PVcon: Localizing Hidden Concurrency Errors With Prediction and Verification

ZHENWU XU1, JINAN SHEN1, PENG LUO2, AND FANG LIANG3

1School of Information Engineering, Hubei Minzu University, Enshi 445000, China
2Huawei Technologies Company Ltd., Shenzhen 518000, China
3School of Mathematics and Statistics, Hubei Minzu University, Enshi 445000, China

Corresponding author: Jinan Shen (jnshen@hust.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61662022, and in part by the Incubation Project for High-Level Scientific Research Achievements of Hubei Minzu University under Grant PY20008.

ABSTRACT Multi-core techniques have been widely used in various hardware platforms, promoting development of concurrent programs. Unfortunately, concurrent programs are prone to concurrency errors. At present, it is still an open issue to provide high coverage detection of concurrency errors. In this paper, we present an enhanced dynamic concurrency error detection technique, called PVcon, which can detect more concurrency errors than existing techniques. PVcon first finds suspicious locations of concurrency errors by applying a novel relation, elastic-causalities (EC), and then strictly verifies each suspicious concurrency errors with a lightweight intentional scheduling technique. We have implemented PVcon in C/C++ and evaluated it by localizing concurrency errors from real-world programs. The experimental results show that PVcon can efficiently detect twice more concurrency errors than other techniques.

INDEX TERMS Concurrency error, detection technique, causal relation, interleaving access.

I. INTRODUCTION

To efficiently utilize hardware resources, concurrent programs are widely developed and adopted in modern multi-core hardware platforms. Unfortunately, concurrent programs are prone to concurrency errors [1]–[3] and it is challenging to debug, detect and repair concurrency errors. Moreover, concurrency errors would cause incorrect outputs, data losses and even system crashes [4]–[6]. They have become a great threat to modern concurrent systems.

Figure 1 shows the statistics of security loopholes caused by data race in the US National Vulnerability Database in recent years. According to the survey, the number of security loopholes caused by data was also around 100 in 2019. It can be seen from the figure that with the widespread adoption of multi-threaded programming, security loopholes caused by data race have increased in recent years.

Typical modern dynamic concurrency error detection techniques use one of two techniques: scheduling test or pattern analysis. The scheduling test techniques [7]–[10] try to explore as many interleavings as possible. However, it is an NP-hard problem to discover all interleavings [11]. They miss many hidden concurrency errors, interleavings of which are not triggered. Moreover, they take long time, because of running subject programs enormous times [8].

The pattern analysis techniques [12]–[14] only need to run programs for a small amount of times. They localize concurrency errors by identifying faulty data-access patterns from execution traces. Some techniques [13] pinpoint faulty patterns using sliding-windows. However, the sliding windows have limited size. Errors would be missed when data-access interleavings exceed the bound of sliding windows. Other techniques [12], [14] predict faulty patterns from execution
traces by analyzing the happen-before (HB) [15] relations. However, HB-based techniques are sensitive to thread interleavings [16], [17], which deem that the events synchronized by every synchronization operation are HB ordered. They cannot find faulty patterns behind accidental HB edges [17], so that many concurrency errors are missed.

Figure 2 shows an example of hidden concurrency error in one program, which cannot be detected by previous techniques. Execution order of the program follows the number of code lines. In this execution, Thread 1 releases a lock and Thread 2 sequentially acquires it. So unlock(l) at line 5 has an HB edge to lock(l) at line 6, and previous techniques would not report any warnings. However, in another execution Thread 2 is also able to acquire the lock first, and Thread 2 will read *p (at line 7) before Thread 1 initializes it (at line 4). Thus, there is a hidden concurrency error leading to an un-initialized read in this program, which cannot be detected by previous techniques.

In the second step, PVcon strictly checks each suspicious interleaving. For the program in Figure 2, PVcon would force Thread 2 to acquire the lock first, and make read(*p) (line 7) happen before init(p) (line 4). In order to reduce times of running subject programs, PVcon adopts a novel technique, called duplicate fork, which does not run subject programs anew for each suspicious concurrency error, but forks one duplication of execution before the interleaving of suspicious concurrency errors. Then PVcon carefully controls the thread scheduling of the forked execution to trigger each interleaving pattern of errors. The duplicate-fork technique can reduce time costs involved in PVcon.

The main contributions of this paper are listed as follows:

- We present a new relation, elastic-causalties (EC), which maximally relaxes the restrictions of previous causal relations. Based on EC, more hidden concurrency errors can be localized compared with previous techniques.
- We present a precise dynamic EC-based concurrency error detection technique, PVcon, which adopts two novel techniques: recursive-analysis to analyze EC relations between events, and duplicate-fork to validate suspicious errors.
- We have implemented PVcon in C/C++ and evaluated it by localizing concurrency errors from 22 real-world benchmarks. The experimental results show that PVcon detects 33 concurrency errors, more than twice of Falcon [13] and Maple [8].

The rest of the paper is organized as follows. Section II presents related work. The new relation, elastic-causalties, is presented in Section III. Our concurrency error detection technique, PVcon, is described in Section IV, and evaluation results are presented and analyzed in Section V. Section VI concludes the paper.

II. RELATED WORK

In this section, we introduce previous studies related to concurrency errors.

Data races are the first raised concurrency errors [11], [18], [19]. Previous race detectors can be divided into two categories: static race detectors [20]–[22] and dynamic race detectors [17], [23]–[25]. Static race detectors directly scan data races from source codes. However, the static detectors are known to have a high false positive rate because of the lack of run-time information.

Dynamic race detectors monitor the execution of multi-threaded programs and detect data races by analyzing program traces. Typically, dynamic race detectors use one of two principles: lockset [17] principle or happen-before (HB) [26] ordering. The lockset-based detectors [26] report race warnings if locks are inconsistently held on accesses to the same shared-memory locations. Since they ignore other synchronization events, they report many false warnings. The HB-based detectors [23], [27], [28] detect data races by analyzing whether two events violate the partial order according to the pre-defined causal relation. This kind of
detector can detect races precisely but miss many hidden races. There are also many hybrid studies [25], [29] that combine the two principles.

