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

Yuxi Chen, Shu Wang, Shan Lu, *Member, IEEE*, and Karthikeyan Sankaralingam, *Member, IEEE*

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 applies transactional memory techniques for concurrency-bug recovery in production runs. Requiring **no** knowledge about where are concurrency bugs, BugTM uses static analysis and code transformation to enable BugTM-transformed software to recover from a concurrency-bug failure by rolling back and re-executing the recent history of a failure thread. BugTM is instantiated as three schemes that have different trade-offs in performance and recovery capability: BugTM$_H$ uses existing hardware transactional memory (HTM) support, BugTM$_S$ leverages software transactional memory techniques, and BugTM$_{\mathrm{HS}}$ is a software-hardware hybrid design. 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.

Index Terms—Concurrency bugs, transactional memory, failure recovery, software availability

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 [2] and have caused disastrous failures [3], [4], [5]. 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 [6]. 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.

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 [7], [8]. Even with support from operating systems changes, periodic full-blown checkpointing still often incurs more than 10 percent overhead [7].

A recently proposed recovery technique, ConAir, conducts single-threaded re-execution and register-only checkpointing [9]. As shown in Fig. 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 percent 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_{kill}) for correctness concerns, severely hurting its chance to recover many failures.

This limitation can be demonstrated by the real-world example in Fig. 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

- Y. Chen, S. Wang, S. Lu are with the Department of Computer Science, the University of Chicago, Chicago, IL 60637. E-mail: {chenyuxi, shuwang, shanlu}@uchicago.edu.
- K. Sankaralingam is with the Department of Computer Science, University of Wisconsin–Madison, Madison, WI 53706. E-mail: karu@cs.wisc.edu.

*Manuscript received 5 June 2018; revised 2 Sept. 2018; accepted 17 Oct. 2018. Date of publication 23 Oct. 2018; date of current version 10 Apr. 2019. (Corresponding author: Yuxi Chen.)
Recommended for acceptance by F. Qin.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2877656*

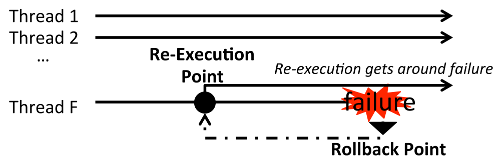


Fig. 1. Single-threaded recovery for concurrency bugs.

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

Fig. 2. A real-world concurrency bug from Mozilla.

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 (A_1 in Fig. 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.

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 Fig. 3. This paper presents BugTM, a set of transactional-memory (TM) inspired designs that thoroughly explore the design space of concurrency-bug failure recovery, also shown in Fig. 3. It greatly improves the combination of recovery-capability and performance over existing techniques, while still guarantees correctness.

1.2.1 Three BugTM Designs

BugTM explores 3 designs, all implemented as compiler passes that automatically instrument software at the byte-code level. The instrumented software conducts checkpoint, rollback, and re-execution for failure recovery in different ways across these three designs, as shown in Table 1.

Hardware BugTM, short for BugTM$_H$, uses hardware transactional memory (HTM) techniques[1] *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$_H$.

Software BugTM, short for BugTM$_S$, extends ConAir through software transactional memory (STM) version-management techniques. When a failure is to happen, just like ConAir, a `longjmp` rolls back register content of the failing thread to the latest `setjmp`. Different from ConAir, selected memory checkpoints[2] are conducted at some `setjmp` locations, offering the option to rollback some memory content before re-execution. Furthermore, BugTM$_S$ conducts *deferred-write* code transformation that delays some shared-variable writes to further address ConAir's failure-recovery limitations.

1. This paper's implementation is based on Intel TSX. However, the principles apply to other vendors' HTM implementations.
2. That is, backing up selected variables using undo logs.

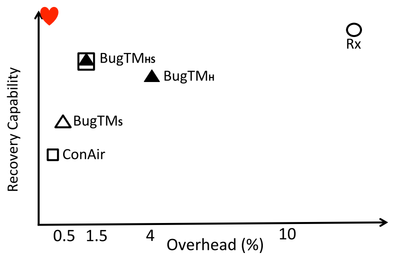


Fig. 3. Design space of concurrency-bug failure recovery (Heart: non-existing optimal design; Rx [7] changes OS).

TABLE 1
Design Comparisons

	ReExecution Point	RollBack Point	Checkpoint Memory ?	ReExecution contains S.W.?
ConAir	`setjmp`	`longjmp`	✗	✗
BugTM$_H$	`StartTx`	`AbortTx`	✓	✓
BugTM$_S$	`setjmp`	`longjmp`	✓	✓✗
BugTM$_{HS}$	`setjmp` **or** `StartTx`	`longjmp` **or** `AbortTx`	✓✗	✓

S.W.: shared-variable writes

Hybrid BugTM, short for BugTM$_{HS}$, uses HTM 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.

