Applying Hardware Transactional Memory for Concurrency-Bug Failure Recovery in Production Runs

Yuxi Chen, Shu Wang, and Shan Lu, *University of Chicago;*
Karthikeyan Sankaralingam, *University of Wisconsin – Madison*

https://www.usenix.org/conference/atc18/presentation/chen-yuxi

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Applying Hardware Transactional Memory for Concurrency-Bug Failure Recovery in Production Runs

Yuxi Chen   Shu Wang   Shan Lu
University of Chicago

Karthikeyan Sankaralingam
University of Wisconsin – Madison

Abstract

Concurrency bugs widely exist and severely threaten system availability. Techniques that help recover from concurrency-bug failures during production runs are highly desired. This paper proposes BugTM, an approach that leverages Hardware Transactional Memory (HTM) on commodity machines for production-run concurrency-bug recovery. Requiring **no** knowledge about where are concurrency bugs, BugTM uses static analysis and code transformation to insert HTM instructions into multi-threaded programs. These BugTM-transformed programs will then be able to recover from a concurrency-bug failure by rolling back and re-executing the recent history of a failure thread. BugTM greatly improves the recovery capability of state-of-the-art techniques with low run-time overhead and no changes to OS or hardware, while guarantees not to introduce new bugs.

1 Introduction

1.1 Motivation

Concurrency bugs are caused by untimely accesses to shared variables. They are difficult to expose during in-house testing. They widely exist in production-run software [26] and have caused disastrous failures [23, 32, 40]. Production run failures severely hurt system availability: the restart after a failure could take long time and even lead to new problems if the failure leaves inconsistent system states. Furthermore, comparing with many other types of bugs, failures caused by concurrency bugs are particularly difficult to diagnose and fix correctly [50]. Techniques that handle production-run failures caused by concurrency bugs are highly desired.

Rollback-and-reexecution is a promising approach to recover failures caused by concurrency bugs. When a failure happens during a production run, the program rolls back and re-executes from an earlier checkpoint. Due to the unique non-determinism nature of concurrency bugs, the re-execution could get around the failure.

This approach is appealing for several reasons. It is generic, requiring no prior knowledge about bugs; it improves availability, masking the manifestation of concurrency bugs from end users; it avoids causing system inconsistency or wasting computation resources, which often come together with naive failure restarts; even if not successful, the recovery attempts only delays the failure by a negligible amount of time.

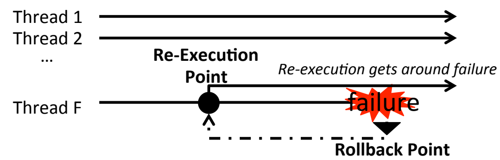


Figure 1: Single-threaded recovery for concurrency bugs

This approach also faces challenges in performance, recovery capability, and correctness (i.e., not introducing new bugs), as we elaborate below.

Traditional rollback recovery conducts full-blown multi-threaded re-execution and whole-memory checkpointing. It can help recover almost all concurrency-bug failures, but incurs too large overhead to be deployed in production runs [35, 39]. Even with support from operating systems changes, periodic full-blown checkpointing still often incurs more than 10% overhead [35].

A recently proposed recovery technique, ConAir, conducts single-threaded re-execution and register-only checkpointing [55]. As shown in Figure 1, when a failure happens at a thread, ConAir rolls back the register content of this thread through an automatically inserted `longjmp` and re-executes from the return of an automatically inserted `setjmp`, which took register checkpoints. This design offers great performance (<1% overhead), but also imposes severe limitations to failure-recovery capability. Particularly, with no memory checkpoints and re-executing only one thread, ConAir does not allow its re-execution regions to contain writes to shared variables (referred to as W_s) for correctness concerns, severely hurting its chance to recover many failures.

This limitation can be demonstrated by the real-world example in Figure 2. In this example, the `NULL` assignment from Thread-2 could execute between the write (A_1) and the read (A_2) on `s→table` from Thread-1, and cause failures. At the first glance, the failure could be recovered if we could rollback Thread-1 and re-execute both A_1 and A_2. However, such rollback and re-execution cannot be allowed by ConAir, as correctness can no longer be guaranteed if a write to a shared variable is re-executed (W_s in Figure 2): another thread t could have read the old value of s→table, saved it to a local pointer, the re-execution then gave s→table a new value, causing inconsistency between t and Thread-1 and deviation from the original program semantics.

```
//Thread-1                                  //Thread-2
s->table = newTable(...); //A1, Ws
                                            s->table = NULL;
if(!s->table)            //A2
    //fatal-error message; software fails
```

Figure 2: A real-world concurrency bug from Mozilla

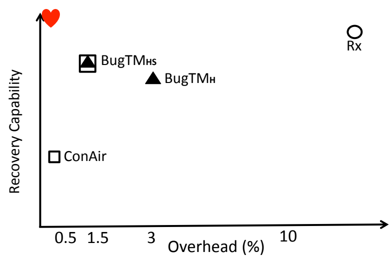


Figure 3: Design space of concurrency-bug failure recovery (Heart: non-existing optimal design; Rx [35] changes OS)

1.2 Contributions

Existing recovery techniques only touch two corners of the design space — good performance but limited recovery capability or good recovery capability but limited performance — as shown in Figure 3. It is desirable to have new recovery techniques that combine the performance and recovery capability strengths of the existing two corners of design, while maintaining correctness guarantees. BugTM provides such a new technique leveraging hardware transactional memory (HTM) support that already exists in commodity machines.

At the first glance, the opportunity seems obvious, as HTM provides a powerful mechanism for concurrency control and rollback-reexecution. Previous work [46] also showed that TM can be used to **manually** fix concurrency bugs **after** they are detected.

However, **automatically** inserting HTMs to help tackle **unknown** concurrency bugs during **production runs** faces many challenges not encountered by *manually* fixing *already detected* concurrency bugs *off-line*:

Performance challenges: High frequency of transaction uses would cause large overhead unacceptable for production runs. Unsuitable content of transactions, like trapping instructions[1], high levels of transaction nesting, and long loops, would also cause performance degradation due to repeated and unnecessary transaction aborts.

Correctness challenges: Unpaired transaction-start and transaction-commit could cause software to crash. Deterministic aborts, such as those caused by trapping instructions, could cause software to hang if not well handled. We need to guarantee these cases do not happen and ensure software semantics remains unmodified.

	ReExecution Point	RollBack Point	Checkpoint Memory ?	ReExecution contains W_s?
ConAir	setjmp	longjmp	✗	✗
BugTM$_H$	StartTx	AbortTx	✓	✓
BugTM$_{HS}$	setjmp or StartTx	longjmp or AbortTx	✓✗	✓

Table 1: Design comparisons (W_s: shared-variable writes)

Failure recovery challenges: In order for HTM to help recovery, we need to improve the chances that software executes in a transaction when a failure happens and we need to carefully design HTM-abort handlers to correctly process the corresponding transaction aborts.

BugTM addresses these challenges by its carefully designed and carefully inserted, based on static program analysis, HTM start, commit, and abort routines. Specifically, we have explored two BugTM designs: BugTM$_H$ and BugTM$_{HS}$, as highlighted in Table 1. They are both implemented as LLVM compiler passes that automatically instrument software in the following ways.

Hardware BugTM, short for BugTM$_H$, uses HTM techniques[2] *exclusively* to help failure recovery. When a failure is going to happen, a hardware transaction abort causes the failing thread to roll back. The re-execution naturally starts from the beginning of the enclosing transaction, carefully inserted by BugTM$_H$.