Most of the previous concurrency error detection methods can be divided into two types: heuristic method and test method. The heuristic method [13], [14], [30] monitors the execution of programs and analyzes traces to detect and locate concurrency errors. CTrigger [14] detects atomicity violations by analyzing the execution traces of programs. It studies the interleaving characteristics of synchronization events in concurrent programs and finds that the atomicity violation errors are caused by four types of interleaving patterns. Falcon [13] is a typical dynamic concurrency error detector that monitors the execution of multithreaded programs and detects concurrency errors by analyzing the traces. It uses a sliding window technique to match the interleaving patterns of concurrency errors. However, it may miss the concurrency errors because of limited size of sliding windows. UNICORN [30], similar to previous detectors, localizes interleaving patterns on concurrency error detection and ranks the patterns by their suspiciousness scores.

The test method usually applies scheduling techniques that run programs with random or full system scheduling to trigger concurrency errors. Burckhardt et al. [9] present a randomized scheduler, called PCT, to detect concurrency errors based on randomness. Nagarakatte et al. [7] present a parallelized version of the PCT algorithm (PPCT) that can test multi-threaded programs in parallel. Pinso [31] also exploits the PCT algorithm to trigger concurrency errors and analyzes the interleaving patterns of the reported errors. Maple [8] presents the full scheduling method, which traverses all interleavings to detect concurrency errors. To reduce time overhead, ColFinder [32] proposes a static method to find the suspicious locations and verifies them with thread scheduling. Test case generating technique is used in test method. He et al. [33] present a test method automatically generating test cases to trigger concurrency errors.

Recently, some theoretical studies [34]–[38] have been developed to analyze concurrent programs. For instance, Huang [35] present two algorithms to identify concurrent accesses to shared-memory locations. Their work is helpful to concurrency error detection. Ou and Demsiky et al. [36] propose a correctness model to check concurrent data structures. Their method can detect the concurrency bugs in concurrent data structures. Hayes et al. [38] start his research on algebra and use it to infer the dependency/guaranteed concurrency of the shared memory model. In the abstract algebra of the paper, their proof is simpler, so a higher degree of automation can be achieved. Algebra has been coded in Isabelle/HOL, which provides the basis for tool support for program verification.

Concurrency problems are nasty, and many techniques [39]–[45] have been developed to prevent and repair them. For instance, Battig and Gross [39] present a programming approach to present concurrency errors. Their approach synchronizes all access events in atomic sections and a programmer manually splits the atomic sections to increase threads.

CFix [41] is an automatic tool that can repair concurrency bugs. It identifies concurrency bugs with the assistance of other detectors [14], [46], and analyzes the interleaving pattern of each bug. Parizek and Lhotak et al. [44] propose the DFS-RB algorithm, which extends the standard algorithm of depth-first traversal through early backtracking. Specifically, before all outgoing transformations are studied, it is possible to trace back from a state early. The DFS-RB algorithm is non-deterministic (it uses random numbers and the values of several parameters to determine when and how early backtracking takes place in the search).

III. ELASTIC-CAUSALITIES

This section introduces our proposed relation, elastic-causalities (EC), and subsequently presents the function of EC on concurrency error detection.

A. DEFINITIONS

The model in this paper considers that all events in an execution trace have a global order and the causal relations are related to the global order. The definition of EC is as follows.

Definition 1 (Elastic Causalities): Elastic causalities \( \xrightarrow{\text{EC}} \) is the causal relation of events that cannot be reordered in execution traces. Formally, it is the smallest relation described as follows:

1) Events in the same thread are ordered by execution sequence. \( \alpha \xrightarrow{\text{EC}} \beta \) if \( \alpha \) and \( \beta \) are in the same thread and \( \alpha \) happens before \( \beta \).
2) Events synchronized by non-lock synchronization operations are ordered as they appear. \( \alpha \xrightarrow{\text{EC}} \beta \) if \( \alpha \) and \( \beta \) are synchronized by non-lock synchronization operations and \( \alpha \) happens before \( \beta \).
3) Two lock-based critical sections are ordered if they contain EC ordered events. \( \alpha \xrightarrow{\text{EC}} \beta \) if \( \alpha \) and \( \beta \) are in two different critical sections which contain two events \( \gamma \) and \( \delta \), and \( \gamma \xrightarrow{\text{EC}} \delta \).
4) \( \xrightarrow{\text{EC}} \) is closed under composition with itself. \( \alpha \xrightarrow{\text{EC}} \beta \) if \( \alpha \xrightarrow{\text{EC}} \epsilon \) and \( \epsilon \xrightarrow{\text{EC}} \beta \).

\( \xrightarrow{\text{EC}} \) is a subset of HB relations. The differences between HB and EC appear in the cases of critical sections. More accurately, the events synchronized by lock operations can be reordered only if the lock operations are not synchronized by other synchronization operations. For instance, in Figure 2, Thread 1 releases a lock and Thread 2 sequentially acquires it. Then HB deems that there is an HB edge between the two events, as shown at line 5 and line 6. However, their lock operations are not synchronized by other operations, there is no EC edge between them, and Thread 2 can also acquire the lock first. Then their events can be reordered. The cases that the critical sections contain EC ordered events appear when the lock operations are synchronized by other synchronization operations. We give two examples in Figure 3. In Figure 3(a), Thread 1 first acquires the lock, and then Thread 1 forks Thread 2. Thus, Thread 1 can always
acquire the lock first, and the two critical sections cannot be reordered. In Figure 3(b), Thread 1 releases the lock before it forks Thread 2. The critical section in Thread 1 can always happen before the critical section in Thread 2. The critical sections in Figure 3 all are ordered and have EC edges. They are the cases of two pairs of nested synchronization operations. If there are more than two pairs of nested synchronization operations, we need to analyze their causal relations by peeling each nested relation. We solve this complex problem in the next section.

EC maintains the causal relations in semantics defined by programmers. It maximally relaxes the restrictions of relations between events. Thus, we have the following theorem.

**Theorem 1 (EC Is Maximal):** Given two events in an execution trace, if the execution sequence of them can be reordered, then the two events are not EC-ordered.