1.2.2 Three Design Tradeoffs

These three BugTM designs offer different performance and recovery-capability tradeoffs, as illustrated by Fig. 3.

BugTM$_H$ and BugTM$_S$ both support longer re-execution regions than ConAir by allowing shared-variable writes inside re-execution regions, and hence both achieve better recovery capability than ConAir. BugTM$_H$ tends to have better recovery capability than BugTM$_S$ benefiting from HTM, which we will explain in details in Sections 3 and 5. On the other hand, BugTM$_H$ and BugTM$_S$ are both slower than ConAir. For BugTM$_H$, this is because HTM costs more than `setjmp/longjmp`. For BugTM$_S$, this is because BugTM$_S$ takes extra lightweight memory checkpoints. Overall, their performance is still good, much better than full-blow memory checkpointing.

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

1.2.3 Challenges

At the first glance, transactional memory techniques provides a powerful mechanism for concurrency control and rollback-reexecution, automatically inserted transactions can likely help both proactively prevent failures by avoiding certain conflicting data accesses and reactively recover failures by automated rollback and re-execution. Previous work [10] also showed that TM can be used to *manually* fix concurrency bugs and concluded that TMs often offer conceptually simpler patches than locks, although the performance may suffer[3]*after* they are detected.

However, applying TM techniques for production-run failure recovery is not trivial. The three BugTM designs all need to handle performance challenges (low runtime overhead), correctness challenges (retain correctness for non-failure runs), and failure recovery challenges (survive software from failure run) encountered by *automatically* inserting transactions to help tackle *unknown* concurrency bugs during *production runs*, which wouldn't be encountered by *manually* fixing *already detected* bugs *off-line*:

Performance Challenges. Full-blown software transactional memory techniques can incur several times of slowdowns [11]. Consequently, BugTM$_S$ has to adapt and selectively apply STM techniques for its production-run uses. HTM is much faster than STM. However, high frequency of hardware transaction uses would still cause large overhead unacceptable for production runs. Unsuitable content of transactions, like trapping instructions,[4] high levels of transaction nesting, and long loops, would also cause performance degradation due to repeated and unnecessary transaction aborts. Consequently, BugTM$_H$ has to carefully insert transactions to avoid excessive slowdowns.

Correctness Challenges. BugTM$_S$ needs to guarantee that its adapted STM techniques do not modify software semantics. BugTM$_H$ also faces challenges, as unpaired transaction-start and transaction-commit could cause software to crash. Furthermore, deterministic aborts, such as those caused by trapping instructions, could cause software to hang if not well handled.

Failure Recovery Challenges. In order for transactions 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 transaction-abort handlers to correctly process the corresponding transaction aborts.

In Sections 3, 4, and 5, we will show details how BugTM addresses those challenges from HTM perspective.

1.2.4 Results

We have conducted a thorough evaluation for BugTM$_H$, BugTM$_S$, and BugTM$_{HS}$ using 29 real-world concurrency bugs, which contain *all* the concurrency bugs used by a set of recent papers on concurrency bug detection and avoidance [9], [12], [13], [14], [15], [16]. Our evaluation shows that BugTM schemes can recover from more concurrency-bug failures than previous state of the art, ConAir, while still keeping good run-time performance—3.08, 0.42, and 1.39 percent overhead on average for BugTM$_H$, BugTM$_S$, and BugTM$_{HS}$. In addition, BugTM$_S$ can provide useful failure diagnosis information.

Overall, BugTM greatly improves the state of art in production-run failure recovery for concurrency bugs by offering easily deployable techniques. It provides three valuable points in the design space of production-run failure recovery. BugTM$_H$ and BugTM$_{HS}$ also present a novel way of using HTM. Instead of using transactions to replace existing lock synchronization, BugTM$_H$ and BugTM$_{HS}$ automatically insert 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.

3. At that time [10], HTM was not available on commodity machines.

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

2 Background

2.1 Transactional Memory (TM)

TM is a widely studied parallel programming construct [17], [18]. Developers can wrap a code region in a transaction (Tx), and the underlying TM system guarantees its atomicity, consistency, and isolation. Hardware transactional memory provides much better performance than its software counterpart (STM), and has been implemented in IBM [19], Sun [20], and Intel commercial processors [21].

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.

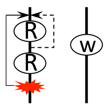
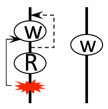
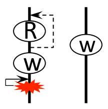
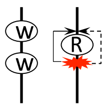
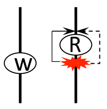
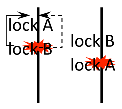
Recovery Capability Limitations and Killing Writes w_{kill}. ConAir does not allow its re-execution regions to contain any writes to shared variables. Consequently, many of its re-execution points (i.e., `setjmp` locations) are placed right after shared-variable writes, which we refer to as *killing writes* or w_{kill}. In many cases, ConAir could not recover from a failure because a successful recovery demands killing writes to be re-executed.

ConAir fundamentally cannot recover *any* RAW[5] violations (e.g., the bug in Fig. 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 they are killing writes for ConAir.