BugTM$_H$ provides better recovery capability than ConAir — benefiting from HTM, its re-execution region can contain shared variable writes. However, HTM costs more than `setjmp/longjmp`. Therefore, the performance of BugTM$_H$ is worse than ConAir, but much better than full-blown checkpointing, as shown in Figure 3.

Hybrid BugTM, short for BugTM$_{HS}$, uses HTM techniques and `setjmp/longjmp` *together* to help failure recovery. BugTM$_{HS}$ inserts both `setjmp/longjmp` and HTM APIs into software, with the latter inserted only when beneficial (i.e., when able to extend re-execution regions). When a failure is going to happen, the rollback is carried out through transaction abort if under an active transaction or `longjmp` otherwise.

BugTM$_{HS}$ provides performance almost as good as ConAir and recovery capability even better than BugTM$_H$ by carefully combining BugTM$_H$ and ConAir.

We thoroughly evaluated BugTM$_H$ and BugTM$_{HS}$ using 29 real-world concurrency bugs, including *all* the bugs used by a set of recent papers on concurrency bug detection and avoidance [17, 19, 41, 55, 56, 57]. Our evaluation shows that BugTM schemes can recover from

[1]Certain instructions such as system calls will deterministically cause HTM abort and are referred to as trapping instructions.

[2]This paper's implementation is based on Intel TSX. However, the principles apply to other vendors' HTM implementations.

many more concurrency-bug failures than state of the art, ConAir, while still provide good run-time performance — 3.08% and 1.39% overhead on average for BugTM$_H$ and BugTM$_{\mathrm{HS}}$, respectively.

Overall, BugTM offers an easily deployable technique that can effectively tackle concurrency bugs in production runs, and presents a novel way of using HTM. Instead of using transactions to replace existing locks, BugTM automatically inserts transactions to harden the most failure-vulnerable part of a multi-threaded program, which already contains largely correct lock-based synchronization, with small run-time overhead.

2 Background

2.1 Transactional Memory (TM)

TM is a widely studied parallel programming construct [13, 15]. Developers can wrap a code region in a transaction (Tx), and the underlying TM system guarantees its atomicity, consistency, and isolation. Hardware transactional memory (HTM) provides much better performance than its software counterpart (STM), and has been implemented in IBM [12], Sun [8], and Intel commercial processors [1].

In this paper, we focus on Intel Transactional Synchronization Extensions (TSX). TSX provides a set of new instructions: `XBEGIN`, `XEND`, `XABORT`, and `XTEST`. We will denote them as `StartTx`, `CommitTx`, `AbortTx`, and `TestTx`, respectively for generality. Here, `CommitTx` may succeed or fail with the latter causing Tx abort. `AbortTx` explicitly aborts the current Tx, which leads to Tx re-execution unless special fallback code is provided. `TestTx` checks whether the current execution is under an active Tx.

There are multiple causes for Tx aborts in TSX. *Unknown abort* is mainly caused by trapping instructions, like exceptions and interrupts (abort code `0x00`). *Data conflict abort* is caused by conflicting accesses from another thread that accesses (writes) the write (read) set of the current Tx (abort code `0x06`). *Capacity abort* is due to out of cache capacity (abort code `0x08`). *Nested transaction abort* happens when there are more than 7 levels Tx nesting (abort code `0x20`). *Manual abort* is caused by `AbortTx` operation, with programmers specifying abort code.

2.2 ConAir

ConAir is a static code transformation tool built upon LLVM compiler infrastructure [22]. It is a state-of-the-art concurrency bug failure recovery technique as discussed in Section 1. We describe some techniques and terminologies that will be used in later sections below.

Recovery capability limitations ConAir does not allow its re-execution regions to contain any writes to shared variables. Many of its re-execution points (i.e.,

```
1  //Thread-1                                  1 //Thread-2
2  if(thd->proc){    //A1                      2
3      *buf++ = ' '; //Ws                      3
4      strcat(buf,thd->proc);//A2              4 thd->proc = NULL;
5                          //failure site
6  }
```

Figure 4: A real-world concurrency bug from MySQL

`setjmps`) are put right after shared-variable writes, which prevent re-execution regions from growing longer and severely limit the recovery capability of ConAir.

ConAir fundamentally cannot recover **any** RAW[3] violations (e.g., the bug in Figure 2) and WAR violations, as Table 2 shows. The reason is that the (RA)W and W(AR) have to be re-executed for successful recoveries, but ConAir cannot re-execute shared-variable writes.

ConAir also cannot recover other types of concurrency bugs if a shared-variable write happens to exist between the failure location and the ideal re-execution point. For example, the RAR atomicity violation in Figure 4 cannot be recovered by ConAir due to the write to `*buf` on Line 3. If Line 3 did not exist, ConAir could have rolled back Thread-1 to re-execute Line 2 and gotten around the failure. With Line 3, ConAir can only repeatedly re-execute the `strcat` on Line 4, with no chance of recovery.

Failure instruction f ConAir automatically identifies where failures may happen so that rollback APIs can be inserted right there. This identification is based on previous observations that >90% of concurrency bugs lead to four types of failures [56]: assertion violations, segmentation faults, deadlocks, and wrong outputs. BugTM will reuse this technique to identify potential failure locations, denoted as *failure instructions* f in the remainder of the paper. Specifically, ConAir identifies the invocations of `__assert_fail` or other sanity-check macros as failure instructions for assertion failures. ConAir then automatically transforms software to turn segmentation faults and deadlocks into assertion failures: ConAir automatically inserts assertions to check whether a shared pointer variable v is `null` right before v's dereference and check whether a pointer parameter of a string-library function is `null` right before the library call; ConAir automatically turns lock functions into time-out lock functions, with a long timeout indicating a likely deadlock failure, and inserts assertions accordingly. ConAir can help recover from wrong output failures as long as developers provide output specifications using assertions.

3 BugTM$_H$

3.1 High-Level Design

We discuss our high-level idea about where to put Txs, and compare with some strawman ideas based on perfor-

[3] (R/W)A(R/W) is short for (Read/Write)-after-(Read/Write).

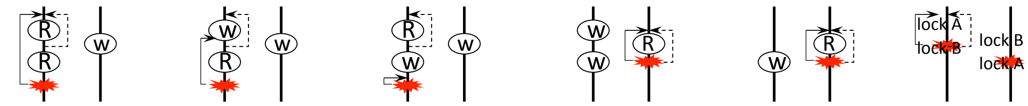
	Atomicity Violations				Order Violations	Deadlocks
	Read-after-Read (a) RAR	Read-after-Write (b) RAW	Write-after-Read (c) WAR	Write-after-Write (d) WAW	(e)	(f)
Types						
BugTM$_H$	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
ConAir	✓	–	–	✓	✓	✓

Table 2: Common types of concurrency bugs and how BugTM$_H$ and ConAir attempt to recover from them. (R/W: read/write to a shared variable; thick vertical line: the execution of one thread; dashed arrowed line: the re-execution region of BugTM$_H$; thin arrowed line: the re-execution region of ConAir; explosion symbol: a failure; -: cannot recover; ✓: sometimes can recover if the recovery does not require re-executing shared-variable writes; ✓✓: mostly can recover. The recovery procedure under BugTM$_{HS}$ is a mix of BugTM$_H$ and ConAir and hence is not shown in table.)

mance and failure-recovery capability.

Strawman approaches One approach is to chunk software to many segments and put every segment inside a hardware Tx [28]. This approach could avoid some atomicity violations, the most common type of concurrency bugs. However, it does not help recover from order violations, another major type of concurrency bugs. Furthermore, its excessive use of Txs will lead to unacceptable overhead for production-run deployment. Another approach is to replace all lock critical regions with Tx. However, this approach will not help eliminate many failures that are caused by missing lock.