**Proof:** In the execution trace, if the execution order of two events is not determined, then the two events are not ordered by any synchronization operations or are ordered by some lock operations that are not nested with any EC-ordered synchronization operations. If they are not ordered by any synchronization operations, they are not EC-ordered. If they are ordered only by some lock operations that are not nested with any EC-ordered synchronization operations, they are also not EC-ordered. Hence, the two events are not EC-ordered.

This maximality theorem means that in an execution trace if two events can be reordered, they must not be EC ordered. Thus, based on EC, we can pinpoint all the events that can be reordered, which is the key of our concurrency error detection.

**Definition 2 (EC-Conflict):** Two events are conflicting events, if they access the same shared-memory location, at least one of them is a write, and they are not EC ordered. We say there is a conflict.

### B. CONCURRENCE ERRORS FOR LOCALIZATION

Interleaving accesses to shared-memory locations consist of conflicting events, which lead to concurrency errors [13]. In this paper, we focus on deadlock-free concurrency errors. If two conflicting events happen simultaneously, there is a data race (we also say there is a race). If two or more conflicting events happen in an unintended sequence of operations leading to unintended program behavior, then there is an order violation; if their execution sequence interrupts an atomic region of program segmentations, there is an atomicity violation. In the following sections, we refer to atomicity violations and order violations as violations.

In order to apply EC into concurrency error detection, we divide conflicting events into two types: races and non-race conflicts. If two conflicting events are not protected by the same lock set and can happen simultaneously, there is a race (condition). Otherwise, there is a non-race conflict. We analyze the cases that conflicts lead to concurrency errors as follows.

**Data race.** It is obvious that there is a conflict within each data race. There is a hidden race when two conflicting events, which are not protected by the same lock set, are temporally synchronized by lock release-acquire (there are HB edges) that are not EC-ordered. The hidden races would occur if the lock operations are reordered.

**Violations.** The error patterns of violations are interleaving accesses to shared-memory locations. This paper subdivides interleaving accesses to conflicts. If the conflicting events leading to violations can happen simultaneously, there are also races between them. Thus, the EC-based detector can also pinpoint violations on race detection. In particular, our technique can locate violations in the verification phase if the conflicting events leading to violations are protected by locks, there are race-free conflicts. Hence, conflicts can indicate all violations.

### IV. THE PVcon FRAMEWORK

In this section, we introduce our dynamic EC-based concurrency error detection technique, PVcon, which has been implemented in C/C++. The inputs of PVcon are binary programs.

Figure 4 shows an overview of PVcon. PVcon consists of two steps. The first step pinpoints all conflicts in execution traces, which are suspicious interleavings of concurrency errors. There are three parts in this step, including monitor, EC builder, and analyzer. The monitor obtains a sequence of events in execution order, which are sent to the EC builder. The EC builder constructs EC relations between the events, and the analyzer locates conflicting events. In the second step, PVcon verifies each conflict with intentional scheduling with a validator. The validator outputs the information of real concurrency errors.

PVcon adopts two key techniques. The first technique is recursive-analysis, which is used in the EC builder. In execution traces, the cases of nested synchronization operations commonly exist. PVcon analyzes EC relations involved in nested operations using recursive-analysis to peel each nested relation.

The second technique is duplicate-fork, which forks one duplication of execution to trigger as many conflicts as possible. This technique is applied in the validator. It reduces a lot of time cost. The following sections introduce each part of PVcon in details.
**A. THE MONITOR**

The monitor is a instrumentation tool based on PIN framework [47], which is a cross-platform binary instrumentation framework. It does not make any changes to the subject programs and just instruments itself to the executions of binary executable files. The monitor tracks the execution of subject programs at run-time and records the information of access events to shared-memory locations and synchronization operations. It records detailed run-time information of events, including thread-id, read-write, father-thread-id, program counter, and memory location.

To make all events in a global order, the monitor binds all threads of subject programs to a given core of a processor. In this implementation, we achieve it by setting CPU-affinity of threads (using pthread_setaffinity_np). Then the obtained events are in a sequential order. For instance, the sequential events in Figure 4 are tracked by the monitor. This trace comes from the execution of the program in Figure 2. Then, all recorded events are sequenced in execution order as input to the EC builder.

**B. THE EC BUILDER**

The EC builder analyzes sequential events from the monitor and constructs EC edges between events. The builder applies a novel structure, called EC-frame, to store access information and EC relations. One EC-frame consists of frames and EC edges. The causal relations among events are represented by the EC edges between frames. For instance, in Figure 4 there is a built EC-frame based on the sequential events from the monitor. The arrows indicate EC edges. Each frame stores the information of events, including read/write and lock.

The EC builder applies vector clocks to implement EC edges, which is similar to vector time [48]. Each frame in an EC-frame has a vector clock. The EC edges between two frames are represented by their vector clocks. For instance, to check relations between two frames, we can compare their vector clocks.

**Frame.** Frame is the carrier of events. Each frame stores a series of continuous accesses to shared-memory locations, which means that the access events in each frame are not interrupted by any synchronization operations. Each frame also has its vector clock to indicate EC edges with other frames. So vector clocks of frames carry EC relations of events. To maintain atomicity of locks, the frames also store the information of lock set. For instance, in Figure 4 each frame contains the formation of lock set, and all of the events in each frame are protected by the lock set. In addition, each frame uses a hash table to store the access information, and the hash key is the access address.

The construction of EC edges is based on synchronization operations. The frames in each thread are sequenced in execution order. The frames in different threads maintain the EC edges decided by synchronization operations. We present the construction protocols of building EC-frame as follows.

**EC principle.** If and only if each entry of thread a's vector clocks are not bigger than thread b's, thread a does not happen after thread b, $V_a \leq V_b$ (concurrent) iff $\forall t. V_a(t) \leq V_b(t)$.

**Initialization.** When a thread $a$ is created, PVcon creates a new frame $F_a$ for thread $a$, and the entry of thread in the new frame’s vector clock is filled with one, $V_{F_a}(a) = 1$.

