Concurrent Predicates: A Debugging Technique for Every Parallel Programmer

Justin E. Gottschlich
Intel Corporation
justin.e.gottschlich@intel.com

Gilles A. Pokam
Intel Corporation
gilles.a.pokam@intel.com

Cristiano Pereira
Intel Corporation
cristiano.l.pereira@intel.com

Youfeng Wu
Intel Corporation
youfeng.wu@intel.com

Abstract—To reduce the complexity of debugging multithreaded programs, researchers have developed many techniques that automatically detect bugs that arise from shared memory errors. These techniques can identify a wide range of bugs, but it can be challenging for a programmer to reproduce a specific bug that he or she is interested in using such techniques. This is because these techniques were not intended for individual bug reproduction but rather an exploratory search for possible bugs.

To address this concern we present concurrent predicates (CPs) and concurrent predicate expressions (CPEs), which allow programmers to single out a specific bug by specifying the schedule and program state that must be satisfied for the bug to be reproduced. We present the recipes, that is, the mechanical approaches used by software developers to debug their sequential programs stems from the fact that multicore processors have outgrown the application of traditional software debug methods on uniprocessor systems. On these systems, two popular approaches used by software developers to debug their sequential programs are breakpoints and assertions. With breakpoints, a programmer can suspend a program execution when and where he or she wants to, while assertions enable the programmer to check assumptions about a program execution at runtime. Note that, in both cases, these are powerful features not meant to automatically detect bugs, but rather to assist a programmer in minersing the root cause of a known bug, or testing a hypothesis about some unwanted program behavior. Unfortunately, these two features do not have counterparts on multicore systems, yet, they are critical to professional software development environments where the majority of programmers are assigned known bugs that they must first reproduce and then fix.

The main challenge applying these constructs to multithreaded applications is that a programmer must consider more than one thread at a time to understand when and where a program execution must be suspended or he or she must infer the program properties that need to be checked at runtime. The combinatorial state space explosion of multithreaded programs complicates this task. Yet, a more fundamental limitation is the lack of a mechanism for controlling thread schedules that would enable a programmer to enforce his or her choices.

In this paper, we introduce a new programming construct for debugging parallel programs called concurrent predicates. Concurrent predicate is to parallel programmers what breakpoints and assertions are to sequential programmers: an instrument for developers to use to explore their bug hypotheses. We introduce a programming construct for concurrent predicate and show how, using a set of recipes, it can be used to root-cause and reproduce hard to detect concurrency errors.

This paper makes the following technical contributions:

1) We present concurrent predicates (CPs) and concurrent predicate expressions (CPEs), two novel synchronization primitives that can be used to reproduce concurrency violations, which have different programming interfaces.

2) We present the concepts of non-essential thread interference and self stability and demonstrate how they impact the ability and reliability of bug reproduction, respectively.

3) We present recipes for the mechanical use of CP and CPE to reproduce a variety of concurrency bugs.

4) Using our recipes, we show real-world uses of CP and how we reproduced and fixed three previously known but unresolved multithreaded bugs in TBoost.STM, an open source software transactional memory library.

I. INTRODUCTION

A major obstacle for software developers writing parallel programs stems from the fact that multicore processors have outgrown the application of traditional software debug methods on uniprocessor systems. On these systems, two popular approaches used by software developers to debug their sequential programs are breakpoints and assertions. With breakpoints, a programmer can suspend a program execution when and where he or she wants to, while assertions enable the programmer to check assumptions about a program execution at runtime. Note that, in both cases, these are powerful features not meant to automatically detect bugs, but rather to assist a programmer in minersing the root cause of a known bug, or testing a hypothesis about some unwanted program behavior. Unfortunately, these two features do not have counterparts on multicore systems, yet, they are critical to professional software development environments where the majority of programmers are assigned known bugs that they must first reproduce and then fix.

II. RELATED WORK

To reduce the complexity of debugging multithreaded programs, researchers have developed innovative ways to automatically detect concurrency violations [4], [10], [13], [17]. Research in this area has generally focused on systematic model checking to exhaustively test all possible thread interleavings [2], [12], [21] or random testing to overcome impracticality issues caused by state space explosion [5], [19]. Although many bugs may be found by these automated systems, it can be challenging for a programmer to reproduce a specific bug that he or she is interested in. This is because these
systems were not intended for individual bug reproduction, but rather for exploratory search of concurrency errors. Yet, reliable and efficient bug reproduction is usually the first step to fixing software defects. Unfortunately, even with record and replay systems [11], [15], [16], reproducing a specific bug can only be achieved if the bug manifests in the recorded execution, which relies on luck or a way to force the bug’s root cause and effect to be satisfied, that is, the necessary conditions that are required to reproduce the bug.

To address these concerns, Schwartz-Narbonne et al. propose parallel assertions, which allows the programmer to embed traditional-like assertions within the context of parallel programs that fire if a limited range of conditions or invariants in one thread are violated by another [18]. Although parallel assertions do capture specific concurrency-related events, they do not capture the thread and, more importantly, the instruction triggering the event. Therefore, parallel assertions fall short of revealing the root cause of the concurrency bugs they trigger.