Our approach In BugTM$_H$, we selectively put hardware Txs around places where failures may happen, like the invocation of an `__assert_fail`, the dereference of a shared pointer, etc. This design has the potential to achieve good performance because it inserts Txs only at selected locations. It also has the potential to achieve good recovery capability because in theory it can recover from all common types of concurrency bugs, as shown in Table 2 and explained below.

An atomicity violation (AV) happens when the atomicity of a code region $\mathbb{C}$ is unexpectedly violated, such as the bug shown in Figure 2. It contributes to more than 70% of non-deadlock concurrency bugs based on empirical studies [26], and can be further categorized into 4 sub-types depending on the nature of $\mathbb{C}$, as demonstrated in Table 2. Conflicting accesses would usually trigger a rollback recovery before the failure occurs, shown by the dashed arrow lines in Table 2(a)(b)(c), benefiting from the strong atomicity guarantee of Intel TSX — a Tx will abort even if the conflicting access comes from non-Tx code. For the bug shown in Figure 2 (an RAW atomicity violation), if we put the code region in Thread-1 inside a Tx, the interleaving NULL assignment from Thread-2 would trigger a data conflict abort in Thread-1 before the `if` statement has a chance to read the NULL. The re-execution of Thread-1 Tx will then re-assign the valid value to `s` $\rightarrow$ `table` for the `if` statement to read from, successfully avoiding the failure.

An order violation (OV) happens when an instruction *A* unexpectedly executes after, instead of before, instruction *B*, such as the bug in Figure 5. Different from AVs, conflicting memory accesses related to OVs may not all happen inside a small window. In fact, *A* may not have executed when a failure occurs in the thread of *B*. Consequently, the Tx abort probably will be triggered by a software failure, instead of a conflicting access, depicted by the dashed arrow in Table 2(e). Fortunately, the rollback reexecution will still give the software a chance to correct the unexpected ordering and recover from the failure. Take the bug shown in Figure 5 as an example. If we put a hardware Tx in Thread-1, when order violation leads to the assertion failure, the Tx will abort, rollback, and re-execute. Eventually, the pointer `ptr` will be initialized and the Tx will commit.

```
//Thread-1                       //Thread-2
                                 //ptr is NULL until
assert (ptr); //B                //initialized at A
//should execute after A         ptr = malloc (K); //A
```

Figure 5: A real-world OV bug (simplified from `Transmission`)

Deadlock bugs occur when different threads each holds resources and circularly waits for each other. As shown in Table 2(f), it can be recovered by Tx rollback and re-execution too, as long as deadlocks are detected.

Of course, BugTM$_H$ cannot recover from *all* failures, because some error-propagation chains cannot fit into a HTM Tx, which we will discuss more in Section 7.

Next, we will discuss in details how BugTM$_H$ surrounds failure sites with hardware Txs— how to automatically insert `StartTx`, `CommitTx`, `AbortTx`, and fallback-/retry code into software, while targeting three goals: (1) good recovery capability; (2) good run-time performance; (3) not changing original program semantics.

3.2 Design about `AbortTx`

BugTM$_H$ uses the same technique as ConAir to identify where failures would happen as discussed in Sec-

```
        if(_xtest()){
                //manually abort with abort code 0xFF
                _xabort(0xFF);
        }
```

Figure 6: BugTM$_H$ `AbortTx` wrapper function (`my_xabort`)

tion 2.2. BugTM$_H$ puts an `AbortTx` wrapper function `my_xabort` right before every failure instruction f, so that a Tx abort and re-execution is triggered right before a failure manifests. `my_xabort` uses a unique abort code `0xFF` for its `AbortTx` operation (as shown in Figure 6), so that BugTM$_H$ can differentiate different causes of Tx aborts and handle them differently.

3.3 Design about `StartTx` and `CommitTx`

Challenges We elaborate on two key challenges in placing `StartTx` and `CommitTx`, and explain why we can**not** simply insert well-structured atomic blocks (e.g., `__transaction_atomic` supported by GCC) into programs.

First, poor placements could cause frequent Tx aborts. Trapping instructions (e.g., system calls) and heavy TM nesting (>7 level) deterministically cause aborts, while long Txs abort more likely than short ones due to timer-interrupts and memory-footprint threshold. These aborts hurt not only performance, but also recovery — deterministic aborts of a Tx will eventually force us to execute the Tx region[4] in non-transaction mode, leaving no hope for failure recovery.

Second, poor placements could cause unpaired execution of `StartTx` and `CommitTx`, hurting both correctness and performance. When `CommitTx` executes without `StartTx`, the program will crash; when `StartTx` executes without a pairing `CommitTx`, its Tx will repeatedly abort.

Taking Figure 7 as an example, we want to put A_1 and A_2, both accessing global variable `G`, into a Tx together with `__assert_fail` on Line 6 for failure recovery. However, if we naively put `StartTx` on Line 2 and `CommitTx` on Line 12, forming a well structured atomic block, correct runs will incur repeated Tx aborts and huge slowdowns due to I/Os on Line 10. Simply moving `CommitTx` to right after Line 4 and keeping `StartTx` on Line 2 still will not work — when `else` is taken, the earlier `StartTx` has no pairing `CommitTx` and the Tx still aborts due to I/Os.

We address the first challenge by carefully placing `StartTx` and `CommitTx`. We address the second challenge mainly through our `StartTx`, `CommitTx` wrapper-functions.

Where to `StartTx` and `CommitTx` The design principle is to minimize the chance of aborts that are unrelated to concurrency bugs, tackling the first challenge above. BugTM$_H$ achieves this by making sure that its Txs do

[4]We will refer to the code region between our `my_xbegin` and `my_xend` as a Tx region, which may be executed in transactional mode.

```
1  void func(...){
2
3    G = g;  //A1
4    if(!G){          //A2
5
6      __assert_fail;//f: failure instr.
7    }
8    else{
9
10     IO(...);   //computation & I/O
11   }
12
13 }
```

```
1  void func(...){
2  + my_xbegin();
3    G = g;
4    if(!G){
5  +   my_xabort();
6      __assert_fail;
7    }
8    else{
9  +   my_xend();
10     IO(...);
11   }
12 + my_xend();
13 }
```

Figure 7: A toy example adapted from Figure 2 (left-side) and its BugTM$_H$ transformation (right-side)

```
     if(_xtest() == 0){//no active Tx
         Retrytimes = 0;
         prev_status = -1;
retry:   if((status = _xbegin()) == _XBEGIN_STARTED){
           //Tx starts
         }else{
           //abort fallback handler, no active Tx at this point
            Retrytimes++;
            if(status==0x00||status==0x08){
            //unknown or capacity abort
              if(!(prev_status==0x00 && status==0x00) &&
                  !(prev_status==0x08 && status==0x08))
               { prev_status=status; goto retry;}
             }else if(status==0x06 || status==0xFF){
               if(Retrytimes < RetryThreshold)
                 {prev_status=status; goto retry;}
             }
             //continue execution in non-Tx mode
         }
     }
```

Figure 8: BugTM$_H$ `StartTx` wrapper function (`my_xbegin`)

not contain function calls, which avoids system calls and many trapping instructions, or loops, which avoids large memory footprints. The constraint of not containing function calls will be relaxed in Section 3.5.

Specifically, for every failure instruction f inside a function F, BugTM$_H$ puts a `StartTx` wrapper function right after the first function call instruction or loop-exit instruction or the entrance of F, whichever encountered first along every path tracing backward from f to the entrance of F. BugTM$_H$ puts `CommitTx` wrapper functions right before the exit of F, every function call in F, and every loop header instruction in F, unless the corresponding loop contains a failure instruction, in which case we want to extend re-execution regions for possible failures inside the loop.