**Synchronization operation.** When a thread $a$ performs an operation $a.\delta$ (such as fork, join and lock operations), the EC builder in PVcon first implements the following operation.

- Creates a new frame $F_a$ for thread $a$, this new frame inherits lock set and the vector clock of its previous frame, and the vector clock of $F_a$ increments one to the entry of thread $a$, $V_{F_a}(a) = V_{F_{a-1}}(a) + 1$.

Then, for different operations PVcon takes different actions.

**Thread fork.** Thread $a$ forks a child thread $b$.

- Creates the first frame $F_b$ for thread $b$, $F_b$ inherits the vector clock of $F_a$, and the entry of thread $b$ in the vector clock of $F_b$ is set as 1, $V_{F_b}(b) = 1$.

**Thread join.** Thread $a$ joins another thread $b$, and $F_b$ is the last frame of $b$.
Vector clock of $F_a$ holds the maximum values between $F_a$ and $F_b$, and increments one to the entry of thread $b$, $V_{F_b} = \tau_t.\max (V_{F_a}(t), V_{F_b}(t)), V_{F_b}(b) = V_{F_b}(b) + 1.$

Access events. Thread $a$ reads/writes a shared-memory location:
- Thread $a$ adds the access information, including thread-id, read-write, father-thread-id, program counter, and memory location, to the last frame of thread $a$.

Recursive-analysis. When the lock operations are nested with other operations, they may be EC ordered. To analyze and distinguish EC edges, we peel the nested relations. Focusing on the lock-operations, thread $a$ releases a lock $l$ (lock-release), and thread $b$ sequentially acquires it. $F_a$ and $F_b$ are the newly-built frames in thread $a$ and thread $b$, respectively. $F_{a,l}$ is the created frame by lock-acquire corresponding to the lock-release in thread $a$. Then the recursive-analysis is that:
- $F_a$ removes lock $l$ from its lock set, and $F_b$ adds lock $l$ to the lock set.
- If the entry of thread $a$ in the vector clock of $F_{a,l}$ is not bigger than that of $F_b$, then vector clock of $F_b$ holds the maximum values between $F_b$ and $F_a$, $V_{F_b} = \tau_t.\max (V_{F_b}(t), V_{F_a}(t)).$

Figure 5 shows an example that two locks are nested with a fork operation. Thread 1 first acquires lock l1, and next forks a new thread Thread 2. Then PVcon creates two frames F12 and F13 for Thread 1, one frame F21 for Thread 2. After that, Thread 1 acquires another lock l2, and then releases the two locks. PVcon creates another three frames F14, F15 and F16 for Thread 1. Next, Thread 2 acquires lock l1. PVcon creates F22 for Thread 2, whose vector clock is (3,2).

The lock operations are nested with the fork, and PVcon performs recursive-analysis. It checks the vector clock of the frame created by previous lock-acquire, F13:(3,0), with the vector clock of F22:(3,2). Their entries in Thread 1 are equal. According to the protocols, the vector clock of F22 holds the maximum values between F22 and F15, and PVcon updates the vector clock of F22 as (5,2). The same analysis is performed with T2.lock(l2), and the vector clock of newly-built frames.

Algorithm 1 Concurrent Frames Identification

```
function detect(p, l)
    for each pair (ξ, p) ∈ Ω ∩ Ω(p, l)
        if ξ = ζ
            return true
    return false
end function
```

Finding concurrent frames is an essential step of detecting conflicts. Since the storage unit is the frame, we first find concurrent frames, which are not EC-ordered. All events that access the same shared-memory location in two different concurrent frames are conflicting events. Since we perform each detection analysis upon the time when a new frame is built, there is only the need to check whether the last frame of the target thread is concurrent with frames of other threads, which would save time cost and avoid duplication. We illustrate the algorithm to find concurrent frames in Algorithm 1.

Algorithm 1 presents the procedure to identify concurrent frames. First, let the last frame $(F_{ξ,l})$ of the target thread $ξ$ be the target frame $ξ$ (line 1). Then it is compared with the prior frame $η$ of the last frame $(F_{p,l})$ in one of the other threads $p$ (traversing all threads) (lines 2-3). If it is not the last frame of $p$ and does not happen before $η$ or the entry of $p$ in $η$’s vector time ($V_{η}(p)$) is not smaller than $ξ$’s vector time ($V_{ξ}(p)$)
Algorithm 2 Conflict Detection (detect)

1: for memory locations \( M \) such that \((M \in A) \cap (M \in B)\) do
2: \( (M_{A,r} \cap M_{B,w}) \cup (M_{A,w} \cap M_{B,w}) \cup (M_{A,w} \cap M_{B,r}) \) then
3: \( A \cap B \cap I = \emptyset \) then
4: Report races between \( M_A \) and \( M_B \)
5: else
6: Report non-race conflicts between \( M_A \) and \( M_B \)
7: end if
8: end if
9: end for

(line 4), then \( \eta \) and \( \zeta \) are concurrent frames, and the conflicts should be detected between them (line 5). Finally, the algorithm specifies the previous frame of the comparison frame and repeats the process (lines 6-8). Algorithm 1 traverses all threads and checks the frames in each thread from the bottom up to the top frame or up to the frame that has an EC edge to the target frame.

From the detection analysis, the fastest case is that the target frame compares only one frame in each thread (the target frame and the first compared frame in each of the other thread are EC-ordered). In this case, the complexity of Algorithm 1 is \( O(N) \), in which \( N \) is the number of threads. The slowest case is that the target frame compares all frames in each thread (the target frame and the compared frames in other threads are not EC-ordered). In this case, the complexity of Algorithm 1 is \( O(M) \), in which \( M \) is the number of synchronization operations. In summary, the average complexity of Algorithm 1 is also \( O(M) \).

3) DETECTING CONFLICTS

We detect conflicts (that are divided into races and non-race conflicts) from concurrent frames.