Park and Sen address this with their concurrent breakpoint system which captures both the cause and the effect of a given concurrency violation [14]. Unfortunately, concurrent breakpoints have not been demonstrated to scale beyond two threads, provide no mechanism to prevent threads that do not contribute to a specific concurrency bug from interfering with bug reproduction, and do not provide a guarantee that once a concurrency bug is triggered that its necessary state will remain intact once the breakpoints fire. Lastly, it is not clear if concurrent breakpoints can be integrated directly into a debugger.

III. CP EXAMPLE

At the highest level, CP and CPE can be thought of as synchronization primitives that enable a programmer to synchronize multiple threads based on a sequence of events. Because CP is the foundation upon which CPE was developed, we begin by explaining CP and then discuss CPE in Section VI.

Any number of CPs can be active within a program at a time, enabling programmers to investigate multiple bugs within the context of a single execution. Alternatively, multiple groups of CPs can be used to investigate the same bug from different vantage points. CP captures both the effect and root cause of multithreaded bugs and increases their likelihood of occurrence by altering the execution schedule. CP also provides specific timing guarantees, called self-stability, in which the predicates of the program will remain satisfied, enabling deterministic bug reproduction within certain constraints. Furthermore, CP provides a mechanism to prevent non-essential thread interference, those threads that do not contribute to the reproduction of a bug but can obfuscate it. Preventing non-essential thread interference is critical to reproducing complex, real-world multithreaded bugs where interfering threads can often drastically reduce the likelihood of reproducing a bug.

To demonstrate the usefulness of CP, consider the code shown in Figure 1(A) which assumes $y$ is a shared variable between Threads 1 and 2 and the program has the invariant of $y \neq 0$. Because of an atomicity violation in Thread 1 (between its critical sections), Thread 2 might see $y = 0$, resulting in a divide by zero exception.

Without a mechanism to alter each thread’s schedule and a predicate to tell us when such alteration is needed (e.g., in this case when $y = 0$), the atomicity violation that leads to the divide by zero exception can be challenging to reproduce. This is especially true if calculate() returns a wide range of numbers where the likelihood of 0 is low. With CP, as shown in Figure 1(B), we can reproduce this bug with significantly improved probability (e.g., CP improved this bug’s probability from $\approx 5\%$ to $100\%$ as shown in Table I, bug name “CP-4b”). Improving the probability of bug reproduction significantly reduces the challenge for parallel programmers to fix them, because, when a bug occurs on (nearly) every execution, the programmer can cyclically debug the program, learning more about the bug with each subsequent execution.

IV. THE CP CONTROL STRUCTURE

In this section we present the CP control structure. We then show how it is applied to reproduce the divide by zero bug initially shown in Figure 1(B). However, this time, we do not omit any details of the CP control structure. On the other hand, we do not show how CP can manage non-essential thread interference. We delay explaining that additional complexity until Section V.

The CP control structure syntax is shown in Figure 2. The pre compound statement is executed when the CP control structure is entered and prior to evaluating its predicate. Likewise, the post compound statement is executed just before exiting the CP control structure. Both the pre and post compound statements are always executed exactly once regardless of the CP’s predicate value.

Alternatively, if_satisfied is only executed if the user supplied predicate, predicate in Figure 2, is true along with

---

1Some details of the CP control structure have been omitted to simplify Figure 1(B)’s example. They are expanded upon in Section IV.
some additional condition checks (details forthcoming). The else compound statement is only executed if predicate is never true. For instance, if a CP’s predicate is \( y = 0 \) but the program always holds \( y \neq 0 \), then the CP’s else will eventually be executed, once its retryTime has been exceeded. At most, only one of these two compound statements, if_satisfied or else, will be executed when executing any CP control structure.

The pre, if_satisfied, and else compound statements are serialized with respect to all concurrently executing CPs. For example, if thread, \( T_1 \) is executing the pre portion of a CP, no other threads may execute either the pre, if_satisfied, or else portions of another CP until \( T_1 \) completes its execution of pre. This behavior is critical to enabling self-stability, as described in more detail in Section V-B.

For completeness, the following list includes the definitions of each of CP’s parameters. However, only the state, control, and predicate parameters are essential to understanding the examples presented in this paper.

- **state**: an instance of CpState, highlighted as \( s \) in Figure 3, that must be created for each multithreaded bug and shared across the CPs that are necessary to reproduce the bug. state maintains global state to track the number of CPs that must be satisfied concurrently to trigger the if_satisfied compound statement. state has the following fields:
  - \( N \): the first field of CpState’s constructor, which, along with the to_satisfy field, is used to determine if verify(), shown in Algorithm 1, returns true.
  - to_satisfy: the second field of CpState’s constructor, which is a conditional operator (e.g., =, <, >, \( \neq \)) that is applied to \( N \) and used in Algorithm 1.
  - satisfied: a set of thread IDs which have a predicate held as true. This is internally updated by the CP system as shown in Algorithm 1.

- **priority**: a non-unique priority of the CP, where 0 is the highest priority. When multiple CPs’s if_satisfied are to be executed, the CP with the highest priority goes first. In the event of a tie, there is no ordering guarantee.

- **control**: a boolean that, if true, along with predicate and verify(), will trigger if_satisfied to execute. If control is false, and the value of predicate remains true, the CP will wait until retryTime has been exhausted and then execute else.

