USENIX HotPar 2012, Berkeley, CA.

Concurrent Predicates: Finding and Fixing the Root Cause of Concurrency Violations

Justin E. Gottschlich   Gilles A. Pokam   Cristiano L. Pereira

Intel Corporation

Abstract

To reduce the complexity of debugging multithreaded programs, researchers have developed compile- and run-time techniques that automatically detect concurrency bugs. These techniques can identify a wide range of shared memory errors, but are sometimes impractical because they produce many false positives making it difficult to triage and reproduce specific bugs. To address these concerns, we introduce a control structure, called concurrent predicate (CP), which allows programmers to single out a specific bug by specifying the conditions that must be satisfied for the bug to be triggered. Using bugs from a test suite of 23 programs, applications from RADBench, and TBoost.STM, we show how CP is used to diagnose and reproduce such bugs that could not otherwise be reproduced using similar techniques.

1. Introduction

To reduce the complexity of debugging multithreaded programs, researchers have developed innovative ways to automatically detect concurrency violations [3, 13, 16, 20]. Research in this area generally focuses on systematic model checking to exhaustively test all possible thread interleavings [1, 15, 25] or random testing to overcome impracticality issues caused by state-space explosion [4, 22]. Although many bugs may be found by these automated systems, it can be challenging for a programmer to reproduce a specific bug he or she is interested in using such techniques because of false positives or the emergence of non-critical bugs. Yet, reliable bug reproduction is usually the first step to fixing software defects. Unfortunately, without record and replay systems [14, 18, 19], reproducing a specific bug can only be achieved when the root cause is known; that is, when the *conditions* required to expose the bug are satisfied.

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 [21]. Any number of parallel assertions can be placed in a program, enabling a programmer to track multiple parallel assertion violations within a single execution. 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 root cause information of concurrency bugs.

Park and Sen resolve this issue with their novel *concurrent breakpoint* system which captures both the cause and the effect of a given concurrency violation [17]. Once a set of concurrent breakpoints is found to reproduce a bug, the programmer can attempt to fix the bug because he or she knows its root cause. After a bug fix is added to the code, the programmer can then build confidence that the fix is correct by re-executing the program and ensuring the previously inserted concurrent breakpoints no longer trigger.

In this paper, we present concurrent predicate (CP) a programming control structure inspired by parallel assertions and concurrent breakpoints that extends both ideas to provide a more complete and generalized solution to reproduce concurrency violations. Like parallel assertions, any number of CPs can be active within a program at a time, enabling programmers to reproduce multiple bugs within the same execution or to reproduce the same bug from multiple vantage points. Like concurrent breakpoints, CP captures both the effect and root cause of multithreaded bugs and increases their likelihood of reproduction by using programmer-supplied delays. Yet, unlike either of them, CP provides specific a timing guarantee in which the predicates of the program will remain satisfied, enabling deterministic bug reproduction within certain constraints. CP also manages *non-essential thread interference*, those threads that do not contribute to the reproduction of a bug but can obfuscate it, which is critical to reproducing complex, real-world multithreaded bugs where interfering threads often reduce bug reproducibility.

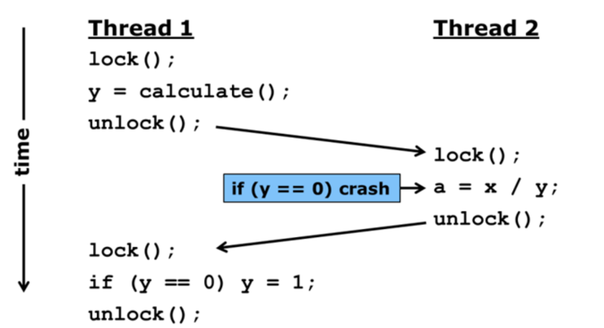


Figure 1. An Atomicity Violation where, if calculate() returns 0, the program can exhibit a divide by zero exception.