Even for those root-cause types that ConAir can handle in Table 2, its recovery capability is limited, because a killing write may exist between the failure location and the ideal re-execution point. For example, the RAR atomicity

5. (R/W)A(R/W) is short for (Read/Write)-after-(Read/Write).

TABLE 2
Common Types of Concurrency Bugs and how BugTM$_H$ and ConAir Attempt to Recover from them

	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	✓	–	–	✓	✓	✓

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.

violation in Fig. 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 percent of concurrency bugs lead to four types of failures [15]: 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 performance and failure-recovery capability.

Strawman Approaches. One approach is to chunk software to many segments and put every segment inside a hardware Tx [23]. 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 Fig. 2. It contributes to more than 70 percent of non-deadlock concurrency bugs based on empirical studies [2], 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)(d), 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 Fig. 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` → `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 Fig. 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

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

Fig. 4. A real-world concurrency bug from MySQL.

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

Fig. 5. A real-world OV bug (simplified from `Transmission`).

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

Fig. 6. BugTM$_H$ AbortTx wrapper function (my_xabort).

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 Fig. 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.

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 9.

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 Section 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 Fig. 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[6] 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 Fig. 7 as an example, we want to put A_1 and A_2, both accessing global variable `G`, into a Tx together with

6. 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 }
```

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

`__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 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 Fig. 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.

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 Fig. 8; `CommitTx` will execute only when there exists an active Tx, as shown in Fig. 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

```
 1         if(_xtest() == 0){//no active Tx
 2             Retrytimes = 0;
 3             prev_status = -1;
 4   retry:    if((status = _xbegin()) == _XBEGIN_STARTED){
 5               //Tx starts
 6             }else{
 7               //abort fallback handler, no active Tx at this point
 8                Retrytimes++;
 9                if(status==0x00||status==0x08){
10                //unknown or capacity abort
11                  if(!(prev_status==0x00 && status==0x00) &&
12                      !(prev_status==0x08 && status==0x08))
13                   { prev_status=status; goto retry;}
14               }else if(status==0x06 || status==0xFF){
15                  if(Retrytimes < RetryThreshold)
16                    {prev_status=status; goto retry;}
17               }
18               //continue execution in non-Tx mode
19             }
20         }
```

Fig. 8. BugTM$_H$ `StartTx` wrapper function (`my_xbegin`).

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 (Fig. 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 (Lines 14–17). To avoid endless retry, BugTM$_H$ keeps a retry-counter `Retrytimes` (Fig. 8). This counter is configurable in BugTM$_H$, with the default being 1,000,000.

Data conflict aborts (Lines 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 (Lines 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 Fig. 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

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

Fig. 9. BugTM$_H$ `CommitTx` wrapper function (`my_xend`).

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 Roadmap for Further Exploration

HTM in BugTM$_H$ and `setjmp/longjmp` in previous state-of-the-art ConAir [9] are almost at the two ends of the design spectrum. While the former provides much better recovery capability, it has higher overhead than the latter. Furthermore, HTM disallows certain operations in a Tx (e.g., `malloc`, `memcpy`, `pthread_cond_wait`), which could be addressed by software techniques [24], [25].

To further explore the design space, the next two sections will explore two designs to combine the strengths of BugTM$_H$ and ConAir.

One design, BugTM$_S$, is to implement some functionalities of HTM, tailored for concurrency-bug recovery, as software extensions for ConAir. Looking at the three TM principles of conflict detection, conflict resolution, and version management, we decide to try the latter two and give up conflict detection, as conflict detection is too expensive to implement in software. The conflict-resolution technique for shared-variable reads can provide extra options for ConAir's re-execution: reading the latest copy means delaying the current Tx (thread), whereas reading an earlier one using an *undo log* means delaying the conflicting one. The version-management technique for shared-variable writes, which completely does not exist in ConAir, can extend the types of regions that can be reexecuted for recovery in ConAir. This design will be discussed next in Section 5.

The other design, BugTM$_{HS}$, directly combines ConAir and BugTM$_H$. This design is feasible as Intel TSX allows `setjmp/longjmp` to execute inside Txs.. Therefore, we can actually apply BugTM$_H$ to a program already hardened by ConAir or any `setjmp/longjmp` recovery scheme and