- **retryTime**: the minimum number of milliseconds a CP will be retried before exiting when predicate and verify() remain false. A CP is guaranteed to wait at least as long as retryTime if predicate and verify() have not yet returned true, but it may wait longer.

- **retryIfFalse**: a boolean that, if true, will cyclically re-evaluate its predicate even when predicate is false. Otherwise, the CP will exit its control structure as soon as predicate is found to be false.

**Algorithm 1 Verify**

1. procedure verify(st)
2. \( S \leftarrow st.\text{satisfied}.\text{size} \)
3. \( N \leftarrow st.N \)
4. if \( st.\text{to}\_\text{satisfy} \equiv \text{no}\_\text{predicates} \land S \equiv 0 \) then return true
5. else if \( st.\text{to}\_\text{satisfy} \equiv \text{less}\_\text{than} \land S < N \) then return true
6. else if \( st.\text{to}\_\text{satisfy} \equiv \text{greater}\_\text{than} \land S > N \) then return true
7. else if \( st.\text{to}\_\text{satisfy} \equiv \text{equals} \land S \equiv N \) then return true
8. else if \( st.\text{to}\_\text{satisfy} \equiv \text{not}\_\text{equals} \land S \neq N \) then return true
9. else if \( st.\text{to}\_\text{satisfy} \equiv \text{active}\_\text{predicates} \land S \equiv st.\text{in}\_\text{predicates} \) then return true
10. end if
11. return false

A. Reproducing the Divide by Zero Bug With CP

Now that the individual pieces of the CP control structure have been discussed, we can provide a complete CP solution for the original divide by zero bug that we showed in Figure 1.

Figure 3 illustrates how the CP control structure, including all of its necessary parameters, are used to reproduce the divide by zero bug. The programmer first creates a shared instance of CpState and then adds CP control structures (highlighted in Figure 3) to both Threads 1 and 2 to control the forward progress of the program based on the state necessary to reproduce the bug (i.e., \( y = 0 \)). Once in place, the programmer executes the program.

In Thread 1’s execution, if calculate() (abbreviated as calc()) returns 0, then the first part of the bug, its root cause, is captured. In this case, when Thread 1’s CP is invoked its execution is stalled for up to the specified retryTime. This is because although \( y = 0 \), the CP’s associated state, \( s \), is not satisfied. \( s \) is only satisfied when two CP predicates are concurrently true, as defined by CpState \( s(2, \text{equals}) \). Next, we assume Thread 2 begins execution, which immediately enters its CP control structure. Thread 2’s CP predicate...
is immediately satisfied, as is its associated state, s. Because Thread 2’s control parameter is true, it then executes the if_satisfied portion of its CP control structure. These instructions then reproduce the divide by zero bug and the program terminates.

V. CP DESIGN AND RUN-TIME ALGORITHM

CP has three variants: general (cp()), serial (cp_serial()), and serial(id) (cp_serial(id)). These variants are meant to be used together to reproduce complex heisenbugs that cannot be easily reproduced by using only one. For our experiments, the most commonly used type, referenced in Figure 3, is the general CP (cp). Its high-level algorithm is described in Algorithm 2. We say that the general CP is fully concurrent because an unbounded number of threads can be concurrently active in it. Both the serial and serial(id) versions of CP do not exhibit this behavior, which is the key difference between them and the general CP.

The serial CP limits its concurrent execution to one thread at a time, while serial(id) limits its concurrent execution to one thread per unique id. By constraining the amount of possible concurrency, the serial CPs aim to reduce a bug to its most basic components and eliminate additional and non-essential thread contention. Due to space limitations, we only include the algorithmic details for the general CP.

A. Managing Non-Essential Thread Interference

A feature that distinguishes CP from prior work is its ability to manage non-essential thread interference by limiting concurrency for certain regions of code that could otherwise interfere with the system’s ability to reproduce a bug. This is illustrated in Figure 4, which revisits the divide by zero bug presented in Figure 1 of Section I, where Thread 1 is replaced by Threads 1 through N – 1 and Thread 2 is replaced by Thread N. Without a mechanism to prevent non-essential thread interference, this minor modification to the problem space can result in a near impossibility of bug reproduction when N is large. In general, the greater N, the less likely the bug will occur due to thread interference. This is counterintuitive, as one might expect that the more threads that are present in a program, the more likely an execution will produce a multithreaded bug. However, in this case, the additional threads only interfere with bug reproduction, greatly reducing its frequency without some mechanism to manage the interference.

CP manages this interference by restricting threads 1 through N – 1 to serial execution by using the cp_serial control structure. The operations that could interfere between threads 1 through N – 1 are placed within the pre and post sections of the cp_serial control structure, thereby eliminating their potential concurrent interference. Thread N’s code is managed by the fully concurrent cp, because (i) its code can only be accessed by one thread and (ii) even if multiple threads could execute the code, because of its read-only nature, such concurrent executions would not interfere with one another. Finally, because cp and cp_serial are not serialized with respect to each other, the bug is still reproducible once one cp_serial is satisfied from Threads 1 through N – 1 and one cp is satisfied within Thread N.

Algorithm 2 The CP Run-Time Algorithm

Require: state is shared memory for all threads.
Require: threadId is the ID of the active thread.

