
| 2 | b_account | 1 | 8 | 2 | ✓(2/1) | - | - | - | - | 0.01 | 2 | 1 | 1 | 0.01 |
| 3 | c_list | 3 | 11 | 5 | 0 | 0 | 0 | ✓(6/1) | - | 0.12 | 14 | N | 1 | 0.01 |
| 4 | s_buffer | 1 | 8 | 2 | ✓(1/1) | - | - | - | - | 0.01 | 3 | N | 1 | 0.01 |
| 5 | l_proc_sweep | 1 | 4 | 1 | ✓(1/1) | - | - | - | - | 0.01 | 4 | 1 | 2 | 0.01 |
| 6 | MySQL-1_e | 2 | 3 | 2 | 0 | 0 | 0 | ✓(1/1) | - | 0.15 | 2 | N | 1 | 0.01 |
| 7 | MySQL-2_e | 2 | 32 | 1 | ✓(1/1) | - | - | - | - | 0.01 | 8 | 1 | 1 | 0.01 |
| 8 | Pbzip2 | 1 | 224 | 1 | ✓(1/1) | - | - | - | - | 0.13 | 35 | 1 | 1 | 0.01 |
| 9 | Transmission | 1 | 142 | 1 | 0 | ✓(60/1) | - | - | - | 0.42 | 57 | N | 5 | 0.06 |
| 10 | FFT | 2 | 7187 | 7 | 0 | ✓(7/1) | - | - | - | 8.1 | 50 | N | 1 | 0.03 |
| 11 | Aget | 2 | 7027 | 2 | 0 | 0 | 0 | 0 | ✓(1/1) | 33 | 37 | N | 2 | 0.01 |
| 12 | Apache-1 | 1 | 16625 | 1 | ✓(1/1) | - | - | - | - | 27.66 | 751 | 9 | 1 | 2.33 |
| 13 | Apache-2 | 1 | 1691 | 2 | ✓(2/1) | - | - | - | - | 1.81 | 658 | 1 | 26 | 1.43 |
| 14 | Apache-3 | 2 | 4899 | 2 | 0 | 79 | 0 | 0 | ✓(1/1) | 388.3 | 563 | N | 23 | 1.23 |
| 15 | MySQL-3 | 1 | 780977 | 2 | ✓(2/1) | - | - | - | - | 740.8 | 1670 | 1 | 69 | 10.66 |
| 16 | MySQL-4 | 1 | 5510 | 1 | ✓(1/1) | - | - | - | - | 13.25 | 1730 | 1 | 282 | 29.15 |

**Table 4:** Localization results. "#SV": number of shared variables involved; "#AP": number of access pairs in failed runs; "#BAP": number of buggy access pairs; "TP": Test Procedure; "#P": number of patterns in failed runs; "rank": the rank of the buggy patterns; "#rank1": number of patterns with the highest score; "✓$(m/n)$": buggy access pairs are localized by this test procedure, $m$ is the number of access pairs that pass the check of this test procedure, $n$ is the rank of buggy access pairs; "N": failed to localize the bug.

| Applications | Pin | Locon | Falcon |
|---|---|---|---|
| Pbzip2 | 2.77X | 19.96X | 17.85X |
| Transmission | 3.05X | 7.2X | 8.17X |
| FFT | 1.36X | 3.08X | 2.01X |
| Aget | 2.35X | 41.34X | 49.52X |
| Apache | 1.25X | 43.75X | 44.25X |
| MySQL | 1.31X | 141.81X | 131.13X |

**Table 5:** Normalized runtime overhead of the online profiler.

the current access to the same memory location is a $R$ from the same thread, Falcon discards the $R$. This causes a loss of the first $R$ in the collected buggy access pattern.

## 6.2 Study 2: Efficiency

Table 5 shows the overhead of the online profiler in Locon. The overhead of Pin is also included, which is the overhead of executing an application under Pin without any instrumentation. All runtime results are normalized to the native execution time. Note that the original Falcon was implemented in Java and was compared with the runtime of Java byte code. Our implementation is in C/C++ and is compared with the runtime of native binaries. In our implementation, Locon introduces comparable overhead in each run to that of Falcon on average. We believe the overhead can be further reduced, which is left as part of our future work.

The two columns "time(s)" in Table 4 respectively shows the time consumed by the offline analyzer in Locon and Falcon. In most cases, Locon is efficient. But, for some bugs, e.g. Bug#14 and Bug#15, the offline analysis time of Locon could be somewhat long. To understand this phenomenon, the process of offline analysis is further partitioned into two steps. First, the analyzer reads access pairs from the available successful runs. Second, the analyzer applies test procedures to localize the bugs. The time consumed by each step is shown in Figure 7.