Algorithm 2 outlines how to detect conflicts between two concurrent frames. The algorithm checks all shared-memory locations in two concurrent frames (line 1). For each pair of accesses to the same shared-memory location \( M \), if at least one of them is a write, there is a conflict (lines 2-8). If the two accesses do not share the same lock set and at least one of them is a write, the algorithm reports races (line 4). If the two accesses share the same lock set and at least one of them is a write, the algorithm reports non-race conflicts (line 6). With Algorithm 2, both of races and non-race conflicts can be detected.

The access information of shared-memory locations is stored in frames by using hash storage, and the complexity of the search algorithm is linear with the number of shared memory locations. The complexity of Algorithm 2 is \( O(N_M) \) (\( N_M \) is the number of shared-memory locations).

D. VALIDATOR

The validator checks each detected conflict and identifies the conflict to be races and violations. If conflicting events happen in an unintended sequence leading to unintended program behavior, there is an order violation; if a conflict interrupts an atomic region of code, there is an atomicity violation.

To validate each conflict, programs should be run again, which takes a lot of time. To reduce time cost, we propose a novel technique, called duplicate-fork, to validate each conflict. Duplicate-fork is based on two key insights. First, the nondeterminacy of concurrent programs can be exhibited by conflicts, and if we sort all conflicting events in the original order, which assists to replay their executions without considering specific procedures, such as random number generation. Second, if we run one more time for each conflict of the same program, there are a large number of repetitive executions. Combining the two insights, the validator runs programs for few times.

Figure 6 shows an overview of the duplicate-fork technique. The black thick lines represent the first test execution of a concurrent program. PVcon locates many conflicts in test executions. For each conflict, the validator puts fork points which have two types.

- If the conflict is a race conflict, the fork points are before the race accesses.
- If the conflict is a non-race conflict, the fork points are before lock-acquire operations.

Then the validator will fork a duplication of execution that performs the conflicting events in reverse order (with original execution). The main execution continues to run in the original order. In this way, the validator checks all conflicts.

Then we introduce the implementation details of intentional scheduling. First, the validator binds all threads of programs to a given core of a processor by setting CPU-affinity. Then the thread-schedule mode is set to be priority scheduling (SCHED_FIFO). Next, the validator sets different priorities for the conflicting events before fork points. Last, the validator tracks outputs and error-signals to judge whether conflicts would cause concurrency errors or not. The validator records detailed information of concurrency errors.
V. EVALUATION

This section verifies PVcon’s search capabilities and performance by finding data competitions in popular benchmarks and actual applications. Some experiments carry out to test the effectiveness of PVcon and compared with the latest technologies including HB [15], Falcon [13], and other techniques.

We evaluate effectiveness and efficiency of PVcon on the detection of concurrency errors from 22 real-world benchmarks. We have conducted three evaluation studies, and several key findings are listed as follows.

- We study the detection coverage of EC with HB [15] (Section V-B). We find that EC is weaker than HB, and EC-based detector (PVcon) can detect much more concurrent frames than HB-based detector. However, the EC-based detector only need about 0.8x more time cost than HB-based detector. Moreover, EC-based detector needs less space overhead, because the EC-based detector implements a new data structure (EC-frame).

- We compare with the pattern technique (Section V-C). We evaluate PVcon and the state-of-art pattern technique, Falcon [13], to detect concurrency errors from real-world benchmarks. The experimental results show that PVcon can detect 33 concurrency errors, more than that of Falcon (only 16).

- We compare with the state-of-art scheduling technique, Maple [7] (Section V-D). We evaluate their detection coverage and compare the running times of subject programs. The results indicate that PVcon runs programs for much less times than Maple, and can still locate more concurrency errors than Maple.

- We compare with the other technique (Section V-E). The comparison is based on the number of benchmarks and errors. PVcon can detect 33 concurrent errors from 13 programs. The novel technique, in recent 3 years, can detect 33 actual errors from 19 programs. According to the comparison, PVcon has certain advantages in error detection.

A. EXPERIMENTAL SETUP

Table 1 shows the information of benchmarks used in our experiment. The first column lists the benchmarks, from three well-known benchmark suites, including SPLASH2 [49], PSet [50] and Inspect [51]. Aget is a multi-threaded download accelerator, cherokee is a lightweight web server, and pbzip2 is a parallel file compressor. Columns 2-5 show statistics of benchmarks, such as number of threads, shared-memory locations, lock operations and non-lock synchronization operations. Shared memory locations include global variables and heap memory locations. Table 1 shows that most test programs have more than three threads, and the test program has a large number of shared memory locations, far exceeding the number of simultaneous operations. #Thrd describes the total number of created threads. #LOC displays one of the key complexity of concurrency error detection as that interleaving accesses to shared-memory locations cause concurrency error. We divide the synchronization operations into lock operations and non-lock operations, because the causal relations synchronized by lock operations may be not EC ordered. Table 1 shows that the number of lock operations is much more than that of other synchronization operations.

The first set of (nine) programs are from SPLASH2 [49]. It is convenient to change threads of them, and we use them to evaluate the detection coverage of HB and EC in our experiment. In SPLASH2, barnes, fnm, ocean volrend and water are apps benchmarks, and cholesky, fft, lu and radix are kernel benchmarks.

The second set of (five) applications are from PSet [50]. They are real-world applications. Each of them has at least one concurrency error. The last set of (eight) benchmarks are from Inspect [51]. They contain three kinds of concurrency errors, including data races, violations and deadlocks.