1: procedure CP(state, priority, control, retryTime, retryIfFalse, predicate)
2:     Lock state.mutex
3:     executeIfSatisfied ← false
4:     Execute pre-execution operations
5:     Insert threadId into state.in_predicate
6:     if control then
7:         Insert (threadId, priority) into state.priorities
8:     end if
9:     Unlock state.mutex
10:    while retryTime > 0 do
11:        beginTime ← Clock()
12:        Lock state.mutex
13:        if predicate then
14:            Insert theadId into state.satisfied
15:        else
16:            Remove theadId from state.satisfied
17:        end if
18:        if verify(state) ∧ predicate then
19:            executeIfSatisfied ← true
20:        if ¬control then
21:            No-Op
22:        else if control ∧ priority ∈ state.top_priority then
23:            Remove theadId from state.satisfied
24:        Remove theadId from state.in_predicates
25:        Remove (theadId, priority) from state.priorities
26:        Execute if_satisfied operations
27:        Unlock state.mutex
28:        Break
29:     end if
30:    end if
31:    Unlock state.mutex
32:    if ¬retryIfFalse then
33:        Break
34:    end if
35:    if SLEEP(1)
36:        endTime ← Clock()
37:        retryTime ← retryTime − (endTime − beginTime)
38:    end while
39:    Lock state.mutex
40:    if ¬executeIfSatisfied then
41:        Execute else operations
42:    end if
43:    if end
44:        Remove theadId from state.satisfied
45:        Remove theadId from state.in_predicates
46:        Remove (theadId, priority) from state.priorities
47:        Unlock state.mutex
48:        Execute post-execution operations
49:     end procedure

B. Self Stability

A key characteristic of our CP design is in the self stability it promises. For CP, the self stability property ensures the resulting predicate and verify() values, whether they are true or false, will not be perturbed by another concurrently executing CP for the duration of time in which the CP’s respective if_satisfied or else execution occurs. The notion of self stability that we use has been lifted from Dinsdale-Young et al. [3].

Informally, Dinsdale-Young et al. define self stability as an execution property that ensures that once a predicate condition has or has not been satisfied it remains in that state for operations that are dependent upon it. The way this applies to CP is that the returned predicate state and its associated
if_satisfied or else operations are free from concurrently executing CP interference until they have completed their execution.

Dinsdale-Young et al. use self stability in a theoretical setting for their formalism of a disjoint logic. Our use of self stability is notably different, although the notion is the same. We use it to guarantee that predicates that have captured a precise program state are preserved until the if_satisfied and else that rely on such predicates are executed without predicate perturbation; that is, without the predicates’ evaluation changing between the time they were initially checked and the time the final if_satisfied or else operation of the CP control structure is executed.

By ensuring this limited form of self stability, concurrency bugs can be deterministically reproduced, within certain constraints, once their associated predicates have been satisfied. Without self stability, approaches like CP can still reproduce concurrency bugs, but cases likely emerge when the state that is required to reproduce a bug is reached and lost before the operations that reveal the effect of the bug have been executed, making such approaches practically unattractive.

CP’s self stability is implemented in the following manner. Assuming a CP’s control is true, once its predicate and verify() have been satisfied, or they have not been satisfied and the CP has timed out, the CP is given permission to execute its if_satisfied or else operations, respectively. During this time, other CPs that are active, that is, are currently being executed, are prevented from making forward progress. This prevents the active CPs from changing the predicate state in which the original CP’s if_satisfied or else operations rely.

It is important to note that this guarantee does not extend beyond CP’s execution. In particular, threads whose executions are outside the lexical scope of a CP control structure may still alter the program’s state such that a predicate’s state can change before the execution of a subsequent if_satisfied or else of an active CP. However, in our experience, such perturbations are easily managed by adding CPs to the program locations that might mutate the shared data that is used within a predicate. When following this technique, we have not encountered any self stability issues that have prevented us from reproducing a concurrency bug.

VI. CONCURRENT PREDICATE EXPRESSIONS

Our experience with CPs has been that they can reproduce any type of concurrency bug, yet, they can be challenging to use at first. To address this, we created CPE. CPE is notably easier to use, as we show in this section, but it currently has limited functionality compared to CP.

Our eventual goal is to automate all aspects of CP with CPE to bring CP’s debugging power to every parallel programmer. In this section, we demonstrate that the general framework of CP can be reproduced with CPE with significantly reduced complexity. However, there are still some notable features that have not yet been integrated into the framework, such as automated debugger support and user-controlled timing constraints. We are working to address these missing pieces of functionality.

A. Revisiting Divide By Zero

To demonstrate this simplification, let us revisit the divide by zero bug shown initially in Figure 1 and then again with a complete CP solution in Figure 3. Without CP, reproducing this divide by zero bug is challenging. Yet, even with CP, the programmer is burdened with a certain degree of complexity that is embedded within the CpState and the control structure of CP.

Specifically, to reproduce the divide by zero bug using CP, the programmer first constructs a shared CpState instance for the concurrency bug he or she is investigating. The programmer then specifies how many predicates must be satisfied to reproduce the bug, and then he or she adds the CP control structures to the program that check the predicate and include the proper compound statement clauses.