To demonstrate the usefulness of this approach, consider the code shown in Figure 1 which assumes `y` is a shared variable between Threads 1 and 2 and the program has the invariant of `y != 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.

Parallel assertions will reproduce this bug, but it will not capture the root cause. Concurrent breakpoints, on the other hand, will capture both the effect and the root cause. Yet, because concurrent breakpoints does not manage thread interference, if multiple threads were to simultaneously execute Thread 1's code, `y` could be set to `1` immediately after it was set to `0` by another thread, effectively eliminating any chance to reproduce the bug. Secondly, because concurrent breakpoints does not provide a timing guarantee, Thread 1's breakpoint might timeout after it has been concurrently satisfied with Thread 2. Thread 1 might then subsequently execute its second critical section performing `y = 1` before Thread 2 reports the bug, resulting in a bug report that includes unsatisfied conditions. As we describe in Section 3, the CP control structure handles both of these concerns.

1.1 Non-Essential Thread Interference and Timing Guarantees

An observation we made when designing CP is that while only a few threads may contribute to a bug, often times many threads contribute to the obfuscation of such bugs, by perturbing the program state once a bug's conditions have been satisfied. As a result, careful attention must be paid to ensure these non-essential, interfering threads do not further obscure already obscure bugs. Although our experiments demonstrate that most concurrency violations can be captured with only two threads (some exceptions exist), they also show that a system that intends to reproduce concurrency violations must specifically have measures in place to manage, that is, *eliminate*, interference from threads that do not contribute to the bug. In Section 3, we discuss the three variants of the CP control structure, each of which manages varying degrees of thread interference, and show how CP eliminates thread interference for a modified divide by zero bug as is summarized in Figure 1.

Additionally, systems like CP must provide a timing guarantee regarding the duration of time in which the conditions required to reproduce a bug will remain satisfied. Without such a guarantee, programmers will not fully understand how such systems behave or how to effectively utilize them. We call such a timing guarantee *self stability*, a notion lifted from Dinsdale-Young et al., and describe it in more detail in Section 3 [2]. Figure 2 illustrates this guarantee, at a high-level. Unlike the works of parallel assertions and concurrent breakpoints, the CP system guarantees that the divide by zero defect is *deterministically* reproduced if *(i)* the program state that is necessary to reproduce the bug is reached in Thread 1 (i.e., `y == 0`) and *(ii)* the CP control structure in Thread 2 is waiting for notification of this program state from Thread 1. [1] Under such conditions, CP will always reproduce the bug.

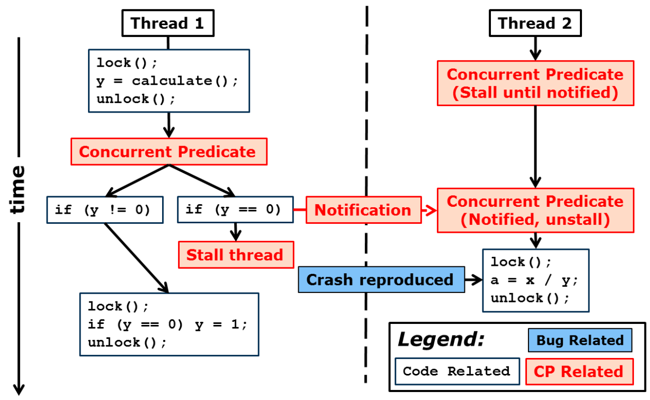


Figure 2. An Overview of How CP Controls a Program's Execution to Reproduce a Divide By Zero Exception.

2. The CP Control Structure

In this section we illustrate how CP is programmed to reproduce the divide by zero bug shown in Figures 1 and 2. We first present the CP control structure, including a brief discussion of its compound statements and parameters, and then show the code that is added to the original program. In order to limit the initial example's complexity, our first CP solution does not manage thread interference. In Section 3, we present a second CP solution for a modified divide by zero bug that manages an unlimited amount of thread interference.

```
cp(state, priority, control, retryTime,
   retryIfFalse, predicate)
{
   pre { /* serialized multi-ops */ }

   if_satisfied { /* serialized multi-ops */ }
   else { /* serialized multi-ops */ }

   post { /* unserialized multi-ops */ }
}
```

Figure 3. The Concurrent Predicate Control Structure.