```
 1   //Thread-1                       1 //Thread-2
 2   if(thd->proc){                   2
 3 +     tmp = buf;                   3
 4 -     *buf = ' ';                  4 thd->proc = NULL;
 5       buf++;
 6
 7       strcat(buf,thd->proc);//failure site
 8 +     *tmp = ' '; //moved killing write
 9                //used to be *buf=' '
10   }
```

Fig. 10. BugTM$_S$ *deferred write* transformation, denoted by '+' '-', makes a ConAir-unrecoverable bug recoverable.

obtain the *union* of each component's recovery capability, which we will discuss in more details in Section 6.

We do not explore combining ConAir and BugTM$_S$, because BugTM$_S$ itself is already a direct extension of ConAir. We also do *not* explore combining BugTM$_H$ and BugTM$_S$, because this combination will be worse than combining BugTM$_H$ and ConAir (i.e., BugTM$_{HS}$)—in terms of performance, BugTM$_S$ is slower than ConAir; in terms of recovery capability, the advantage of BugTM$_S$ over ConAir is mostly overshadowed by BugTM$_H$.

5 BugTM$_S$

This section will focus on extending the basic `setjmp/longjmp` recovery scheme ConAir with two TM techniques (1) *deferred write* version management; and (2) *undo log* rollback. Our implementation will not rely on HTM and is purely based on compiler techniques. The resulting tool BugTM$_S$ not only improves the failure recovery capability of ConAir with negligible performance impact, but also well complements BugTM$_H$ by offering better performance and more design flexibility at the cost of losing some recovery capability owned by BugTM$_H$.

5.1 Deferred Writes for Failure-Unrelated w_{kill}

Fig. 10 shows an RAR atomicity violation, where the `NULL` assignment from Thread-2 could cause Thread-1 to crash at Line 7. Theoretically, ConAir can recover RAR atomicity-violation failures. However, with a killing write, which is actually unrelated to the bug, at Line 4, ConAir cannot extend its re-execution region to include both reads of `thd->proc` in Thread-1 and hence cannot recover from the failure.

To address this problem, BugTM$_S$ tries moving failure-unrelated w_{kill} to after the failure instruction, emulating the *deferred write* version-management technique in TM, so that the re-execution region can go beyond these killing writes.

5.1.1 Feasibility Checking

For each w_{kill} and the corresponding failure site f, BugTM$_S$ checks two things: (1) whether moving w_{kill} would change program semantics; and (2) whether the moving will cut short other failure sites' re-execution regions. If w_{kill} fails either checking, it is not moved.

The second checking is straightforward. For the first condition, BugTM$_S$ collects all instructions along any path from w_{kill} to f, and checks whether there exists any write-after-write, read-after-write, or write-after-read dependency between any of such instruction with w_{kill}.

```
- W_KILL
+ flag = TRUE;
  ...
+ if(flag){
+    W_KILL //new location
+    flag = FALSE;
+ }
```

Fig. 11. Moving a killing write (`flag` is initialized as `FALSE`).

If there is no such dependency, moving w_{kill} is guaranteed not to change program semantics.[7] If there exists such a dependency upon global/heap variables, we give up the moving. If the dependency is upon a stack variable, such as `buf` in Fig. 10, we try code transformation to eliminate the dependency. Note that, since w_{kill} writes to a shared variable, the stack variable dependency here must be a write-after-read dependency as the one between Lines 4 and 5 in Fig. 10.

To eliminate the write-after-read dependency between w_{kill} and i on a stack variable v_s, BugTM$_S$ will create temporary stack variable tmp to keep a copy of v_s at the original code location of w_{kill}, move w_{kill}, and let the moved w_{kill} read from tmp instead of v_s, as demonstrated by Fig. 10.

5.1.2 Moving the w_{kill}

To make sure the moved w_{kill} will execute for the same number of times as in the original program, BugTM$_S$ conducts the following analysis and transformation:

First, check if w_{kill} and f are inside one function F with neither inside a loop in F. If not, we give up the move.

Second, collect all the basic blocks $\mathbb{B}$ in F that are on path from w_{kill} to f, and copy w_{kill} to every edge that connects a basic block inside $\mathbb{B}$ to a basic block outside $\mathbb{B}$. This guarantees that the new location of w_{kill} will be touched exactly once in F, either immediately after f or immediately when there is no chance for f to execute. This way, w_{kill} will get a chance to execute, even if f is not executed.

Third, a stack variable is introduced to make sure that the newly moved w_{kill} would not execute if its original location was not touched, as shown in Fig. 11.

Now BugTM$_S$ can recover from some ConAir-unrecoverable failures, like the one shown in Fig. 10. It has almost no performance impact to the original ConAir, and guarantees to preserve program semantics.

5.2 Undo Log for Failure-Related Killing Writes

When killing writes are dependent upon by the corresponding failure instruction, which are true for all RAW violations and WAR violations, *deferred write* does not apply. For these cases, BugTM$_S$ enhances ConAir by offering an extra mode of rollback: ConAir only rolls back registers for re-execution; BugTM$_S$ offers checkpointing and rolling back the content of selected shared-memory locations, emulating *undo log* in TM. This option can help recover from some Read-After-Write (RAW) atomicity violations, while preserving program semantics and introducing little overhead.

Basic Algorithm. Fig. 12a shows a toy example of RAW atomicity violation: if another thread changes the value of

7. This guarantee holds based on the fact that almost all architectures, including Alpha, ARM, POWER, SPARC, x86, and many others, allow compilers to reorder stores to execute after independent loads.