While CP can reproduce this bug, non-expert parallel programmer may have difficulty using its interface. CPE simplifies this, by moving much of the programming complexity to its runtime algorithm and its data structure, the CpeSequence of Figure 5, that captures the happens-before ordering specified by the programmer. To demonstrate this, consider the necessary steps to reproduce the same divide by zero bug using CPE as shown in Figure 5.

![Fig. 5. Using CPE to Reproduce the Divide by Zero Exception.](image-url)
The execution of the CPE calls shown within Figure 5 are handled in the following manner. Because the programmer specified \(p1 <\& p2\) for this bug (details forthcoming), once CPE(seq, "p1", \(y = 0\)) is satisfied the CPE system will wait for a bounded amount of time for the CPE(seq, "p2", \(y = 0\)) to be satisfied. If both are satisfied concurrently, with \(p1\) being satisfied first, the CPE system will trigger a breakpoint allowing the programmer to inspect the state of the program. A key difference between CPE and CP is that CPE does not allow the programmer to specify a particular type of behavior once the concurrency bug is triggered. Instead, it forces the program to behave in an particular fashion. For our purposes, triggering a breakpoint or outputting some logged behavior has been the most useful.

Although there are three steps to reproduce the divide by zero bug using CPE, the program logic (lines of code (LOC) and function call complexity) compared to CP is notably reduced. The LOC is reduced from 17 (some of which was omitted in the original example) when using CP to three when using CPE. The original call to CP required six parameters, the revised CPE call only requires three. It is important to note that the simplification provided by CPE is achieved by forfeiting some programmer control provided by the original CP design. In short, CPE does not invalidate CP. Instead, we consider both approaches to be useful for different contexts, as explained above.

B. The CPE Happens-Before Operators

The key to CPE’s reduced complexity as compared to CP is in the way it abstracts away the happens-before ordering of events away from its programmed interface. In particular, rather than embedding such ordering within its control structure, as is done in CP, CPE specifies an expressions grammar to denote the happens-before ordering that is sufficient to reproduce a concurrency bug as is shown in Step 1 in Figure 5. The operators we define for the CPE grammar are the following (where \(p1\) and \(p2\) are predicates and a satisfied predicate indicates the predicate is true):

- \(p1 <\& p2\) - \(p1\) is satisfied and then \(p2\) is satisfied
- \(p1 <\& p2\) - \(p1\) satisfied and stays satisfied until \(p2\) satisfied
- \(p1 <\& p2\) - \((p1 <\& p2)\) or \((p2 <\& p1)\)
- \(p1 \| p2\) - \(p1\) satisfied or \(p2\) satisfied
- \(p1 \# N\) - \(p1\) satisfied \(N\) times
- \(p1 \# N\) - \(p1\) satisfied \(N\) times concurrently

Furthermore, CPE has both serial (CPE_serial()) and serial(id) (CPE_serial(id)) interfaces, like CP, which help it manage non-contributing thread interference as is demonstrated in Figure 4. These interfaces behave identically to their CP counterparts discussed in Section V.

C. Debugger Integration

In addition to reducing the complexity over CP, CPE has the potential for debugger integration. Specifically, once the CPE calls have been inserted into the user’s software, Steps 2 and 3 in Figure 5, the debugger can be extended to take as input the CPE’s expression. By integrating the expression into the debugger, the programmer can change the happens-before ordering without changing the program code. Therefore, multiple schedules can be attempted to reproduce a specific bug without recompiling or rebuilding the software.

We believe such an approach will be useful as it will help explore the many different ways to reproduce the same bug with minimal effort. In particular, our experience has shown that once the key event points of a bug have been found, identifying the precise happens-before ordering to trigger the concurrency bug is not always straightforward. Also, in some cases many valid schedules exist to reproduce the same concurrency bug. Therefore, being able to explore many possible happens-before orderings with the same set of events has helped us expose latent concurrency bugs that were not observed with the initial happens-before ordering that was specified. Secondly, such an approach has helped us reproduce the same bug in ways that we initially did not consider, providing us with a more complete view of the existing schedules and states that cause it, which have helped us design more complete bug fixes.

VII. Recipes

To successfully use CPs and CPEs to reproduce concurrency bugs, the programmer must determine the correct location for the CPs or CPEs in the program’s source code. In this section, we present the CP and CPE recipes, that is, the mechanical processes that we use with CP and CPE, to reproduce three types of concurrency violations: data races, atomicity violations, and deadlocks. To simplify our recipes discussion, we restrict ourselves to using CPEs. However, the same bugs can be reproduced by replacing the CPEs with their equivalent CPs.

A. A Sketch to Use CPEs

Parallel programs, like any other class of programs, have defects that are generally found by observing unwanted effects. When we first begin investigating a concurrency bug with CPE, we place a CPE just before the location of the bug’s effect, what we refer to as the effect CPE, as is done in Thread 2 of the divide by zero example of Figure 5. We refer to the thread that contains the effect CPE as the effect thread. We start with the effect CPE because the bug’s effect location is generally known when the bug is first observed. We place the effect CPE before the bug’s effect operation because it enables us to see the program state before triggering the bug effect. Alternatively, placing the effect CPE after the bug effect would be unwise because once the bug effect is triggered the program may exhibit side-effects that crash the program or cause undesirable and undefined behavior.