All experiments are performed on a machine with 4-cores-8-threads 3.3GHz Intel Xeon CPUs (8 MB cache) and 8 GB RAM running Linux with kernel version 2.6.32.

| Programs  | #Thrd | #LOC | #Lock | #Sync |
|-----------|-------|------|-------|-------|
| barnes    | 16    | 38.4K| 10.2K | 302   |
| fnm       | 16    | 131.2K| 17.5K | 547   |
| ocean     | 16    | 353.8K| 45.3K | 10.5K |
| volrend   | 16    | 362.9K| 14.0K | 670   |
| water     | 16    | 825   | 38.7K | 350   |
| cholesky  | 16    | 987.4K| 40.9K | 31    |
| fft       | 16    | 2.8K  | 124.2K| 32.1K |
| lu        | 16    | 15.3K | 136.4K| 35.2K |
| radix     | 16    | 618.8K| 1.0K  | 306   |
| aget      | 12    | 9.2K  | 3.6K  | 33    |
| cherokee  | 40    | 108K  | 40    | 5.2K  |
| memcached | 4     | 1.1K  | 385   | 22    |
| pbzip2    | 5     | 112   | 22    | 15    |
| transmission | 3 | 83.1K | 42.2K | 6     |
| bbf      | 5     | 48    | 38    | 24    |
| bzip2cmp  | 6     | 804   | 138   | 49    |
| pscan     | 4     | 797   | 128   | 59    |
| stateful  | 3     | 27    | 12    | 7     |
| streamcluster | 4 | 288   | 16    | 16    |
| swarm     | 3     | 4.5K  | 198   | 81    |
| sync      | 3     | 22    | 8     | 9     |
| thread-pool | 4 | 90    | 4     | 7     |
There are three cases about the relationship between concurrent frames and threads. The first case is that there is a yawning gap on the increase rates between HB and EC, such as barnes, fmm, ocean, volrend, fft and lu. They have many numbers of lock operations, from 2.8K to 362.9K. However, the number of other synchronization operations is much less. The cause of this case is that HB is sensitive to the lock operations, while EC is not. When two or more threads compete to one lock, their execution order may be different in different executions.

The second case is that the increase rates of HB and EC are very close (paralleled), such as water and radix. Most of their synchronization operations are lock operations, much more than non-lock operations. For instance, for water, it has 38.7K lock operations, but only 350 non-lock operations.
The number of lock operations is 100 times than that of other synchronization operations.

The last case is that the increase rates of HB and EC tend to be close, such as cholesky. The trace of cholesky with 16 threads has more than 40K synchronization operations. However, most of its operations are lock operations. It also has the most shared-memory locations among the nine benchmarks.

2) EFFICIENCY
We then evaluate the efficiency of HB and EC. Figure 16 displays the time costs of detection instrumented with HB and EC. All benchmarks have 16 threads. On the instruments with HB and EC, it is obvious that EC needs more analysis. Thus, EC takes more time cost, as shown in Figure 16. Only for radix, EC needs a little less time cost. The average time cost of EC is 252.4s, and that of HB is 140.8s. EC can detect much more concurrent frames than HB, and needs only 0.8X more time cost than HB. For lu, HB and EC need the most time cost, because its trace has the most synchronization operations. EC and HB need to cost much time on the analysis of causal relations synchronized by operations. The least time cost is the detection on radix, which has the least synchronization operations. Since it has few lock operations and we have implemented EC-frame, EC needs less time cost than HB. Figure 17 shows the space overhead of HB and EC. The results illustrate that EC needs less space overhead than HB. For most benchmarks, EC takes less space overhead, and only for water EC needs a little more. The main reason is that we do not store vector time for each shared-memory location, but all accesses in one frame share one vector clock. In this way, our technique needs less space overhead than previous techniques. The average space overhead of EC is 43.9MB, 74.2% of that of HB.

C. COMPARED WITH PATTERN ANALYSIS TECHNIQUE
In this study, we compare our technique with the pattern analysis technique. We have evaluated PVcon and Falcon, the state-of-art pattern analysis technique, to detect concurrency errors from 13 benchmarks. Table 2 shows the results that PVcon has located 33 concurrency errors, more than twice of that of Falcon. The experimental results illustrate that our technique has a higher detection coverage.

| Programs       | Races | Atomics violations | Order violations | Deadlocks |
|----------------|-------|--------------------|------------------|-----------|
| Falcon         | 1     | 0                  | 0                | 0         |
| PVcon          | 21    | 9                  | 0                | 0         |
| Falcon         | 2     | 1                  | 0                | 0         |
| PVcon          | 10    | 0                  | 0                | 0         |
| Falcon         | 0     | 0                  | 0                | 0         |
| PVcon          | 0     | 0                  | 0                | 0         |
| Falcon         | 0     | 0                  | 0                | 0         |
| PVcon          | 0     | 0                  | 0                | 0         |
| Falcon         | 0     | 0                  | 0                | 0         |
| PVcon          | 0     | 0                  | 0                | 0         |
| Falcon         | 0     | 0                  | 0                | 0         |
| PVcon          | 0     | 0                  | 0                | 0         |

PVcon has detected 21 races, 9 more than that of Falcon. PVcon can detect all races uncovered by Falcon. Nine of the 21 races are hidden by HB edges synchronized by lock operations, and the rate is 42.9%. PVcon pinpoints a new race hidden in pbzip2, shown in Figure 18. fifo means priority scheduling, and mut means lock operation. Falcon is able to detect the races with shared variables of fifo and fifo->empty, but it misses the race with the global variable of allDone, which can be modified by the main thread without any locks protecting and may be read concurrently by other threads. There are multiple lock operations before the read operations on allDone in other threads. In most executions, Thread 1 acquires lock fifo->mut first, and the two critical sections have an HB edge. Then Falcon considers that allDone is synchronized by the lock operations, and does not report any warnings with accesses to allDone. In PVcon, the lock operations will not affect EC if they are not nested with any EC-ordered synchronization operations. Thus, the PVcon has a higher detection coverage than Falcon.

PVcon has localized 9 violations, while Falcon only 4. PVcon can detect the violations synchronized by lock operations, which cannot be detected by Falcon. For instance, there is an atomicity violation synchronized by lock operations in memcached. The two threads concurrently modify a shared
variable \( i \) in two critical sections, and there are no races. However, PVcon can detect this error caused by a non-race conflict with the shared-memory location \((i)\).

The goal of PVcon is to detect non-deadlock concurrency errors. However, in the validation experiment of each conflict, we have localized three deadlocks in \(alet, bbuf\) and \(sync\). There are four non-race conflicts in \(bbuf\) and one conflict in \(sync\). When we reorder their lock acquisition, the deadlocks occur. Thus, PVcon can detect deadlocks.