Analysis for different failure instructions may decide to put multiple `StartTx` (`CommitTx`) at the same program location. In these cases, we will only keep one copy.

For the toy example in Figure 7, the intra-procedural BugTM$_H$ identifies Line 2 to put a `StartTx`, and identifies Line 9 and 12 to put `CommitTx`, as shown in the figure.

```
if(_xtest())
   _xend(); //terminate an active transaction
```

Figure 9: BugTM$_H$ `CommitTx` wrapper function (`my_xend`)

How to `StartTx` and `CommitTx` The above algorithm does not guarantee one-to-one pairing of the execution of `StartTx` and `CommitTx`, the second challenge discussed above. BugTM$_H$ addresses this through TestTx checkings conducted in `my_xbegin` and `my_xend`, BugTM$_H$ wrapper functions for `StartTx` and `CommitTx`. That is, `StartTx` will execute only when there is no active Txs, as shown in Figure 8; `CommitTx` will execute only when there exists an active Tx, as shown in Figure 9.

Overall, our design so far satisfies performance, correctness, and failure-recovery goals by guaranteeing a few properties. For performance, BugTM$_H$ guarantees that its Txs do not contain system/library calls or loops or nested Txs, and always terminate by the end of the function where the Tx starts. For correctness, BugTM$_H$ guarantees not to introduce crashes caused by unpairing `CommitTx`. For recovery capability, BugTM$_H$ makes the best effort in letting failures occur under active Txs.

3.4 Design for fallback and retry

Challenges It is not trivial to automatically and correctly generate fallback/retry code for all Txs inserted by BugTM$_H$. Since many Tx aborts may be unrelated to concurrency bugs, inappropriate abort handling could lead to performance degradation, hangs, and lost failure-recovery opportunities.

Solutions BugTM$_H$ will check the abort code and react to different types of aborts differently. Specifically, BugTM$_H$ implements the following fallback/retry strategy through its `my_xbegin` wrapper (Figure 8).

Aborts caused by `AbortTx` inserted by BugTM$_H$ indicates software failures. We should re-execute the Tx under HTM, hoping that the failure will disappear in retry (Line 14–17). To avoid endless retry, BugTM$_H$ keeps a retry-counter `Retrytimes` (Figure 8). This counter is configurable in BugTM$_H$, with the default being 1000000.

Data conflict aborts (Line 14–17) are caused by conflicting accesses from another thread. They are handled in the same way as above, because they could be part of the manifestation of concurrency bugs.

Unknown aborts and capacity aborts (Line 9–13) have nothing to do with concurrency bugs or software failures. In fact, the same abort code may appear repeatedly during retries, causing performance degradation without increasing the chance of failure recovery. Therefore, the fallback code will re-execute the Tx region in non-transaction mode once these two types of aborts are observed in two consecutive aborts. Nested Tx aborts would not be encountered by BugTM$_H$, because BugTM$_H$ Txs are non-nested.

The above wrapper function not only implements fallback/retry strategy, but also allows easy integration into the target software, as demonstrated in Figure 7.

3.5 Inter-procedural Designs and Others

The above algorithm allows no function calls or returns in Txs, keeping the whole recovery attempt within one function F. This is too conservative as many functions contain no trapping instructions and could help recovery.

To extend the re-execution region into callees of F, we put `my_xend` before every system/library call instead of every function call. To extend the re-execution region into the callers of F, we slightly change the policy of putting `my_xbegin`. When the basic algorithm puts `my_xbegin` at the entrance of F, the inter-procedural extension will find all possible callers of F, treat the callsite of F in its caller as a failure instruction, and apply `my_xbegin` insertion and `my_xend` insertion in the caller.

We then adjust our strategy about when to finish a BugTM$_H$ Tx. The basic BugTM$_H$ may end a Tx too early: by placing `my_xend` before every function exit, the re-execution will end in a callee function of F before returning to F and reaching the potential failure site in F. Our adjustment changes the `my_xend` wrapper inserted at function exits, making it take effect only when the function is the one which starts the active Tx.

Finally, as an optimization, we eliminate Txs that contain no shared-variable reads the failure instruction f has control or data dependency on. In these cases, the execution and outcome of f is deterministic during re-execution, and hence the failure cannot be recovered.

4 BugTM$_{\text{HS}}$

Rollback and re-execution techniques based on HTM (Section 3) and `setjmp`/`longjmp` [55] each has its own strengths and weaknesses. The former allows re-execution regions to contain shared variable writes, which is a crucial improvement over the latter in terms of failure recovery capability. However, it also has higher overhead than the latter. Furthermore, some operations not allowed inside an HTM Tx (e.g. `malloc`, `memcpy`, `pthread_cond_wait`), could potentially be correctly re-executed through software techniques [37, 45].

To combine the strengths of the above two approaches, we design BugTM$_{\text{HS}}$. The high level idea is that we apply ConAir to insert `setjmp` and `longjmp` recovery code into a program first[5]; and then, only at places where the growth of re-execution regions are stopped by shared-variable writes, we apply BugTM$_H$ to extend re-execution regions through HTM-based recovery.

[5] Intel TSX allows `setjmp`/`longjmp` to execute inside Txs.

Next, we will discuss in details how we carry out this high level idea to achieve the **union** of BugTM$_H$ and ConAir's recovery capability, while greatly enhancing the performance of BugTM$_H$.

Where to `setjmp` and `StartTx` ConAir and BugTM$_H$ insert `setjmp` and `StartTx` using similar algorithms, easing the design of BugTM$_{\text{HS}}$. That is, for every failure instruction f inside a function F, ConAir (BugTM$_H$) traverses backward through every path p that connects f with the entrance of F on CFG, and puts a `setjmp` wrapper function (`StartTx` wrapper function) right after the first appearance of a *killing instruction*. We will refer to this location as $\texttt{loc}_{\text{setjmp}}$ and $\texttt{loc}_{\texttt{StartTx}}$, respectively. For ConAir, the killing instructions include the entrance of F, writes to any global or heap variables, and a selected set of system/library calls; for BugTM$_H$, the killing instructions include the entrance of F, the loop-exit instruction, and all system/library calls [6].

BugTM$_{\text{HS}}$ slightly modifies the above algorithm. Along every path p, BugTM$_{\text{HS}}$ inserts the `setjmp` wrapper function at every $\texttt{loc}_{\text{setjmp}}$, where ConAir would insert it. In addition, BugTM$_{\text{HS}}$ inserts the `StartTx` wrapper function at $\texttt{loc}_{\texttt{StartTx}}$, when $\texttt{loc}_{\texttt{StartTx}}$ is farther away from f than $\texttt{loc}_{\text{setjmp}}$ (i.e., offering longer re-execution). Note that BugTM$_{\text{HS}}$ inserts `setjmp` at every location $\texttt{loc}_{\text{setjmp}}$ where ConAir would have inserted `setjmp` because every $\texttt{loc}_{\text{setjmp}}$ might be executed without an active hardware transaction due to unexpected HTM aborts and others. When $\texttt{loc}_{\text{setjmp}}$ is same as $\texttt{loc}_{\texttt{StartTx}}$, BugTM$_{\text{HS}}$ would only insert `setjmp` without inserting `StartTx` wrapper function.

Where to `CommitTx` BugTM$_{\text{HS}}$ inserts `CommitTx` wrapper functions exactly where BugTM$_H$ inserts them. Note that, BugTM$_{\text{HS}}$ inserts fewer `StartTx` than BugTM$_H$, and hence starts fewer Txs at run time. Fortunately, this does not affect the correctness of how BugTM$_{\text{HS}}$ inserts `CommitTx`, because the wrapper function makes sure that `CommitTx` executes only under an active Tx.