The CP control structure syntax is shown in Figure 3. 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 their CP until T_1 completes its execution of `pre`. This behavior enables self stability, as described in Section 3.2. The compound statements that are part of CP's control structure are as follows.

[1] This assumes no outside interference, as described in Section 3.2.

- `pre`: a compound statement that is executed when a CP's control structure is entered. `pre` is executed before any of CP's control structure parameters are evaluated, and before any of the other compound statements are executed.
- `if_satisfied`: a compound statement that is executed once it is approved by the CP system.
- `else`: a compound statement that is executed only if the CP does not execute its `if_satisfied`.
- `post`: a compound statement that is executed before the CP control structure is exited. This compound statement executes immediately after `if_satisfied` or `else`.

For completeness, the following list includes the definitions of each of CP's parameters. However, the only parameters that are essential to understand the examples presented in this paper are: `state`, `control`, and `predicate`, the first, third, and final parameter of CP.

- `state`: an instance of `ConcurrentPredicateState`, highlighted as CP *state* in Figure 4, that must be created for each multithreaded bug and shared across the CPs that are necessary to reproduce the bug. `state` has the following fields:
  - `N`: the programmer supplied number of CPs that, along with the `to_satisfy` field, is used to determine if `verify()`, shown in Algorithm 1, returns `true`.
  - `to_satisfy`: the programmer supplied conditional operator (e.g., ==, <, >, !=) that is applied to `N` and used in `verify()`, shown 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`, once `predicate` and `verify()` are also found to be `true`, `if_satisfied` will 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`. CPs are guaranteed to wait at least as long as `retryTime` if `predicate` and `verify()` have not yet returned `true`, but they 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`.
- `predicate`: user-supplied condition that must return `true` for the CP's `if_satisfied` to be executed.

When a CP's `verify()`, shown in Algorithm 1, and its `predicate` and `control` are `true`, along with it having the highest `priority` amongst active CPs, it will be allowed to execute its `if_satisfied` operations. The CPs that are sufficient to reproduce the divide by zero bug are shown in Figure 4. The programmer first creates a shared instance of `ConcurrentPredicateState` and then adds CP control structures (highlighted in Figure 4) to control the forward progress of the program based on its current state (i.e., y == 0).

Algorithm 1 Verify

procedure VERIFY(st)
    $S \leftarrow st.satisfied.size$
    $N \leftarrow st.N$
    if $st.to\_satisfy \equiv no\_predicates \wedge S \equiv 0$ **then return** $true$
    else if $st.to\_satisfy \equiv less\_than\_N \wedge S < N$ **then return** $true$
    else if $st.to\_satisfy \equiv greater\_than\_N \wedge S > N$ **then return** $true$
    else if $st.to\_satisfy \equiv equal\_to\_N \wedge S \equiv N$ **then return** $true$
    else if $st.to\_satisfy \equiv active\_predicates \wedge S \equiv st.in\_predicates$ **then return** $true$
    end if
    return $false$
end procedure

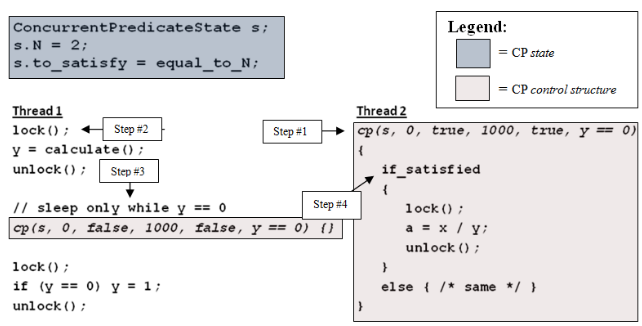


Figure 4. An Overview of the CP Control Structures Used to Reproduce the Divide by Zero Exception.