From the experimental results, we have two observations. First, not all non-race conflicts cause concurrency errors. For instance, PVcon detects several non-race conflicts from \(cherokee\), but these non-race conflicts do not cause any concurrency errors. This is because that concurrency errors
Z. Xu et al.: PVcon: Localizing Hidden Concurrency Errors With Prediction and Verification

FIGURE 18. A race hidden in pbzip2.

are sometimes application-dependent. Second, certain conflicts are more complicated, with a combination of both races and non-race conflicts. In cherokee, pbzip2, bzip2smp, streamcluster, and thread-pool, for example, some conflicts classified by PVcon as races also have other concurrency errors with them.

D. COMPARED WITH SCHEDULING TEST TECHNIQUE

This study compares our technique with the scheduling test technique. We compare PVcon with the state-of-art scheduling test technique, Maple [8]. In experiments, we find that it is time-consuming to implement scheduling test technique. We choose six programs. Each program has been fully tested with Maple and our PVcon.

Table 3 shows the experimental results. The results indicate two findings. First, scheduling technique needs to run programs much more times than PVcon taking more time. Second, though scheduling technique tries its best to trigger as many interleavings as possible, PVcon can detect more concurrency errors than the scheduling technique. Maple applies HB to identify interleavings and tests each program to go through the interleaving patterns. As the HB is too strict, Maple only locates two concurrency error, while PVcon detects 17.

TABLE 3. Concurrency errors and running times.

| Programs    | Detected concurrency errors | Running times |
|-------------|-----------------------------|---------------|
|             | Maple | PVcon | Maple | PVcon |
| aget        | 1     | 3     | 30    | 4     |
| pbzip2      | 1     | 4     | 13    | 4     |
| bzip2smp    | 0     | 5     | 793   | 8     |
| pscans      | 0     | 1     | 258   | 3     |
| stateful    | 0     | 1     | 17    | 3     |
| swarm       | 0     | 3     | 386   | 4     |

TABLE 4. Benchmark metrics: number of threads (#Thrd), number of code lines (#L), the amount of errors (#FE) and number of actual errors (#AE).

| Programs       | #Thrd | #L | #FE | #AE |
|----------------|-------|----|-----|-----|
| account_bad.c  | 2     | 49 | 3   | 3   |
| arithmetic_prog_bad.c | 2 | 82 | 2   | 2   |
| carter_bad.c   | 4     | 43 | 1   | 1   |
| circular_buffer_bad.c | 2 | 109 | 7   | 7   |
| queue_bad.c    | 2     | 153| 4   | 4   |
| sync01_bad.c   | 2     | 64 | 1   | 0   |
| sync02_bad.c   | 2     | 39 | 2   | 2   |
| token_ring_bad.c | 2 | 56 | 1   | 1   |
| twostage_bad.c | 9     | 128| 1   | 1   |
| wronglock_bad.c | 7    | 111| 1   | 1   |
| race-1_1-join_true-unreach-call.c | 1 | 58 | 1   | 1   |
| bigshot_p_false-unreach-call.c | 2 | 35 | 2   | 2   |
| tib_bench_false-unreach-call.c | 2 | 44 | 2   | 2   |
| fib_bench_longer_false-unreach-call.c | 2 | 44 | 2   | 2   |
| lazy01_false-unreach-call.c | 3 | 51 | 1   | 1   |
| stateful01_false-unreach-call.c | 2 | 56 | 2   | 2   |
| qw2004_false-unreach-call.c | 1 | 60 | 1   | 1   |
| half_sync.c    | 2     | 22 | 1   | 0   |
| no_sync.c      | 2     | 21 | 1   | 0   |

E. COMPARISON WITH OTHER TECHNIQUES

Alves E H et al. [52] propose a new method to locate faults in concurrent programs based on bounded model checking and serialization techniques. The main novelty of the method is to reproduce wrong behavior in sequential versions of concurrent programs. In their paper, they analyze the counterexamples generated by the model checker into the new instrumentation sequence program and searched for the diagnostic value corresponding to the actual circuit in the program. This method helps to improve the debugging process of concurrent programs because it tells which line should be corrected and which values can be successfully executed. The experimental results are shown in the table 4.

The benchmarks include the same programs for evaluating ESBMC [53], programs for concurrent C programs, and other programs extracted from the SV-COMP 2016 concurrency suite, available on the ESBMC website (http://esbmc.org/benchmarks/ejss2016.zip). Experimental results show that this method can detect 33 actual errors from 19 programs. Table 2 shows the results that PVcon has located 33 concurrency errors. Table 2 shows the results of 33 concurrent errors found by PVcon from 13 programs. The experimental results show that our results have some advantages.

VI. CONCLUSION

We have proposed a new relation, elastic-causalties (EC) to uncover conflicts in concurrent programs. Based on this relation, an innovative and maximal predictive detection approach, PVcon, is presented to detect concurrency errors. The key idea of PVcon is to identify and build EC for accesses to shared-memory locations based on synchronization operations. A novel data structure, EC-frame, has been proposed to represent EC. PVcon has been implemented to evaluate its effectiveness and efficiency with experiments of detecting concurrency errors from 22 real-world benchmarks. It has been shown that PVcon can effectively and efficiently detect more concurrency errors than the pattern analysis technique and scheduling test technique.
REFERENCES

[1] G. Liu, “Some complexity results for the soundness problem of workflow nets,” IEEE Trans. Services Comput., vol. 7, no. 2, pp. 322–328, Apr. 2014.

[2] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: A comprehensive study on real world concurrency bug characteristics,” in Proc. ASPLOS, 2008, pp. 329–339.

[3] Z. Yu, Y. Zhao, and Y. Zhao, “Concurrency: A concurrency bug avoier based on transparent software transactional memory,” in J. Int. Parallel Process., vol. 48, no. 1, pp. 32–60, Feb. 2020.

[4] B. Richards and J. R. Larus, “Protocol-based data-race detection,” in Proc. 24th ACM Symp. Operating Syst. Princ. SOSP, 2013, pp. 305–319.

[5] M. Pruvolovic, “CORD: Cost-effective (and nearly overhead-free) order- recording and data race detection,” in Proc. 12th Int. Symp. High-Perform. Comput. Archit., Feb. 2006, pp. 232–243.