How to retry ConAir and BugTM$_H$ insert `longjmp` and `AbortTx` wrapper functions, which are responsible for triggering rollback-based failure recovery, using the same algorithm — right before a failure is going to happen as described in Section 2.2 and Section 3.2.

BugTM$_{\text{HS}}$ inserts its rollback function (Figure 10) at the same locations. We design BugTM$_{\text{HS}}$ rollback wrapper to first invoke HTM-rollback (i.e., `AbortTx`) if it is under an active transaction, which will allow a longer re-execution region and hence a higher recovery probability. The BugTM$_{\text{HS}}$ rollback wrapper invokes `longjmp` rollback if it is not under an active transaction. To make

```
if(_xtest())
  _xabort(0xFF); //terminate an active transaction
else //use longjmp for recovery
    if(longjmp_retry ++ < 1000000) // avoid endless retry
        longjmp(buf1,-1);
```

Figure 10: BugTM$_{\text{HS}}$ rollback wrapper function

sure that the program would not keep attempting hopeless recoveries, BugTM$_{\text{HS}}$ continues to use the HTM-abort statistics in the `StartTx` wrapper function shown in Figure 8 and continues to keep the `longjmp` retry count threshold shown in Figure 10.

For examples shown in Figure 2, 4, and 7, BugTM$_{\text{HS}}$ would insert both `setjmp` and `StartTx` into the buggy code regions, because `StartTx` would provide longer re-execution regions in all three cases. However, if the `*buf++ = ' ';` statement does not exist in Figure 4, BugTM$_{\text{HS}}$ would not insert `StartTx` there. Consequently, if failures happen, `longjmp` will be used for recovery.

Overall, we expect BugTM$_{\text{HS}}$ to improve the performance of BugTM$_H$ and improve the recovery capability of both BugTM$_H$ and ConAir. This will be confirmed through experiments in Section 7.

5 Failure Diagnosis

Previous recovery techniques like ConAir and naive system restart leave failure diagnosis completely to developers, which is often very time consuming. To address this limitation, we design BugTM$_{\text{HS}}$ to support failure diagnosis through the root-cause inference routine shown in Figure 11 and extra logging during recovery.

The root-cause inference shown in Figure 11 is mostly straightforward. The rationale of diagnosis based on the number of re-executions (Line 5 and 7) is the following. If the recovery success relies on a code region $\mathbb{C}$ in the failure thread to re-execute atomically, probably one re-execution attempt is sufficient, because another unserializable interleaving during re-execution is very rare. This case applies to RAR violation, as shown in Table 2. If the recovery success relies on something to happen in another thread, multiple re-executions are probably needed. This applies to WAW violations and order violations, as shown in Table 2.

Note that, BugTM$_{\text{HS}}$ and BugTM$_H$ could detect and recover the software from concurrency bugs before explicit failures getting triggered. As shown in Table 2, for several types of atomicity violation bugs, the retry would be triggered by HTM data-conflict aborts, instead of explicit failures. In these cases (Line 9), BugTM$_{\text{HS}}$ cannot affirmatively conclude that concurrency bugs have happened. It can only provide hints that certain types of atomicity violations may be the reason for HTM aborts. Along this line, future work could extend BugTM to contain more concurrency-bug detection capability, in addi-

[6] BugTM$_{\text{HS}}$ also combines the inter-procedural recovery of ConAir and BugTM$_H$ in a similar way. We skip details for space constraints.

```
Input: information from a successful recovery
if (timeout failures)
    output: deadlock
else if (other explicit failures)
    if (first re-execution succeeds)
      output: RAR atomicity violation
    else
      output: Order Violation or WAW atomicity violation
else if (implicit failures) //HTM data conflict aborts
    output: possible RAR, WAR, or RAW atomicity violations
```

Figure 11: Recovery-guided root-cause diagnosis

tion to its failure recovery capability.

BugTM$_{\text{HS}}$ also logs memory access type (read/write), addresses, values, and synchronization operations during re-execution, which helps diagnosis with no run-time overhead and only slight recovery delay.

Of course, some real-world concurrency bugs are complicated. However, complicated bugs can often be decomposed into simpler ones. Furthermore, some principles still hold. For example, if the re-execution succeeds with just one attempt, it is highly likely that an atomicity violation happened to the re-execution region.

6 Methodology

Implementation BugTM is implemented using LLVM infrastructure (v3.6.1). We obtained the source code of ConAir, also built upon LLVM. All the experiments are conducted on 4-core Intel Core i7-5775C (Broadwell) machines with 6MB cache, 8GB memory running Linux version 2.6.32, and O3 optimization level.

Benchmark suite We have evaluated BugTM on 29 bugs, including *all* the real-world bug benchmarks in a set of previous papers on concurrency-bug detection, fixing, and avoidance [17, 19, 41, 55, 56, 57]. They cover all common types of concurrency-bug root causes and failure symptoms. They are from server applications (e.g., MySQL database server, Apache HTTPD web server), client applications (e.g., Transmission BitTorrent client), network applications (e.g., HawkNL network library, HTTrack web crawler, Click router), and many desktop applications (e.g., PBZIP2 file compressor, Mozilla JavaScript Engine and XPCOM). The sizes of these applications range 50K — 1 million lines of code. Finally, our benchmark suite contains 3 extracted benchmarks: Moz52111, Moz209188, and Bank.

The goal of BugTM is to *recover* from production-run failures, **not** to *detect* bugs. Therefore, our evaluation uses previously known concurrency bugs that we know how to trigger failures. In all our experiments, the evaluated recovery tools do **not** rely on any knowledge about specific bugs in their failure recovery attempts.

Setups and metrics We will measure the recovery capability and overhead of BugTM$_H$ and BugTM$_{\text{HS}}$. We will also evaluate and compare with ConAir [55], the state of the art concurrency-bug recovery technique.

	RootCause	ConAir	BugTM$_H$	BugTM$_{\text{HS}}$
MySQL2011	AV$_{\text{RAR}}$	–	✓	✓
MySQL38883	AV$_{\text{RAR}}$	–	✓	✓
Apache21287	AV$_{\text{RAW}}$	–	✓	✓
Moz-JS18025	AV$_{\text{RAW}}$	–	✓	✓
Moz-JS142651	AV$_{\text{RAW}}$	–	✓	✓
Bank	AV$_{\text{WAR}}$	–	✓	✓
Transmission	OV	✓	–	✓
Total		1	6	7

Table 3: Recovery capability comparison (Moz-JS: Mozilla JavaScript Engine.)

To measure recovery capability, we follow the methodology of previous work [18, 55], and insert `sleeps` into software, so that the corresponding bugs will manifest frequently. We then run each bug-triggering workload with each tool applied for 1000 times.

To measure the run-time overhead. We run the original software **without** any `sleeps` with each tool applied. We report the average overhead measured during 100 failure-**free** runs, reflecting the performance during **regular** execution. We also evaluate alternative designs of BugTM, such as not conducting inter-procedural recovery, not excluding system calls from Txs, not excluding loops, etc. Due to space constraints, we only show this set of evaluation results on Mozilla and MySQL benchmarks, two widely used client and server applications.

7 Experimental Results

Overall, BugTM$_H$ and BugTM$_{\text{HS}}$ both have better recovery capability than ConAir, and both provide good performance. BugTM$_{\text{HS}}$ provides the best combination of recovery capability and performance among the three.

7.1 Failure recovery capability

Among all the 29 benchmarks, 9 cannot be recovered by any of the evaluated techniques, no matter ConAir or BugTM, and the remaining 20 can be recovered by at least one of the techniques (BugTM$_{\text{HS}}$ can recover all of these 20). Table 3 shows the result of 7 benchmarks where different tools show different recovery capability.