It can be challenging to identify the root cause of a concurrency bug, because it is generally not known when the bug is first observed. We refer to the CPE that captures the root cause of the bug the root cause CPE and its thread the root cause thread. For most bugs, we use the following approach to find their root cause CPE. First, root causes are always write operations. Therefore, when searching for the root cause CPE, read operations can be discarded as candidates. Second, because the root cause behavior is needed to reproduce a bug, a root cause CPE candidate is always placed after the candidate
operation has been performed, unlike effect CPEs which are always placed before the effect operation.

Lastly, our experience with CPE has shown us that when the CPEs for a given bug are placed at the correct locations, they usually trigger the bug within the first few executions of the code. We use this information in its contrapositive form; that is, if we cannot reproduce a bug after several executions, it is likely that our root cause CPE is not correct. It is important to note that this is a rule of thumb, not a logical truth. In particular, CPEs can emit false negatives. As such, when a set of CPEs fail to reproduce a bug, it does not necessarily indicate the bug cannot be triggered from such a set, it only means that it has not done so thus far.

B. Data Race

In our experience, real-world data race concurrency bugs occur when either (i) the programmer is unaware of the data race or (ii) he or she incorrectly believes it is benign [1]. For benign cases, CPE is not necessary as the programmer is already aware of the data race location. For cases where the root cause of the data race is unknown, we use the following CPE recipe as highlighted in Figure 6.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig6.png}
\caption{CPE Recipe to Reproduce Data Races.}
\end{figure}

First, for data races to exist they must have a write operation associated with them to cause an inconsistent state. For such an inconsistent state to be observed, the data race must have a read operation that makes the state observable. As described in Section VII-A, we begin our placement of CPEs with the effect CPE, which, in this case, is the read operation that makes the data race observable. This read operation is likely to be known by the programmer, otherwise the data race would not be observable.

Once the effect CPE is in place, the programmer uses a mechanical process to place CPEs to find the corresponding root cause write operation. For any program, there are a finite number of write operations to a shared variable that cause a data race. When attempting to locate the root cause CPE, we use the technique described in Section VII-A, where if the bug does not surface after several executions trying one root cause CPE, we move on to the next location. In our experience using CPE to reproduce extremely rare data races in several real-world applications, this approach has never required more than a fifteen executions to reproduce a data race. \footnote{In the case where fifteen executions were required for CPE to reproduce the bug, which was RADBench SpiderMonkey-2, comparatively, existing testing solutions could only reproduce the same bug once out of every 5,000 executions, making CPE’s bug reproducibility more than two orders of magnitude more reliable.}

C. Atomicity Violation

Atomicity violations (AVs) occur when a programmer breaks a single atomic unit of operations across two or more independently synchronized blocks of execution as shown in Figure 7. AVs can occur using any type of synchronization such as locks, transactional memory, monitors, and semaphores, to name a few. In Figure 7, we abstract away the specific type of synchronization by using the sync() and unsync() calls, which indicate that the code between two such points are properly synchronized.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig7.png}
\caption{CPE Recipe to Reproduce Atomicity Violations.}
\end{figure}

One of the properties associated with an AV is that its side effects usually occur only under certain conditions. The divide by zero bug, discussed throughout this paper, is an example of such an AV. It has no negative side effects unless calculate() returns 0. In such a case, if the AV is exposed, that is, Thread 2 can observe \( y = 0 \), the program can crash. All other executions execute without error because the program invariant is not violated. It is because of this that we include pred1() and pred2() in the CPEs placed in Figure 7, which enables the programmer to constrain execution schedules to just the specific ones he or she is interested in. In many real-world cases, these predicates are unique for each CP and CPE, which is why the CP and CPE interfaces allow for a unique predicate to be passed to each of their calls.

Once an AV is observed, the location of the effect CPE is generally known. However, it is important to note that for AVs that are not data races, that is, all access to the shared memory is synchronized, the effect CPE is usually placed within the existing synchronization as is shown in Thread 2 in Figure 7. This ensures that when the CPEs trigger, the buggy program state is guaranteed to occur. If the effect CPE is placed outside of the synchronization boundaries of the effect AV, it is possible for the AV to be elided because the root cause thread may fix the behavior before the effect thread can observe it. Notice, however, that in Figure 3, CP does not need to be placed within the synchronization, because of its self stability guarantees (see Section V-B), which ensure its if_satisfied or else operations will trigger without any active CP changing the current state of its predicates.

Finding the location of the root cause CPE is usually performed via a mechanical trial and error process, as described in Section VII-A. If the AV is synchronized, the root cause CPE must always be placed after the root cause behavior and synchronization, otherwise, the effect CPE might not be able to trigger; that is, the operations necessary to make the AV observable might not be able to execute unless the synchronization mechanism is available to the effect thread.
D. Deadlock

Deadlocks are one of the most common types of concurrency bugs due to the prominent use of locks in multithreaded software. CPEs can help reproduce deadlocks and identify their root cause whether they are known or remain latent in existing software. An example CPE recipe for deadlocks is shown in Figure 8.