[6] C. Flanagan and S. N. Freund, “Fastack: Efficient and precise dynamic race detection,” in Proc. PLDI, 2009, pp. 121–133.

[7] S. Park, R. Vuduc, and M. J. Harrold, “UNICORN: A unified approach for localizing non-deadlock concurrency bugs,” Softw. Test., Verification Rel., vol. 25, no. 3, pp. 167–190, May 2015.

[8] B. Liu, Z. Qi, B. Wang, and R. Ma, “Pinse: Precise isolation of concurrency bugs via delta triaging,” in Proc. IEEE Int. Conf. Softw. Maintenance Evol., Sep. 2014, pp. 201–210.

[9] Z. Wu, K. Lu, X. Wang, and X. Zhou, “Collaborative technique for concurrency bug detection,” in Int. J. Parallel Program., vol. 43, no. 2, pp. 260–285, Apr. 2015.

[10] Y. He, W. Wu, and Y. Chen, “An efficient method for detecting concurrency errors in object-oriented programs,” Sci. China Inf. Sci., vol. 55, no. 12, pp. 2774–2784, Dec. 2012.

[11] S. Cohen and E. D. Demsky, “Automatic memory reclamation for lock-free data structures,” in Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl. OOPSLA, 2015, pp. 260–279.

[12] J. Huang, “Scalable thread sharing analysis,” in Proc. 38th Int. Conf. Softw. Eng. ICSE, 2016, pp. 1097–1108.

[13] P. Ou and B. Demsky, “Checking concurrent data structures under the C/C++11 memory model,” in Proc. 22nd ACM SIGPLAN Symp. Pract. Prac. Parallel Program., Jan. 2017, pp. 45–49.

[14] S. M. Melo, S. D. R. S. de Souza, F. S. Sarmanho, and P. S. L. D. Souza, “Contributions for the structural testing of multithreaded programs: Coverage criteria, testing tool, and experimental evaluation,” Softw. Qual. J., vol. 26, no. 3, pp. 921–959, Sep. 2018.

[15] I. J. Hayes, L. A. Minicke, K. Winter, and R. J. Colvin, “A synchronous program algebra: A basis for reasoning about shared-memory and event-based concurrency,” Formal Aspects Comput., vol. 31, no. 2, pp. 133–163, Apr. 2019.

[16] M. Bättig and T. R. Gross, “Synchronized-by-Default concurrency for shared-memory systems,” in Proc. 22nd ACM SIGPLAN Symp. Prac. Prac. Parallel Program., Jan. 2017, pp. 299–312.

[17] G. Jin, L. Song, W. Zhang, S. Lu, and B. Llibrit, “Automated atomicity-violation fixing,” in Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Design Implement. PLDI, 2011, pp. 389–400.

[18] G. Jin, W. Zhang, D. Beng, B. Llibrit, and S. Lu, “Automated concurrency-bug fixing,” in Proc. OSDI, 2012, pp. 221–236.

[19] S. Rosario, A. Benveniste, S. Haar, and C. Jard, “Probabilistic QoS and soft contracts for transactional Web services orchestrations,” IEEE Trans. Services Comput., vol. 1, no. 4, pp. 187–200, Oct. 2008.

[20] Y. Xiao and S. Urban, “Using rules and data dependencies for the recovery of concurrent processes in a service-oriented environment,” IEEE Trans. Services Comput., vol. 5, no. 1, pp. 59–71, Jan. 2012.

[21] P. Parizek and O. Hltat, “Fast detection of concurrency errors by state space traversal with randomization and early backtracking,” Int. J. Softw. Tools Technol. Transf., vol. 21, no. 4, pp. 365–400, Aug. 2019.

[22] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, “Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing,” in Proc. 27th ACM Symp. Operating Syst. Princ., Oct. 2019, pp. 162–180.

[23] N. Zhang, C. Sun, and S. Lu, “Commem: Detecting severe concurrency bugs through an effect-oriented approach,” in Proc. ASPLOS, 2010, pp. 179–192.

[24] C. 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 Proc. PLDI, 2005, pp. 1–11.

[25] E. D. Demsky, “Virtual function tracing and global views of distributed systems,” in Parallel and Distributed Alg.-Rithms. Santa Margherita Ligure, Italy: IEEE, 1988, pp. 215–226.

[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: Characterization and methodological considerations,” in Proc. 22nd Annu. Int. Symp. Comput. Archit., 1995, pp. 24–36.

[27] J. Yu and S. Narayanasamy, “A case for an interleaving constrained shared-memory system,” in Proc. 56th Annu. Int. Symp. Comput. Archit. ISCA, 2009, pp. 325–336.
[51] Y. Yang, *Efficient Dynamic Verification of Concurrent Programs*. Lake City, UT, USA: Univ. Utah, 2009.
[52] E. H. D. S. Alves, L. C. Cordeiro, and E. B. D. L. Filho, “A method to localize faults in concurrent C programs,” *J. Syst. Softw.*, vol. 132, pp. 336–352, Oct. 2017.
[53] L. Cordeiro, B. Fischer, and J. Marques-Silva, “SMT-based bounded model checking for embedded ANSI-C software,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 4, pp. 957–974, Jul. 2012.

**ZHENWU XU** received the B.S. degree from the Department of Information Engineering, Henan Institute of Science and Technology (HIST), China, in 2018. He is currently pursuing the M.S. degree with Hubei Minzu University, Enshi, China. His research interests include software development, security in the areas of cloud computing, and network security testing.

**JINAN SHEN** received the Ph.D. degree from the Huazhong University of Science and Technology (HUST), Wuhan, China. He is an Associate Professor of computer science with Hubei Minzu University, Enshi, China. His research interest includes security in cloud computing, focusing on privacy-preserving in the cloud.

**PENG LUO** received the Ph.D. degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2019. He is currently an Engineer with Huawei Technologies Company Ltd. His research interests include software security and reliability in cloud computing.

**FANG LIANG** received the M.S. degree in computer science and technology from Wuhan University, China, in 2012. She is currently an Associate Professor of computer science with Hubei Minzu University, Enshi, China. Her research interest includes security in cloud computing.

***