ConAir fails to recover from 6 out of 7 failures in Table 3, mainly because it does not allow shared-variable writes in re-execution regions. As a result, it cannot recover from any RAW or WAR atomicity bugs, and some RAR bugs, including the one in Figure 4.

BugTM$_H$ can successfully recover from all the 6 failures that ConAir cannot in Table 3. BugTM$_H$ cannot recover from the Transmission bug, because recovering this bug requires re-executing `malloc`, a trapping operation for Intel TSX but handled by ConAir. In fact, `malloc` is allowed in some more sophisticated TM designs [37, 45].

BugTM$_{\text{HS}}$ combines the strengths of BugTM$_H$ and ConAir, and hence can successfully recover from all 7

	Run-time Overhead			#`setjmp`	#`StartTx`		#`StartTx` per 10μs		Abort%	
	ConAir	BugTM$_H$	BugTM$_{HS}$	BugTM$_{HS}$	BugTM$_H$	BugTM$_{HS}$	BugTM$_H$	BugTM$_{HS}$	BugTM$_H$	BugTM$_{HS}$
MySQL2011	0.05%	0.13%	0.08%	643425	2746031	778024	2.3	0.7	0.01	0.01
MySQL3596	0.40%	3.10%	1.12%	144212	110476	39913	3.9	1.4	0.12	0.20
MySQL38883	0.40%	3.08%	1.11%	144119	110471	39904	3.9	1.4	0.11	0.19
Apache21287	0.55%	3.77%	3.00%	40023	72093	45520	22.8	14.5	0.08	0.11
Moz-JS18025	0.57%	9.03%	2.62%	3992	6850	1159	16.3	2.8	0.29	0.04
Moz-JS142651	0.76%	11.9%	5.30%	2145	9666	4007	30.4	12.6	0.33	0.17
Bank	0.15%	2.18%	2.95%	6	5	5	0.1	0.1	0.0	0.0
Moz-ex52111	0.47%	0.53%	0.41%	4	3	0	0.0	0.0	0.0	0.0
Moz-ex209188	0.12%	0.58%	0.77%	2	1	1	0.0	0.0	0.0	0.0
MySQL791	0.35%	1.98%	0.24%	48998	4948	602	2.5	0.4	0.35	0.01
MySQL16582	0.15%	3.03%	0.99%	269543	153532	31222	3.8	0.8	0.03	0.06
Click	0.57%	8.11%	3.60%	4681	5142	2123	18.7	8.1	0.96	0.12
FFT	0.05%	0.03%	0.14%	23	25	19	0.0	0.0	0.0	0.0
HTTrack	0.15%	0.64%	0.04%	9212	15649	1572	0.1	0.0	0.83	0.11
Moz-xpcom	0.38%	0.45%	0.03%	324	1933	154	0.0	0.0	0.31	0.51
Transmission	0.11%	0.22%	0.07%	1093	2123	919	0.1	0.0	0.56	0.40
zsnes	0.05%	0.03%	0.44%	10462	11737	372	0.5	0.0	0.13	0.23
HawkNL	0.09%	0.00%	0.15%	10	19	16	0.0	0.0	0.0	0.07
Moz-JS79054	0.84%	11.7%	4.20%	338	1325	360	9.4	2.6	0.23	0.44
SQLite1672	0.05%	0.98%	0.50%	6	3	3	0.1	0.1	0.0	0.06
Avg.	0.31%	3.08%	1.39%	-	-	-	-	-	-	-

Table 4: Overhead during regular execution and detailed performance comparison (red font denotes >3% overhead; #: count of dynamic instances; Abort%: percentage of aborted dynamic Txs.)

benchmarks in Table 3. It recovers the first 6 failures through HTM retries. It recovers from the Transmission failure through `longjmp` (it rolls back the `malloc` that cannot be handled by HTM-retry through `free`).

Unrecoverable benchmarks There are 9 benchmarks that no tools can help recover for mainly three reasons. Some of these issues go beyond the scope of failure recovery, yet others are promising to address in the future. First, two order violation benchmarks cause failures when the failure thread is unexpectedly slow. Therefore, re-executing the failure thread would not help correct the timing. Fortunately, both failures can be prevented by delaying resource deallocation, a prevention approach proposed before for memory-bug failures [29, 35]. Second, three benchmarks, Cherokee326, Apache25520, and MySQL169, cause failures that are difficult to detect (i.e., silent data corruption). Tackling them goes beyond the scope of failure recovery. Third, the remaining four failures cannot be recovered due to un-reexecutable instructions, which are promising to address. For example, Intel TSX does not support putting `memcpy`, `cond_wait`, or `I/O` into its Txs. More sophisticated TMs with OS support [37, 45] could help recover these failures.

7.2 Performance

Table 4 shows the regular-run overheads of applying BugTM schemes to 20 benchmarks, all the benchmarks that are recoverable by BugTM$_{HS}$.

BugTM$_H$ incurs more overhead, about 3% on average, than ConAir does, about 0.3% on average, mainly because a Tx is much more expensive than a `setjmp`.

Fortunately, BugTM$_{HS}$ wins most of the lost performance back, incurring 1.4% overhead on average and less than 3% for **all but 3** benchmarks. In the worst cases, it incurs 4.2% and 5.3% overhead for two benchmarks in Mozilla JavaScript Engine (JSE), a browser component with little I/O. If we apply BugTM$_{HS}$ to the whole browser, the overhead would be much smaller, as JSE never takes >20% of the whole page-loading time based on our profiling and previous work [31].

Comparing BugTM$_{HS}$ with BugTM$_H$, BugTM$_{HS}$ is faster mainly because it has greatly reduced the number of transactions at run time. For example, for the four benchmarks that incur the largest overhead under BugTM$_H$ (Moz-JS18025, Moz-JS142651, Click, and Moz-JS79054), BugTM$_{HS}$ reduces the #`StartTx` per 10μs from 9.4 — 30.4 to 2.6 — 12.6, and hence dropping the overhead from 8.11–11.9% to 2.6–5.3%.

Tx abort rate is less than 1% for all benchmarks, with more than 95% of all aborts being unknown aborts (timer interrupts, etc.). As Section 7.4 will show, abort rates and overhead are much worse in alternative designs.

Recovery time & Comparison with whole-program restart A successful BugTM failure recovery takes little time. In our experiments, the recovery of atomicity violations and deadlocks mostly takes less than 100 μ-seconds (median is 76 μ-seconds). The recovery of order violations takes slightly longer time, as it highly depends on how much `sleep` is inserted to trigger the failure. BugTM recovery is much faster than a system restart, which could take a few minutes or even more for complicated systems. It also avoids wasting already conducted computation and crash inconsistencies. For example, without BugTM, MySQL791 would crash the database after a table is changed but before this change is logged, leaving inconsistent persistent states.

Understanding BugTM$_H$ overhead The overhead of BugTM$_H$ differs among benchmarks, ranging from

	$BugTM_H$	Intra-proc	Trapping-Ins	Loop
Moz-xpcom	0.45% ✓	0.44% ✗	0.54% ✓	0.20% ✓
Moz-JS18025	9.03% ✓	7.01% ✓	16.8% ✓	11.3% ✓
Moz-JS79054	11.7% ✓	11.4% ✗	14.0% ✓	11.1% ✓
Moz-JS142651	11.9%✓	7.6% ✗	19.6% ✓	12.2%✓
MySQL791	1.98% ✓	1.50% ✓	11.4% ✓	11.5% ✓
MySQL2011	0.13% ✓	0.13% ✗	1.50% ✓	0.06% ✓
MySQL3596	3.10% ✓	3.05% ✓	108% ✗	2.63% ✓
MySQL16582	3.03% ✓	0.16% ✓	93.1% ✓	1.89% ✓
MySQL38883	3.08% ✓	3.04% ✓	106% ✗	2.52% ✓

Table 5: $BugTM_H$ vs. alternative designs (%: the overhead over baseline execution w/o recovery scheme applied; ✓: failure recovered; ✗: failure not recovered.)