3. Design and Algorithm

CP has three variants: general (`cp()`), serial (`cp_serial()`), and serial(id) (`cp_serial(id)`). The three CP variants are meant to be used together to reproduce complex heisenbugs that cannot (easily) be reproduced by using only one.

For our experiments, the most commonly used type, referenced in Figure 4, 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. In particular, 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

Algorithm 2 The CP Run-Time Algorithm

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

```
procedure CP(state, priority, control, retryTime, retryIfFalse, predicate)
    Lock state.mutex
    executeIfSatisfied ← false
    Execute pre-execution operations
    Insert threadId into state.in_predicate
    if control then
        Insert (threadId, priority) into state.priorities
    end if
    Unlock state.mutex
    while retryTime > 0 do
        beginTime ← Clock()
        Lock state.mutex
        if predicate then
            Insert theadId into state.satisfied
        else
            Remove theadId from state.satisfied
        end if
        if verify(state) ∧ predicate then
            executeIfSatisfied ← true
            if ¬control then
                No-Op
            else if control ∧ priority ≡ state.top_priority then
                Remove theadId from state.satisfied
                Remove theadId from state.in_predicates
                Remove (theadId, priority) from state.priorities
                Execute if_satisfied operations
                Unlock state.mutex
                Break
            end if
        end if
        Unlock state.mutex
        if ¬retryIfFalse then
            Break
        end if
        SLEEP(1)
        endTime ← Clock()
        retryTime ← retryTime − (endTime − beginTime)
    end while
    Lock state.mutex
    if ¬executeIfSatisfied then
        Execute else operations
    end if
    Remove theadId from state.satisfied
    Remove theadId from state.in_predicates
    Remove (theadId, priority) from state.priorities
    Unlock state.mutex
    Execute post-execution operations
end procedure
```

basic components and eliminate additional and non-essential thread contention. Due to space limitations, we only include the algorithmic details for the general CP.

3.1 Managing Non-Essential Thread Interference

As discussed in Section 1, a feature that sets our CP design apart from prior works is that it can manage non-essential thread interference by limiting concurrency for certain regions of code that would otherwise interfere with the system's ability to reproduce a bug. This is illustrated in Figure 5, which revisits the divide by zero bug presented in Figure 1 of Section 1, where Thread 1 is replaced by Threads 1 ... N-1. Without some mechanism to prevent non-essential thread interference, this minor modification to the problem results in a decrease of the probability of reproducing the bug. In general, the greater N, the less likely the bug will occur due to thread interference.

CP handles this interference by restricting each of the 1 ... N-1 threads to serial execution by using the `cp_serial` control structure. The operations that might interfere amongst the threads are placed within the `pre` and `post` sections of the `cp_serial` control structure, thereby eliminating their potential for 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` can execute concurrently, the bug is still reproducible once the necessary predicates are satisfied.

```
ConcurrentPredicateState s;
s.N = 2;
s.to_satisfy = equal_to_N;
```

Threads 1...N-1

```
cp_serial(s, 0, false, 1000, false,
          y == 0)
{
  pre { lock(); y=calc(); unlock(); }
  post
  {
    lock();
    if (y == 0) y = 1;
    unlock();
  }
}
```

Thread N

```
cp(s, 0, true, 1000,
   true, y == 0)
{
   if_satisfied
   {
      lock();
      a = x / y;
      unlock();
   }
   else { /* same */ }
}
```

Figure 5. Using CP to Reproduce the Divide by Zero Exception While Eliminating Thread Interference.