```
 1 g1 = 1;
 2
 3
 4
 5
 6 tmp = g1;
 7
 8
 9 if (!tmp){
10
11    ASSERTFAIL;
12 //failure site
13 }
```

(a) Base

```
 1 g1 = 1;
 2 setjmp;
 3
 4
 5
 6 tmp = g1;
 7
 8
 9 if (!tmp){
10    longjmp;
11    ASSERTFAIL;
12 //failure site
13 }
```

(b) ConAir

```
 1 g1 = 1;
 2 ckpt_g1 = 1;
 3 ret=setjmp;
 4
 5 if(ret!=-1)
 6   tmp = g1;
 7 else
 8   tmp = ckpt_g1;
 9 if (!tmp){
10    longjmp;
11    ASSERTFAIL;
12 //failure site
13 }
```

(c) BugTM$_S$

Fig. 12. Memory-checkpoint example.

g1 from 1 to 0 between the write on Line 1 and the read on Line 6, an assertion failure could happen. ConAir cannot recover from this failure, because the re-execution will start after the w_{kill} in Line 1 and can never change the failure-triggering value returned by Line 6 in Fig. 12b. However, if the value of g1 could be checkpointed right at Line 1, as shown in Fig. 12c, the failure could be recovered.

In general, taking a memory checkpoint is straightforward: create a local variable ckpt_g1 and copy the right hand side of the g1-assignment to ckpt_g1 right before setjmp.

Making re-execution use the checkpointed values can be achieved through code transformation. The return value of setjmp is -1 only when it is jumped to from a longjmp, indicating re-execution. As shown in Fig. 12c Lines 5–8, BugTM$_S$ makes the read of g1 conditional on this return value: the read uses the value in ckpt_g1 during re-execution and uses the up-to-date value in g1 during regular execution.

The above BugTM$_S$ transformation can successfully recover from the failure on Line 11 in Fig. 12c, because the checkpointed-reexecution essentially guarantees the RAW atomicity between Lines 1 and 6. This transformation also guarantees to preserve the original program semantics during re-execution: its re-execution is equivalent with what the original program would behave if the re-executed region was executed instantaneously right after the setjmp.

Final Algorithm. When encounters a w_{kill} which the failure site f depends upon, BugTM$_S$ checks whether there exists a read r that satisfies all of the following conditions: (1) r may read from the same memory location written by w_{kill}; (2) f depends on r; (3) r and w_{kill} are inside the same basic block. If such a read r is found, BugTM$_S$ transforms the code region between w_{kill} and r by (1) recording the setjmp return value to a thread-local variable sj_ret; (2) taking checkpoints right before setjmp for all the global/heap variables read between w_{kill} and r including r, no matter related to the failure or not, following their load order; (3) making these accesses conditionally read from either the checkpoint or the up-to-date memory location based on sj_ret.

Note that, we need to checkpoint multiple global/heap variables in their original load order, because some architectures do not allow compilers to re-order loads for memory-consistency concerns (e.g., x86). For a similar reason, we only handle r and w_{kill} inside the same basic block, because otherwise there could be inconsistent load orders among different paths from w_{kill} to f.

As an optimization, when there are multiple memory reads that BugTM$_S$ needs to checkpoint, BugTM$_S$ simply creates a clone of the region from w_{kill} to the end of its basic block, makes every cloned global/heap read gets its value from the checkpoint, and switches between the cloned and the original version based on sj_ret.

When integrating with the original rollback scheme of ConAir, BugTM$_S$ configures the re-execution to use the checkpoints, if they exist, in the first re-execution attempt, and switch to not using checkpoints for following attempts. Since the re-execution using checkpoints is deterministic, there is no point for more attempts if the first attempt fails.

Limitations. This extension does not allow BugTM$_S$ to recover from write-after-read atomicity violations; and may not fundamentally recover from a read-after-write failure. Take the bug in Fig. 2 as an example, by using the checkpointed value of s->table at Line 4, BugTM$_S$ will recover from the original failure on Line 5. However, after the re-execution ends at Line 4, the execution will continue using the update-to-date value of s->table, which is NULL. Software probably will still fail, just at a later point. To fundamentally recover from this failure, we will need BugTM$_H$.

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

As discussed in Section 4, another interesting design point that can leverage both the performance strength of setjmp/longjmp and the recovery strength of HTM is to use them both. The high level idea is that we can apply ConAir to insert setjmp and longjmp recovery code into a program first; and then, only at places where the growth of re-execution regions are stopped by killing writes, we apply BugTM$_H$ to extend re-execution regions through HTM-based recovery.