0.00% to 11.9%. As TM researchers found before, performance in TM systems is often complicated [4, 34]. An indicating metrics for our benchmarks is the frequency of dynamic `StartTx`. As shown in the #`StartTx` per 10μs column of Table 4, $BugTM_H$ executes more than 1 `StartTx` per 10 micro second on average for 10 benchmarks, and incurs more than 1% overhead for 9 of them.

7.3 Diagnosis

$BugTM_{HS}$ can provide diagnosis information for all the 20 benchmarks that it can help recover from. For 13 benchmarks, recoveries through `longjmp` or HTM rollback are initiated right before explicit failures, for which $BugTM_{HS}$ provides accurate root-cause diagnosis following Figure 11. For the other 7, the recoveries are triggered by HTM data-conflict aborts, for which $BugTM_{HS}$ correctly suggests that there might be RAR, RAW, or WAR atomicity violations behind these aborts but cannot provide more detailed root-cause information.

$BugTM_{HS}$ provides the option to log memory accesses during failure recovery attempts initiated by `longjmp`. Evaluation shows that this extra logging incurs 1.01X – 2.5X slowdowns to failure recovery with no overhead to regular execution. The 2.5X slowdown happens during a fast half-microsecond recovery.

7.4 Alternative designs of BugTM

Table 5 shows the performance and recovery capability of three alternative designs of $BugTM_H$. Due to space constraints, we only show results on benchmarks in MySQL database server and Mozilla browser suite (non-extracted). Since $BugTM_H$ is the foundation of $BugTM_{HS}$, an alternative design that degrades the performance or recovery capability of $BugTM_H$ will also degrade $BugTM_{HS}$ accordingly as discussed below.

Inter-procedural vs. Intra-procedural $BugTM_H$ uses the inter-procedural algorithm discussed in Section 3.5. This design adds 0.00 – 4.3 % overhead to its intra-procedural alternative, as shown in Table 5. In exchange, there are 4 benchmarks in Table 5 that require inter-procedural re-execution of $BugTM_H$ to recover from. Among them, two can be recovered by ConAir and hence can still be recovered by intra-procedural $BugTM_{HS}$; the other two require inter-procedural $BugTM_{HS}$ to recover. Recovering MySQL2011, Moz-xpcom, Moz-JS79054 has to re-execute not only function F where failures occur, but also F's caller. As for Moz-JS142651, we need to re-execute a callee of F where a memory access involved in the atomicity violation resides.

Including trapping instructions in Txs Clearly, if $BugTM_H$ did not intentionally exclude system calls from its Txs, more Txs will abort. This alternative design hurts performance a lot, incurring around 100% overhead for three MySQL benchmarks shown in Table 5. Such design also causes $BugTM_{HS}$ to incur more than 20% overhead on these benchmarks. Furthermore, these aborts may hurt recovery capability, as they will cause corresponding Tx regions to execute in non-transaction mode to avoid endless aborts and hence lose the opportunity of failure recovery. This indeed happens for two benchmarks in Table 5. One of them will also fail to be recovered by $BugTM_{HS}$ under this alternative design.

Including loops in Txs could lead to more capacity aborts, which are indeed observed for all benchmarks in Table 5. The overhead actually does not change much for most benchmarks. Having said that, it raises the overhead of MySQL791 from 1.98% to 11.5%.

More Txs We also tried randomly inserting more `StartTx`. The overhead increases significantly. For Moz-JS142651, when we double, treble, and quadruple the number of dynamic Txs through randomly inserted Txs, the overhead goes beyond 30%, 100%, and 800%. The impact to $BugTM_{HS}$ would also be huge accordingly.

7.5 Discussion

As the evaluation and our earlier discussion show, BugTM does not guarantee to recover from all concurrency bug failures, particularly if the bug has a long error propagation before causing a failure. However, we believe BugTM, particularly $BugTM_{HS}$, would provide a beneficial safety net to most multi-threaded software with little deployment cost or performance loss.

Several practices can help further improve the benefit of BugTM. First, as discussed in Section 7.1, some improvements of HTM design would greatly help BugTM to recover from more concurrency-bug failures. Second, developers' practices of inserting sanity checks into software would greatly help BugTM. With more sanity checks, fewer concurrency bugs would have long error propagation and hence more concurrency-bug failures would be recovered by BugTM. Third, different from locks, which protect the atomicity of a code region only when the region and *all* its conflicting code are all protected by the same lock, BugTM can help protect a code region regardless how other code regions are written. Consequently, developers could choose to selectively apply BugTM to parts of software where he/she is least certain about synchronization correctness.

Finally, BugTM can be applied to software that is already using HTMs. BugTM will choose not to make its HTM regions nesting with existing HTM regions.

8 Related Work

Concurrency-bug failure prevention The prevention approach works by perturbing the execution timing, hoping that failure-triggering interleavings would not happen. It either relies on prior knowledge about a bug/failure [19, 27] to prevent the same bug from manifesting again, or relies on extensive off-line training [53, 51] to guide the production run towards likely failure-free timing. It is not suitable for avoiding production-run failures caused by previously unknown concurrency bugs. Particularly, the LiteTx work [51] proposes hardware extensions that are like lightweight HTM (i.e., without versioning or rollback) to constrain production-run thread interleavings, proactively prohibiting interleavings that have not been exercised during off-line testing. BugTM and LiteTx are fundamentally different on how they prevent/recover-from concurrency-bug failures and how they use hardware support.

Automated concurrency-bug fixing Static analysis and code transformation techniques have been proposed to automatically generate patches for concurrency bugs [17, 18, 25, 47]. They work at off-line and rely on accurate bug-detection results. A recent work [16] proposes a data-privatization technique to automatically avoid some read-after-write and read-after-read atomicity violations. When a thread may access the same shared variable with no blocking operations in between, this technique would create a temporary variable to buffer the result of the earlier access and feed it to the later read access. Although inspiring, this previous work is clearly different from BugTM. It does not handle many other types of concurrency bugs, including write-after-read and write-after-write atomicity violations and order violations. Furthermore, it relies on analyzing traces of previous execution of the program to carry out data privatization. The different usage contexts lead to different designs.

Failure recovery Rollback and re-execution have long been a valuable recovery [35, 44] and debugging [7, 20, 33, 43] technique. Many rollback-reexecution techniques target full system/application replay and hence are much more complicated and expensive than BugTM.

Feather-weight re-execution based on idempotency has been used before for recovering hardware faults [6, 9]. Using it to help recover from concurrency-bug failures was recently pioneered by ConAir [55]. BugTM greatly improved ConAir. BugTM$_H$ and ConAir use not only different rollback/reexecution mechanisms, but also completely different static analysis and code transformation. The `setjmp` and `longjmp` used by ConAir have different performance and correctness implications from `StartTx`, `CommitTx`, and `AbortTx`, which naturally led to completely different designs in BugTM$_H$ and ConAir.

Recent work leverages TM to help recover from transient hardware faults [21, 24, 49]. Due to the different types of faults/bugs these tools and BugTM are facing, their designs are different from BugTM. They wrap the whole program into transactions, which inevitably leads to large overhead (around 100% overhead [21, 49]) or lots of hardware changes to existing HTM [24], and different design about how/where to insert Tx APIs. They use different ways to detect and recover from the occurrence of faults, and hence have different Tx abort handling from BugTM. They either rely on **non**-existence of concurrency bugs to guarantee determinism [21] or only apply for single-threaded software [24, 49], which is completely different from BugTM.