Figure 8(B), which is slightly more complex than the prior recipes, shows both the effect and root cause CPEs. Unlike the prior recipes, for deadlocks the effect and root cause CPEs are always sets of two or more CPEs, depending on how many threads are required to reproduce the deadlock. The manner in which these CPEs are added to the program’s source code is entirely mechanical. If the deadlock location is known, the programmer can reliably insert the effect CPEs to the known deadlock location. If it is not known, the programmer can insert effect CPEs based on his or her speculation of the deadlock’s effect. In some cases, even when a deadlock is not yet known to exist, using only effect CPEs, a programmer can reproduce a deadlock. This is because the effect CPEs will change the program’s schedule, enforcing an inconsistent locking order, thereby forcing a deadlock that was previously dormant.

Once the effect CPEs have been correctly placed within the program – that is, they can reliably reproduce the deadlock – the programmer uses a mechanical process to find the root cause CPEs, which includes finding the corresponding inconsistent locking order. Because CPEs cannot emit false positives, once a happens-before sequence of root cause CPEs are triggered, which then subsequently trigger the effect CPEs, the root cause of the deadlock is generally known. In rare cases when trying to reproduce deadlocks, the root cause CPEs may trigger but they may not be the root cause of the bug, because they are subsequently invalidated prior to the effect CPEs. To prevent this from occurring, we use predicates in the effect CPEs to re-validate the intention of the root cause CPEs, as shown in Figure 8(B)’s CPE(., locked(tid, A)) and CPE(., locked(tid, B)) which verifies that Thread 1 still holds lock A and Thread 2 still holds lock B. Using this approach, if the entire sequence of CPEs are satisfied, then, at the very least, the root cause CPEs are known to be at least one part of the bug’s root cause, if not the entire root cause.

Notice that in Figure 8 the CpeSequence schedules p1 and p2 in either order, where p1 can happen first and then p2 or p2 and then p1. This is because the order in which the first set of locks are acquired are irrelevant and constraining such an event to one of the two orders will actually reduce the likelihood of reproducing the bug. Notice also that the second set of events for p3 and p4 also leave the order in which the CPEs are triggered open to further increase the likelihood of bug reproduction.

VIII. Usage Experience

In this section we discuss our experience using CP to reproduce known bugs with known trigger points in our test suite and RADBench [9]. We also discuss our unique opportunity to use CP to reproduce and fix, known but open (i.e., not fixed) bugs with unknown trigger points in TBoost.STM, an open-source software transactional memory library [6], [7]. Our experience using CP to find the trigger points of these open bugs and then fix them is telling of CP and CPE’s real-world practicality and usefulness. We list several of these bugs in Table I.

A. CP Test Suite

To test and verify the usefulness of CP, we created a test suite of 13 hand-crafted concurrency bugs, five bugs from RADBench’s real applications, and three bugs from the TBoost.STM library. The hand-crafted bugs consist of two order violations, three atomicity violations, four data races, and four deadlocks. Due to limited space, we only list five of these bugs in Table I: CP-4b, CP-4c, CP-4d, CP-4e, and CP-4f.

Once we had proven the basic concept of CP using our hand-crafted bugs, we applied it to RADBench, a concurrency bug benchmark suite that uses real-world applications [9]. We successfully reproduced each bug in RADBench that we explored using CP. A brief description of these bugs, labeled SpiderMonkey-1, SpiderMonkey-2, NSPR-1, NSPR-2, and NSPR-3, are shown in Table I.

B. Bug Reproducibility

CP reproduces each bug in our test suite and RADBench with increased probability than without CP. Table I shows that CP can reproduce these bugs with an improved probability ranging from $2\times$ to $10,000\times$ when compared to the likelihood of the same bug occurring without CP. For example, SpiderMonkey-2 is a garbage collection assertion failure that occurs with a frequency of 1 out of every 5,000 executions when using a non-CP bug reproduction technique. With CP, we were able to increase the frequency of bug reproduction to 1 out of every 15 executions, which is more than a two order of magnitude improvement.