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 loc$_{\text{setjmp}}$ and loc$_{StartTx}$, 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.[8]

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

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

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

Fig. 13. BugTM$_{\mathrm{HS}}$ rollback wrapper function.

Where to `CommitTx`. BugTM$_{\mathrm{HS}}$ inserts `CommitTx` wrapper functions exactly where BugTM$_H$ inserts them. Note that, BugTM$_{\mathrm{HS}}$ inserts fewer `StartTx` than BugTM$_H$, and hence starts fewer Txs at run time. Fortunately, this does not affect the correctness of how BugTM$_{\mathrm{HS}}$ 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 Sections 2.2 and 3.2.

BugTM$_{\mathrm{HS}}$ inserts its rollback function (Fig. 13) at the same locations. We design BugTM$_{\mathrm{HS}}$ 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$_{\mathrm{HS}}$ rollback wrapper invokes `longjmp` rollback if it is not under an active transaction. To make sure that the program would not keep attempting hopeless recoveries, BugTM$_{\mathrm{HS}}$ continues to use the HTM-abort statistics in the `StartTx` wrapper function shown in Fig. 8 and continues to keep the `longjmp` retry count threshold shown in Fig. 13.

For examples shown in Figs. 2, 4, and 7, BugTM$_{\mathrm{HS}}$ 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 Fig. 4, BugTM$_{\mathrm{HS}}$ would not insert `StartTx` there. Consequently, if failures happen, `longjmp` will be used for recovery.

Overall, we expect BugTM$_{\mathrm{HS}}$ 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 9.

7 FAILURE DIAGNOSIS

Previous recovery techniques, like ConAir [9], leave failure diagnosis completely to the developers, which is often very time consuming. BugTM supports failure diagnosis through the root-cause inference routine shown in Fig. 14 and extra logging during recovery.

The root-cause inference algorithms for BugTM$_H$ and BugTM$_S$ are shown in Figs. 14a and 14b, respectively. These two algorithms both use the time-out failure symptom to infer deadlock root cause as shown in Line 2 of both algorithms. However, the other parts of the algorithms differ from each other.

For BugTM$_H$, the manifestation of a concurrency bug could lead to a HTM abort due to either (1) an explicit failure inside a HTM transaction (e.g., an assertion gets violated) or (2) a data conflict abort caused by the buggy data race inside a HTM transaction. As illustrated in Table 2, order violations and WAW atomicity violations are much more likely to lead to explicit failure aborts (Lines 4–5 in Fig. 14a), while RAR, RAW, and WAR atomicity violations are much more likely to lead to data conflict aborts (Lines 6–7 in Fig. 14a).

It is difficult for BugTM$_H$ to provide more detailed root-cause inference. Furthermore, in case of a data conflict abort, BugTM$_H$ actually cannot conclude for sure whether the abort is caused by a benign data race or a failure-inducing data race. BugTM$_H$ can only suggest that, if the abort was caused by a buggy race, the buggy race is much more likely to be RAR, RAW, or WAR atomicity violations than order violations or WAW atomicity violations (Lines 6–7 in Fig. 14a). As we will see, this is the major diagnosis-capability limitation of BugTM$_H$, particularly in comparison with BugTM$_S$.

For BugTM$_S$, we can make root-cause triage based on the number of re-executions that were needed for a successful recovery.

When the recovery requires only one re-execution, the root cause is either RAW atomicity violation or RAR atomicity violation (Lines 4–8 in Fig. 14b). In both cases, unserializable interleaving is unlikely to occur again during the re-execution of the expected-to-be-atomic code region, and hence one re-execution should work. We can further differentiate these two types of root causes, as the recovery of RAW atomicity violation requires the use of undo-log, as discussed in Section 5.2 (Lines 5–6 in Fig. 14b).

When the recovery requires multiple re-execution attempts, the root cause is either an order violation or a WAW atomicity violation (Lines 9–10 in Fig. 14b). In case of an order violation, the failure thread, which executes unexpectedly fast, is waiting for the unexpectedly slow thread to catch up, which is likely to take more than one re-execution. In case of a WAW atomicity violation (e.g., code region $w_1 - -w_2$ unserializably interleaved by a read r), the failure thread rolls back and re-executes r, hoping that the re-executed r would occur after w_2, instead of in between w_1 and w_2. However, when the other thread would execute w_2 is