3.2 Self Stability

A key characteristic of our CP design is in the self stability it guarantees for the `if_satisfied` or `else` sequence of operations that execute after its `predicate` and `verify()` conditional checks have returned true or false. The lifted notion of self stability that we use for CP originates from Dinsdale-Young et al. [2]. Informally, Dinsdale-Young et al. define self stability as a property of an execution 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. In essence, the predicate state and their associated post-operations are free from outside interference until the post-operations 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 post-operations (i.e, 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 limitations, once their associated predicates have been satisfied. Without self stability, approaches like CP can still reproduce concurrency bugs that are largely state-dependent, but cases will arise when the state that is required to reproduce a bug is captured and lost again before the operations that reveal the effect of the bug have been executed.

CP's self stability is achieved 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, 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 are based.

This guarantee, however, does not safeguard a CP's execution from threads whose executions are outside the lexical scope of a CP control structure. The programmer can prevent these threads from interfering by adding CPs to all program locations that might mutate the shared data accessed within a given predicate. When following this method, we have not encountered any self stability issues that prevent a concurrency bug from being reproduced once its predicates have been satisfied.

4. Experience with CP

Due to space limitations, we only provide a brief synopsis of the bugs we reproduced and fixed with CP.

4.1 CP Test Suite

The CP test suite currently consists of 23 concurrency bugs that range from violations as simple as accessing unprotected shared variables in two or more threads to complex tests such as multiple threads dynamically acquiring a range of mutexes in a random order that have a low probability of causing a deadlock. CP is capable of reproducing all concurrency violations in our test suite, resulting in a $5\times$ to over a $1000\times$ improvement in the likelihood of reproducing the targeted bug when compared to the original program.

4.2 RADBench

We have successfully applied CP to three of the ten bugs listed in the RADBench concurrency violation test suite: SpiderMonkey-1, NSPR-2, and NSPR-3 [12]. [2] In addition to reproducing these concurrency violations, we have also identified new ways to reproduce NSPR-3, which were not included in the original description of the bug. We believe this demonstrates that CP reduces the complexity of bug manifestation such that, once the CPs that are sufficient to reproduce a bug are found, a programmer can more easily understand the root cause of a bug. This enables him or her to reason about other root causes that may have been overlooked and, perhaps, not covered with a particular bug fix.

[2] Thus far, we have been 100% successful applying CP to RADBench. We plan to apply CP to the remaining 7 bugs, soon.

4.3 TBoost.STM

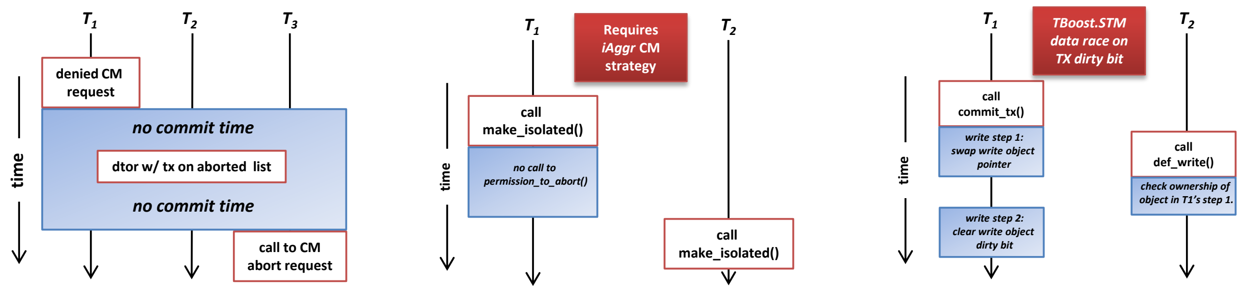
TBoost.STM is a C++ software transactional memory (STM) [10, 23] library that provides a simple C++ programming interface for transactional memory [6, 7, 11]. TBoost.STM is open source and freely available on the web. We have used CP to reproduce and fix three complex concurrency violations in TBoost.STM as shown in Figure 6. At the time we began using CP with TBoost.STM, the three bugs we describe in Section 4.3.2 were open and their root cause was unknown. By utilizing the approach described in Section 4.3.1, we were able to identify each bug's root cause and then provide a fix for each of them. TBoost.STM's source code has been updated to include all of our fixes.