C. Bug Isolation

CP not only quickly reproduces bugs with known reproduction steps, it also helps pinpoint concurrency bugs where the steps for reproduction are not known. For example, the developers of the TBoost.STM library knew there was a concurrency bug somewhere in their library, but, because they could not reliably reproduce it, it remained unfixed for over 18 months. In this section, using only CP, we describe how we were able to isolate
<table>
<thead>
<tr>
<th>Bug Name</th>
<th>Description</th>
<th>Prob. w/o CP</th>
<th>Prob. w CP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP-4b</td>
<td>Atomicity violation, divide by zero</td>
<td>4.4444%</td>
<td>100.0000%</td>
</tr>
<tr>
<td>CP-4c</td>
<td>Order violation, invalid iterator</td>
<td>10.0000%</td>
<td>100.0000%</td>
</tr>
<tr>
<td>CP-4d</td>
<td>Atomicity violation, inconsistent data</td>
<td>22.2222%</td>
<td>100.0000%</td>
</tr>
<tr>
<td>CP-4e</td>
<td>Order violation, STL crash</td>
<td>10.0000%</td>
<td>100.0000%</td>
</tr>
<tr>
<td>CP-4f</td>
<td>Deadlock, dynamic locking</td>
<td>3.3333%</td>
<td>100.0000%</td>
</tr>
<tr>
<td>NSPR-1</td>
<td>Assertion failure</td>
<td>20.0000%</td>
<td>100.0000%</td>
</tr>
<tr>
<td>NSPR-2</td>
<td>Hang</td>
<td>33.3333%</td>
<td>100.0000%</td>
</tr>
<tr>
<td>NSPR-3</td>
<td>Hang</td>
<td>5.0000%</td>
<td>100.0000%</td>
</tr>
<tr>
<td>SpiderMonkey-1</td>
<td>Segfault</td>
<td>50.0000%</td>
<td>100.0000%</td>
</tr>
<tr>
<td>SpiderMonkey-2</td>
<td>Assertion failure</td>
<td>0.0200%</td>
<td>6.6667%</td>
</tr>
<tr>
<td>TBoost.STM-1</td>
<td>Order violation, crash</td>
<td>1.0000%</td>
<td>100.0000%</td>
</tr>
<tr>
<td>TBoost.STM-2</td>
<td>Live lock, stall</td>
<td>3.3333%</td>
<td>92.5000%</td>
</tr>
<tr>
<td>TBoost.STM-3</td>
<td>Data race, inconsistent data</td>
<td>0.0010%</td>
<td>100.0000%</td>
</tr>
</tbody>
</table>

TABLE I. A SNAPSHOT OF SOME BUGS REPRODUCED BY CPE.

Fig. 9. Three TBoost.STM Concurrency Violations Reproduced and Fixed with CP.

this bug (TBoost.STM-1), which is one of the three complex concurrency violations in TBoost.STM that we successfully reproduced and fixed as shown in Figure 9. When we first started working with the TBoost.STM library, these bugs were open; specifically, they were known bugs that, because their reproduction steps were unknown, had not been fixed. This allowed us the unique opportunity to apply CP as would be in a real-world fashion. By using the recipes described in Section VII, we were able to identify each bug’s root cause and provide a fix for each. The developers of TBoost.STM have since reviewed our patches and applied them to TBoost.STM’s latest source code.

TBoost.STM-1 is an order violation leading to a program crash. It requires a schedule and state of three transactions each of which concurrently execute across three threads as shown in Figure 9(a). When we began investigating TBoost.STM-1, all that was known was that under rare and unknown circumstances TBoost.STM would crash. Once we were able to reproduce the crash, the crash point told us where to put the effect CP. In this case, we found that the bug effect was an illegal memory location read access to a contention manager (CM) [8], [20] shared memory transaction container, AbortTxes. We placed an effect CP just before this read access and tried to find the root cause.

We speculated that there was a write operation performed to AbortTxes causing the crash, so we added a root cause CP to all write locations to it. We also created a new shared memory list, ToAbort, that kept track of each transaction added to AbortTxes and did a sanity check to each transaction to ensure it was legal memory. This helped us verify that AbortTxes was not corrupted by an external memory overwrite operation and that at the point in which a transaction was added to AbortTxes that the transaction did in fact exist. Once we verified that both behaviors were correct, the only possible root cause seemed to be that the transaction in question was deleted at some point between its insertion to AbortTxes and its access causing the crash. This meant that AbortTxes was not being properly maintained. After searching the code, we found a single case where a transaction was inserted into AbortTxes but, because of a denied CM request, was never removed. We concluded that if such a transaction was then destroyed, the program would crash if it was later accessed in AbortTxes.

To verify this was the root cause of the bug, we added a root cause CP after this behavior. In the root cause CP, we set ToAbort = AbortTxes and updated our effect CP’s predicate to trigger if AbortTxes contained an element from ToAbort. These root cause and effect CPs triggered the CP system before each crash bug. However, we noticed that the CP system would also trigger when the bug did not occur.
This indicated that there were cases when the transaction was not destroyed. After additional investigation, we found that if a transaction committed after the root cause CP was triggered, the bug would not occur because AbortTxes would be cleared. We then changed the root cause CP to a cp_serial and placed another cp_serial before commit operations, ensuring no transaction could commit. Once these new CPs were in place, we triggered the bug 100% reliability. The fix to this bug was a one line change to clear AbortTxes when the CM denied permission for a committing transaction to abort other in-flight transactions.

IX. Conclusion

In this paper, we presented concurrent predicates (CPs) and concurrent predicate expressions (CPEs), synchronization primitives that facilitate the reproduction of concurrency violations by capturing the program state and necessary schedule to reproduce a specific bug. We discussed how CP and CPE manage interference from other threads and how CP provides an important guarantee, called self stability, that ensures the conditions required for bugs are not perturbed for a specific temporal bound. We also demonstrated the trade-offs between CP’s flexibility and CPE’s ease of use.

Next, we presented a general sketch of how to use CP and CPE and three programming recipes to reproduce data races, atomicity violations, and deadlocks. We discussed our experience with CP and CPE applying it to our test suite of 13 bugs and, using five bug from RADBench, we were able to produce new schedules to reproduce known bugs and reproduce certain bugs with over 100× greater frequency than the previously best known bug reproduction techniques. Finally, we showed how we used our general sketch and recipes to reproduce and eventually fix three highly complex bugs from TBoost.STM that, prior to our patches, were unresolved.

References