```
1 Input: information from a successful recovery
2 if (timeout failures)
3     output: deadlock
4 else if (other explicit failures)
5     output: Order Violation or WAW atomicity violation
6 else if (implicit failures) //HTM data conflict aborts
7     output: maybe RAR, WAR, or RAW atomicity violations
```

(a) diagnosis for BugTM$_H$

```
1  Input: information from a successful recovery
2  if (timeout failures)
3      output: deadlock
4  else if (first re-execution succeeds)
5      if(with checkpoint)
6          output: RAW atomicity violation
7      else
8          output: RAR atomicity violation
9  else if (re-execution succeeds after multiple attempts)
10     output: Order Violation or WAW atomicity violation
```

(b) diagnosis for BugTM$_S$

Fig. 14. Recovery-guided root-cause diagnosis for BugTM$_H$ and BugTM$_S$.

unpredictable. Consequently, it might take multiple re-executions of r for the recovery to succeed.

Note that, BugTM$_S$ cannot recover from failures caused by WAR atomicity violations, and hence WAR atomicity violations are not part of the algorithm in Fig. 14b.

For BugTM$_{HS}$, how much diagnostic information can be provided depends on whether a failure is recovered through HTM retries or `setjmp/longjmp`. When the recovery is through HTM retries, the root-cause inference routine in Fig. 14a applies; on the other hand, the inference routine in Fig. 14b applies.

Comparison and Discussion. As we can see, BugTM$_H$ offers less diagnostic information than BugTM$_S$, mainly because there are a wide variety of reasons behind its transaction aborts (Line 7 in Fig. 14a).

All schemes of BugTM further support failure diagnosis by logging memory access type (read/write), addresses, values, and synchronization operations during re-execution, which helps failure diagnosis with no run-time overhead and only slight recovery delay.

Some real-world concurrency bugs are complicated and may go beyond the categories we discussed above (e.g., multi-variable atomicity violations). 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 a code region inside the re-execution region.

8 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 6 MB 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 [9], [12], [13], [14], [15], [16]. 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 three BugTM designs. We will also evaluate and compare with ConAir [9], the state of the art concurrency-bug recovery technique.

TABLE 3
Recovery Capability Comparison

	RootCause	ConAir	BugTM$_S$	BugTM$_H$	BugTM$_{HS}$
MySQL2011	AV$_{RAR}$	–	✓	✓	✓
MySQL38883	AV$_{RAR}$	–	✓	✓	✓
Apache21287	AV$_{RAW}$	–	✓*	✓	✓
Moz-JS18025	AV$_{RAW}$	–	✓*	✓	✓
Moz-JS142651	AV$_{RAW}$	–	–	✓	✓
Bank	AV$_{WAR}$	–	–	✓	✓
Transmission	OV	✓	✓	–	✓
Total		1	3+	6	7

**: failures partly recovered; Moz-JS: Mozilla JavaScript Engine.*

To measure recovery capability, we follow the methodology of previous work [9], [26], 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.

9 Experimental Results

Overall, three BugTM schemes all have better recovery capability than ConAir, with BugTM$_{HS}$ being the best and BugTM$_H$ a close second. The three schemes also all have good performance, with BugTM$_S$ being the best and BugTM$_{HS}$ the second best. BugTM$_{HS}$ provides the best combination of recovery capability and performance.

9.1 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$_{HS}$ can recover all of these 20).

Table 3 shows the result of 7 benchmarks where different tools show different recovery capability. For the other 13 benchmarks, ConAir and BugTM can all help recover from all of them.

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 Fig. 4.

BugTM$_S$ has better recovery capability than ConAir. Its *deferred write* technique helps it to successfully recover from the two RAR violation failures in the table. The *undo log* technique of BugTM$_S$ allows it to partly recover from two out of three RAW benchmarks. BugTM$_S$ does not apply *undo log* to Moz-JS142651 because the bug involves complicated control flows. Moz-JS18025 is demonstrated in Fig. 2. As discussed earlier, BugTM$_S$ can help recover from the failure shown in the figure, but cannot prevent subsequent failures caused by the `NULL` value of `s->table`. Apache21287 can be recovered by BugTM$_S$ with about 50 percent probability, depending on which bug-related thread fails first. Finally, BugTM$_S$ fundamentally cannot handle WAR violations, as discussed at the end of Section 5.

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 [24], [25].

BugTM$_{\rm HS}$ combines the strengths of BugTM$_H$ and ConAir, and hence can successfully recover from all 7 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 [7], [27]. 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 [24], [25] could help recover these failures.

Hardened Failure Sites. BugTM can help recover from failures caused by concurrency bugs because it statically identifies *potential failure sites* and inserts rollback-reexecution code surrounding them.