For most bugs, i.e. Bug#10, Bug#12, Bug#13, Bug#15, and Bug#16, the reading step consumes most of the analysis time, because the analyzer needs to read all available successful runs before the localization. Different from these bugs, most of the analysis time for Bug#11 is in the localizing step. The reason is that this bug is localized at the loop-level, which needs more complex checks for the localization. However, Bug#14 consumes almost all of the time in the localizing step. After studying this bug, we find that the loop information recorded in the failed run of this bug is more complex than that of Bug#11, which made the localizing step more complex and time-consuming.

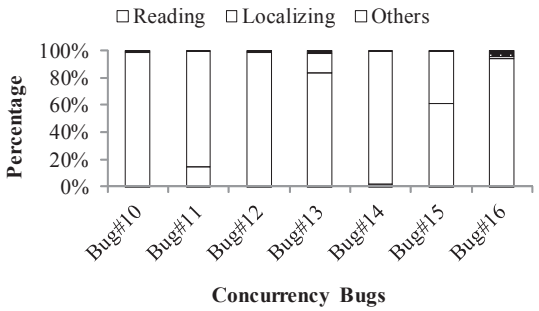
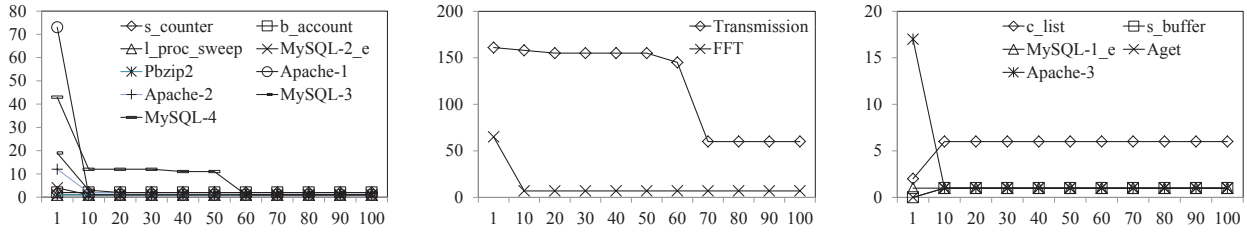



**Figure 7:** Overheads of the offline analyzer

## 6.3 Study 3: Sensitivity

We also study the sensitivity of localization results on the number of successful runs by observing the change of localization results using different numbers of successful runs. Figure 8 shows analysis results. For Test Procedures I and II, the increased number of successful runs makes the localization results more precise. For Test Procedure III, there are some differences. For example, in c_list, the number of coupled access pairs in the localization results increases with the number of successful runs. After studying the localization results, we find that the reason is that Test Procedure III requires access pairs to show up in both successful and failed runs. If the number of successful runs is very small, some access pairs may not show up in those successful runs.

From Figure 8, we can conclude that the localization results of each test procedure are reasonably precise even with a small number of successful runs collected, which is 10 for most of the bugs. Even for MySQL-4 and Transmission, the number of successful runs needed is only 60 and 70, respectively. This demonstrates the practicability of Locon for bug localization because there is no need to gather a huge number of successful and failed runs.




(a) Test Procedure I (b) Test Procedure II (c) Test Procedure III

**Figure 8:** Sensitivity analysis on the number of successful runs. X-axis is the number of available successful runs, and Y-axis is the number of access pairs (for Test Procedures I and II) or coupled access pairs (for Test Procedure III) in localization results.

## 6.4 Threats to Validity

There are several threats to validity of our empirical studies. Threats to internal validity stem from the empirical setup. We assume programmers debug and fix the program by checking shared memory access pairs until the bug is localized. Although it may be different with the real debugging process, this approach has been adopted by many statistical bug localization research. Threats to external validity constraint the extent to which our approach can be used to other types of program bugs, which may introduce potential false positives or false negatives. Due to the lack of widely accepted bug benchmark suites, related work also suffer from this threat. To ease this threat, our bug suit has covered a majority of bugs used in related work.

# 7. RELATED WORK

**Concurrency Bug Localization** Statistical analysis is an effective method for bug localization in sequential programs [14, 16, 17]. Recently, it is also applied to concurrency bugs.

Falcon [30, 29] localizes concurrency bugs by pinpointing faulty interleaving patterns. The patterns are extracted from common types of concurrency bugs that include unserializable and conflicting interleaving patterns. Falcon gathers patterns in successful and failed runs via online pattern identification. Each pattern gathered in failed runs is assigned a suspicious score according to the Jaccard index [1]. The patterns with higher suspicious scores are more likely to be concurrency bugs.

CCI [12] applies the statistical debugging technique for bug isolations. CCI dynamically monitors three types of interleaving-related predicates on program states and behavior to diagnose program failures caused by concurrency bugs. CCI identifies failure predicates via statistical models.