4.3.1 Using CP

Parallel programs, like any other class of programs, have defects that are generally found by observing unwanted effects. Therefore, when we first began investigating a concurrency violation, we generally placed a CP at the location of the bug's effect, what we refer to as the *effect CP*, as is done in Thread 2 of the divide by zero example of Figure 4 because these locations are generally known when the bug is first observed. To identify the root cause of a concurrency violation, which is generally not known when the bug is observed, we used a *divide-and-conquer-like* approach. Identifying the root cause of a bug can be challenging, however, we found that the following techniques generally reduced such difficulty.

First, root causes of concurrency violations are always writes to shared memory. Therefore, read operations should not be considered as root cause candidates. Next, because we generally made no attempt to find the exact location of the root cause on our initial CP placement, we attempted to find the *root cause CP*, the CP placed after a thread performs the root cause behavior (Thread 1 in Figure 4), by placing CPs at the locations that seemed the most unlikely to simultaneously trigger with the effect CP. This approach generally reinforced our understanding of the program and helped us quickly eliminate cases that seemed obviously correct. In the case of TBoost.STM-2, this approach immediately led us to the root cause, because our assumptions were incorrect.

Last, and perhaps most importantly, our experience with CP demonstrated that no matter how complex the bug, when the CPs for a given bug were placed at the correct locations,



(a) TBoost.STM-1: Order Violation. (b) TBoost.STM-2: Livelock. (c) TBoost.STM-3: Data Race.

Figure 6. Three TBoost.STM Concurrency Violations Reproduced and Fixed with CP.

they triggered almost immediately. That meant that if the CPs were *not* placed at the correct locations, only a few executions were generally needed to verify this. Trusting the CP system when CPs did not trigger, indicating our guesses were incorrect, was perhaps the most challenging part of our process. This is because our programmer's intuition did not want us to discard some of our root cause guesses, as we strongly believed we had found the root cause of the bug.

4.3.2 TBoost.STM Bugs

TBoost.STM-1 is an order violation that leads to a program crash and requires both a specific schedule and state of three transactions, T_1, T_2, and T_3, each of which concurrently execute across three threads. T_1 requests to abort conflicting in-flight transactions and is denied permission, leaving a shared container populated with the unaborted transactions, the critical program state information that causes the crash. T_2, a transaction that must be referenced on the shared container T_1 accessed, is then aborted resulting in a dangling pointer in the shared container. T_3 then requests to abort its conflicting transactions, which contains a deallocated reference to T_2, thereby resulting in a program crash. Concurrency violation tools that only perturb schedules are unlikely to reproduce TBoost.STM-1 because the specific order of the events that lead to the bug occur with high frequency. It is only when these events are coupled with the precise state, as described above, that the program crash occurs.

TBoost.STM-2 is a livelock that is caused when two transactions simultaneously request permission to become *irrevocable*, that is, not abortable [5, 26]. Before a transaction can be made irrevocable, it must abort all active transactions. This bug was caused by an inverted conditional check inside the TBoost.STM's contention manager (CM) [9, 24], which grants or denies a transaction permission to abort active transactions. The bug would only occur if the iAggr CM was used [8], because iAggr always grants a transaction permission to abort revocable transactions. By using the iAggr CM strategy and having the inverted conditional check, both transactions are continually denied permission to abort each other and end up spinning in a while loop requesting permission indefinitely. As before, only capturing the specific thread orderings would not cause TBoost.STM-2's liveness condition because the iAggr CM must be active. This is handled with CP by including a check for the CM within the predicate for one of the two necessary CPs.