Others Lots of research was done on HTM and STM [2, 3, 5, 11, 13, 14, 30, 36, 42]. Recent work explored using HTM to speed up distributed transaction systems [48], race detection [10, 54], etc. Previous empirical studies have examined the experience of using Txs, instead of locks, in developing parallel programs [38, 52]. They all look at different ways of using TM systems from BugTM.

9 Conclusions

Concurrency bugs severely affect system availability. This paper presents BugTM that leverages HTM available on commodity machines to help automatically recover concurrency-bug failures during production runs. BugTM can recover failures caused by all major types of concurrency bugs and incurs very low overhead (1.39%). BugTM does not require any prior knowledge about concurrency bugs in a program and guarantees not to introduce any new bugs. We believe BugTM improves the state of the art of failure recovery, presents novel ways of using HTM techniques, and provides a practical and easily deployable solution to improve the availability of multi-threaded systems with little cost.

10 Acknowledgments

We thank Wei Zhang and Linhai Song for their help in our experiments with ConAir. We are also grateful to the anonymous reviewers whose comments helped bring the paper to its final form. This project is funded by NSF (grants CNS-1563956, IIS-1546543, CNS-1514256, CCF-1514189, CCF-1439091) and CERES Center for Unstoppable Computing.

References

[1] Intel 64 and ia-32 architectures optimization reference manual. http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf. Accessed: 2016-07-30.

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA*, 2005.

[3] Tongxin Bai, Xipeng Shen, Chengliang Zhang, William N. Scherer, Chen Ding, and Michael L. Scott. A key-based adaptive transactional memory executor. In *IPDPS*, 2007.

[4] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *ISCA*, 2007.

[5] Dhruva R. Chakrabarti, Prithviraj Banerjee, Hans-J. Boehm, Pramod G. Joisha, and Robert S. Schreiber. The runtime abort graph and its application to software transactional memory optimization. In *CGO*, 2011.

[6] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent processor architecture. In *MICRO '11*.

[7] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *OSDI*, 2014.

[8] Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum, and Marek Olszewski. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS*, 2009.

[9] Shuguang Feng, Shantanu Gupta, Amin Ansari, Scott Mahlke, and David August. Encore: Low-cost, fine-grained transient fault recovery. In *MICRO '11*.

[10] Shantanu Gupta, Florin Sultan, Srihari Cadambi, Franjo Ivancic, and Martin Rotteler. Using hardware transactional memory for data race detection. In *IEEE IPDPS*, pages 1–11, 2009.

[11] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[12] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, et al. The ibm blue gene/q compute chip. *IEEE Micro*, 32(2):48–60, 2012.

[13] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2nd edition, 2010.

[14] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

[15] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[16] Jeff Huang and Charles Zhang. Execution privatization for scheduler-oblivious concurrent programs. In *OOPSLA*, 2012.

[17] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[18] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *OSDI*, 2012.

[19] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI* *https://code.google.com/archive/p/dimmunix/*, 2008.

[20] Samuel King, George Dunlap, and Peter Chen. Debugging operating systems with time-traveling virtual machines. Proceedings of USENIX ATC, 2005.

[21] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Haft: hardware-assisted fault tolerance. In *EuroSys*, 2016.

[22] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[23] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[24] Jianli Li, Qingping Tan, and Lanfang Tan. En-HTM: Exploiting hardware transaction memory for achieving low-cost fault tolerance. In *2013 Fourth International Conference on Digital Manufacturing & Automation*.

[25] Peng Liu, Omer Tripp, and Charles Zhang. Grail: Context-aware fixing of concurrency bugs. In *FSE*, 2014.

[26] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

[27] Brandon Lucia and Luis Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *ASPLOS*, 2013.

[28] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA*, 2008.

[29] Vitaliy B. Lvin, Gene Novark, and Emery D. Berger. Archipelago: Trading address space for reliability and security. In *ASPLOS*, 2008.

[30] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *HPCA*, 2006.

[31] Javad Nejati and Aruna Balasubramanian. An in-depth study of mobile browser performance. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 1305–1315, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

[32] PCWorld. Nasdaq's Facebook Glitch Came From Race Conditions. `http://www.pcworld.com/businesscenter/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html`.

[33] Gilles Pokam, Klaus Danne, Cristiano Pereira, Rolf Kassa, Tim Kranich, Shiliang Hu, Justin Emile Gottschlich, Nima Honarmand, Nathan Dautenhahn, Samuel T. King, and Josep Torrellas. Quickrec: prototyping an intel architecture extension for record and replay of multithreaded programs. In *ISCA*, 2013.

[34] Donald E. Porter and Emmett Witchel. Understanding transactional memory performance. In *ISPASS*, 2010.

[35] Feng Qin, Joe Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies — a safe method to survive software failure. In *SOSP*, 2005.

[36] R. Rajwar and J. R. Goodman. Transactional lock-free execution. In *ASPLOS*, 2002.

[37] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP*, 2007.

[38] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *PPoPP*, 2010.

[39] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *AADEBUG*, 2005.

[40] SecurityFocus. Software bug contributed to blackout. http://www.securityfocus.com/news/8016.

[41] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. Do I use the wrong definition? DefUse: Definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*, 2010.

[42] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra J. Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. In *ISCA*, 2007.

[43] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: Diagnosing production run failures at the user's site. In *SOSP*, 2007.

[44] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *SOSP*, 2011.

[45] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M. Swift, and Adam Welc. xcalls: safe I/O in memory transactions. In *EuroSys*, 2009.

[46] Haris Volos, Andres Jaan Tack, Michael M. Swift, and Shan Lu. Applying transactional memory to concurrency bugs. In *ASPLOS*, 2012.

[47] Yin Wang, Terence Kelly, Manjunath Kudlur, Stephane Lafortune, and Scott Mahlk. Gadara: dynamic deadlock avoidance for mult-threaded programs. In *OSDI*, 2008.

[48] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *SOSP*, 2015.

[49] Gulay Yalcin, Osman S Unsal, Adrian Cristal, Ibrahim Hur, and Mateo Valero. Symptomtm: Symptom-based error detection and recovery using hardware transactional memory. In *IEEE PACT*, pages 199–200, 2011.

[50] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi N. Bairavasundaram. How do fixes become bugs? In *FSE*, 2011.

[51] Jie Yu and Satish Narayanasamy. Tolerating concurrency bugs using transactions as lifeguards. In *IEEE Micro*, pages 263–274, 2010.

[52] Jiaqi Zhang, Wenguang Chen, Xinmin Tian, and Weimin Zheng. Exploring the emerging applications for transactional memory. In *Proceedings of the 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2008.

[53] Mingxing Zhang, Yongwei Wu, Shan Lu, Shanxiang Qi, Jinglei Ren, and Weimin Zheng. AI: a lightweight system for tolerating concurrency bugs. In *FSE*, 2014.

[54] Tong Zhang, Dongyoon Lee, and Changhee Jung. Txrace: Efficient data race detection using commodity hardware transactional memory. In *ASPLOS*, 2016.

[55] Wei Zhang, Marc de Kruijf, Ang Li, Shan Lu, and Karthikeyan Sankaralingam. ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution. In *ASPLOS*, 2013.

[56] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: Detecting concurrency bugs through sequential errors. In *ASPLOS*, 2011.

[57] Wei Zhang, Chong Sun, Junghee Lim, Shan Lu, and Thomas Reps. ConMem: Detecting Crash-Triggering Concurrency Bugs through an Effect-Oriented Approach. *ACM TOSEM*, 2012.