Based on the statistical model used by CCI, PBI [5] is proposed to diagnose concurrency bugs in software production runs. The predicates used by PBI, called MESI-predicates, are cache-coherence events that can be monitored by hardware performance counters.

The commonality of above approaches is that they are all based on some empirical statistical models, which is the main difference between them and LOCON. To precisely localize a concurrency bug, these models require a sufficient number of successful and failed runs. Unfortunately, some concurrency bugs are not easy to trigger in practice [23, 19, 28]. Therefore, it is quite challenging to gather enough failed runs for such models. In contrast, our approach has no such limitation, since it leverages different test procedures to localize buggy access pairs manifested with different scenarios. Even with only one single failed run captured, our approach can still precisely localize the bug. Furthermore, the localization results of our approach have fewer false positives due to the consideration of shared memory accesses missing in failed runs.

**Concurrency Bug Detection** There are many work on concurrency bug detection, including data races [7, 32, 33, 3, 27] and AV bugs [20, 28, 38]. Different with these work, LOCON aims to localize bugs after they are triggered.

**Concurrency Bug Testing** Some methods use different schedule or preemption policies to effectively expose concurrency bugs [23, 6, 37]. These methods can complement LOCON in gathering successful runs for the bug localization.

**Other Work** Some other tools try to automatically fix concurrency bugs [11, 13, 18]. LOCON is also helpful to these tools since they only fix bugs that have been localized.

# 8. CONCLUSION

This paper presents an exhaustive list of scenarios that indicate the manifestation differences of buggy access pairs triggering OV or AV bugs in successful and failed runs. Based on these scenarios, a concurrency bug localization framework is proposed. The framework consists of 3 test procedures to localize buggy shared memory access pairs manifested in those scenarios. Each test procedure is designed with the consideration of minimizing false positives during the fault localization. A bug localization prototype, called LOCON, is also implemented based on the 3 test procedures. Empirical results on 16 commonly used bugs show that LOCON is quite precise and practical for concurrency bug localization. Compared with existing methods, which require a sufficient number of failed runs, LOCON needs only one failed run. Besides, there is also no need to gather a huge number of successful runs for LOCON due to the limited number of access pairs in concurrent programs.

# 9. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their useful feedback. This research is supported by the National Natural Science Foundation of China (NSFC) under grants 61303052, 61332009, 61303051, and 60925009, the National High Technology Research and Development Program of China under grant 2012AA010901, the Innovation Research Group of NSFC under grant 61221062, the National Basic Research Program of China under grant 2011CB302504, the National Science Foundation under grant 1320796, CAREER Award and DOE Early Career Award.

## 10. REFERENCES


[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *TAICPART*, 2007.

[2] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *ISCA*, 1990.

[3] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *ISCA*, 1991.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[5] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu. Production-run software failure diagnosis via hardware performance counters. In *ASPLOS*, 2013.

[6] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.

[7] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI*, 2000.

[8] S. Focus. Software bug contributed to blackout. `http://www.securityfocus.com/news/8032`.

[9] P. Godefroid and N. Nagappan. Concurrency at Microsoft – an exploratory survey. In *Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[10] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. In *IEEE TSE*, 1996.

[11] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[12] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*, 2010.

[13] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *OSDI*, 2012.

[14] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE*, 2005.

[15] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. In *IEEE Computer*, 1993.

[16] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[18] P. Liu and C. Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *ICSE*, 2012.

[19] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.

[20] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[21] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, 2009.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[23] M. Musuvathi and S. Qadeer. Iterative context bouding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[24] A. Muzahid, N. Otsuki, and J. Torrellas. Atomtracker: A comprehensive approach to atomic region inference and violation detection. In *MICRO*, 2010.

[25] MySQL. Bug report time to close stats. `http://bugs.mysql.com/bugstats.php`, Dec. 2011.

[26] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data race using replay analysis. In *PLDI*, 2007.

[27] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.

[28] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[29] S. Park, R. Vuduc, and M. J. Harrold. A unified approach for localizing non-deadlock concurrency bugs. In *ICST*, 2012.

[30] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: Fault localization in concurrent programs. In *ICSE*, 2010.

[31] PCWorld. Nasdaq's facebook glitch came from race conditions. `http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html`.

[32] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *PLDI*, 2006.

[33] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. In *ACM TOCS*, 1997.

[34] Transmission. `http://www.transmissionbt.com/`.

[35] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[36] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *FSE*, 2011.

[37] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *OOPSLA*, 2012.

[38] B. Zhou, J. Too, M. Kulkarni, and S. Bagchi. WuKong: Automatically detecting and localizing bugs that manifest at large system scales. In *HPDC*, 2013.