TBoost.STM-3 is a value-based data race that results in an inconsistent view of memory and occurs with exceptionally low frequency ($\approx 1/10,000,000$ transactions). The bug occurs when one transaction, T_1, is in the process of updating its written data to global memory, while another transaction, T_2, concurrently checks if the same shared memory location is within its write set. The bug rarely occurs because T_2 must access the same memory location that is being updated by T_1 precisely between T_1's pointer `std::swap()` and its subsequent one byte assignment and T_2 must not have already written to the location, so it will not be within its write set, thereby returning an incorrect result on its check for ownership. As with the other two TBoost.STM bugs, TBoost.STM-3 cannot be reproduced by simply perturbing the threads' schedules. Instead, it requires that T_1 and T_2 access the same shared memory location and that T_2's write set does not contain such a memory element. Once the CPs were in place to reproduce this bug, TBoost.STM-3 occurred with a frequency of $\approx 1/100$ transactions, a five order of magnitude improvement (i.e., a $10,000\times$ increase in likelihood), over the original execution.

5. Conclusion

In this paper, we presented concurrent predicate (CP), a programming control structure that facilitates the reproduction of concurrency violations by capturing the program state and producing the specific schedule required to reproduce bugs. We discussed how CP manages interference from other threads and provides an important guarantee, called self stability, that ensures the conditions required for bugs are not perturbed for a specific temporal bound. We used bugs from a test suite of 23 programs, applications from RADBench, and TBoost.STM, to show how CP diagnosed and reproduced bugs that could not otherwise be reproduced using similar techniques.

References

[1] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2:215–222, May 1976.

[2] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 504–528, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[4] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[5] J. E. Gottschlich and J. Chung. Optimizing the concurrent execution of locks and transactions. In *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, September 2011.

[6] J. E. Gottschlich, J. G. Siek, P. J. Rogers, and M. Vachharajani. Toward simplified parallel support in C++. In *Proceedings of the Fourth International Conference on Boost Libraries (BoostCon)*. May 2009.

[7] J. E. Gottschlich, J. G. Siek, M. Vachharajani, D. Y. Winkler, and D. A. Connors. An efficient lock-aware transactional memory implementation. In *Proceedings of the Fourth International ACM Workshop on ICOOOLPS. In conjunction with ECOOP.* July 2009.

[8] J. E. Gottschlich, M. Vachharajani, and J. G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, April 2010.

[9] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In M. K. Aguilera and J. Aspnes, editors, *PODC*, pages 258–264. ACM, 2005.

[10] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the symposium on principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*. May 1993.

[12] N. Jalbert, C. Pereira, G. Pokam, and K. Sen. Radbench: a concurrency bug benchmark suite. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, HotPar'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[13] P. Joshi and K. Sen. Predictive typestate checking of multithreaded java programs. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 288–296, Washington, DC, USA, 2008. IEEE Computer Society.

[14] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 73–84, New York, NY, USA, 2009. ACM.

[15] M. Musuvathi, S. Qadeer, and T. Ball. Chess: A systematic testing tool for concurrent software, 2007.

[16] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

[17] C. S. Park and K. Sen. Concurrent breakpoints. PPoPP '12, New York, NY, USA, 2012.

[18] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 2–11, New York, NY, USA, 2010. ACM.

[19] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, J. Ha, and Y. Wu. Coreracer: A practical memory race recorder for multcore x86 processors. In *Proceedings of the 44th International Symposium on Microarchitecture (MICRO)*, December 2011.

[20] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997.

[21] D. Schwartz-Narbonne, F. Liu, T. Pondicherry, D. August, and S. Malik. Parallel assertions for debugging parallel programs. In *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pages 181 –190, july 2011.

[22] K. Sen. Effective random testing of concurrent programs. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 323–332, New York, NY, USA, 2007. ACM.

[23] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Principles of Distributed Computing*. Aug 1995.

[24] M. F. Spear, L. Dalessandro, V. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP*, Feb. 2009.

[25] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, Washington, DC, USA, 2000. IEEE Computer Society.

[26] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA*, 2008.