Table 4 shows the number of failure sites that are hardened by BugTM$_{\rm HS}$ in each benchmark software. Naturally, BugTM$_{\rm HS}$ identifies and hardens the fewest failure sites in the smallest programs (Bank and HawkNL) and the most failure sites in the largest programs (MySQL and Apache). Benchmarks from the same software have slightly different numbers of failures sites in Table 4, because they come from different versions of the software. We do full instrumentation for each version.

There are four types of potential failure sites: every assertion checking is a potential site for assertion-violation failure; every output function call is a potential site for wrong-output failure; every deference of a heap or global pointer is a potential site for segmentation-fault failures; every lock acquisition that is enclosed by another lock acquisition is considered as a potential deadlock site. Among these four, potential segmentation-fault sites are the most common, and potential deadlock sites are the least common.

BugTM$_H$ and BugTM$_S$ have hardened similar, just slightly fewer, failure sites as BugTM$_{\rm HS}$, as some failure sites hardened by BugTM$_{\rm HS}$ are identified as un-recoverable for BugTM$_H$ and BugTM$_S$, and hence not hardened by them.

TABLE 4
Static Failure Sites Hardened by BugTM$_{\rm HS}$

	Assertion Violation	Wrong Output	Seg. Fault	Deadlock	Total
MySQL2011	151	3,128	15,498	21	18,798
MySQL3596	170	3,297	15,791	19	19,277
MySQL38883	170	3,276	15,820	19	19,285
Apache21287	5	503	16,834	89	17,431
Moz-JS18025	35	34	1,802	10	1,881
Moz-JS142651	1	31	1,812	13	1,857
Bank	1	1	1	0	3
Moz-ex52111	1	1	23	0	25
Moz-ex209188	1	2	34	0	37
MySQL791	129	2,421	17,890	43	20,483
MySQL16582	134	2,943	14,592	20	17,689
Click	2,134	32	2,234	0	4,400
FFT	5	32	14	0	51
HTTrack	657	504	3,146	0	4,307
Moz-xpcom	1	117	6,791	0	6,909
Transmission	430	190	2,151	0	2,771
zsnes	1	50	331	0	382
HawkNL	0	0	5	2	7
Moz-JS79054	0	5	134	6	145
SQLite1672	0	25	47	1	73

9.2 Performance

Table 5 shows the regular-run overheads of applying BugTM schemes to 20 benchmarks, all the benchmarks that are recoverable by BugTM$_{\rm HS}$.

Overall, BugTM$_S$ has the best performance, incurring less than 1 percent overhead for *all* benchmarks at run time, almost a free lunch for production failure recovery. Considering that each reexecution point only takes a few nanoseconds to execute (a `setjmp`), the low overhead of BugTM$_S$ is understandable.

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

Fortunately, BugTM$_{\rm HS}$ wins most of the lost performance back, incurring 1.4 percent overhead on average and less than 3 percent for *all but 3* benchmarks. In the worst cases, it incurs 4.2 and 5.3 percent overhead for two benchmarks in Mozilla JavaScript Engine (JSE), a browser component with little I/O. If we apply BugTM$_{\rm HS}$ to the whole browser, the overhead would be much smaller, as JSE never takes >20 percent of the whole page-loading time based on our profiling and previous work [28].

Comparing BugTM$_{\rm HS}$ with BugTM$_H$, BugTM$_{\rm HS}$ is faster mainly because it has greatly reduced the number of hardware 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$_{\rm HS}$ 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 percent.

Tx abort rate is less than 1 percent for all benchmarks, with more than 95 percent of all aborts being unknown aborts (timer interrupts, etc.). As Section 9.4 will show, abort rates and overhead are much worse in alternative designs.

We do not show the execution frequencies of failure sites. In our performance experiments behind Table 5, failures are never triggered. Of course, code regions surrounding failure

TABLE 5
Overhead During Regular Execution and Detailed Performance Comparison

	Run-time Overhead				#setjmp		#StartTx		#StartTx per 10 μs		Abort%	
	ConAir	BugTM$_S$	BugTM$_H$	BugTM$_{HS}$	BugTM$_S$	BugTM$_{HS}$	BugTM$_H$	BugTM$_{HS}$	BugTM$_H$	BugTM$_{HS}$	BugTM$_H$	BugTM$_{HS}$
MySQL2011	0.05%	0.04%	0.13%	0.08%	642974	643425	2746031	778024	2.3	0.7	0.01	0.01
MySQL3596	0.40%	0.43%	3.10%	1.12%	144119	144212	110476	39913	3.9	1.4	0.12	0.20
MySQL38883	0.40%	0.41%	3.08%	1.11%	144109	144119	110471	39904	3.9	1.4	0.11	0.19
Apache21287	0.55%	0.73%	3.77%	3.00%	39918	40023	72093	45520	22.8	14.5	0.08	0.11
Moz-JS18025	0.57%	0.86%	9.03%	2.62%	3987	3992	6850	1159	16.3	2.8	0.29	0.04
Moz-JS142651	0.76%	0.86%	11.9%	5.30%	2269	2145	9666	4007	30.4	12.6	0.33	0.17
Bank	0.15%	0.23%	2.18%	2.95%	6	6	5	5	0.1	0.1	0.0	0.0
Moz-ex52111	0.47%	0.65%	0.53%	0.41%	4	4	3	0	0.0	0.0	0.0	0.0
Moz-ex209188	0.12%	0.12%	0.58%	0.77%	2	2	1	1	0.0	0.0	0.0	0.0
MySQL791	0.35%	0.84%	1.98%	0.24%	48933	48998	4948	602	2.5	0.4	0.35	0.01
MySQL16582	0.15%	0.33%	3.03%	0.99%	269230	269543	153532	31222	3.8	0.8	0.03	0.06
Click	0.57%	0.80%	8.11%	3.60%	4893	4681	5142	2123	18.7	8.1	0.96	0.12
FFT	0.05%	0.05%	0.03%	0.14%	23	23	25	19	0.0	0.0	0.0	0.0
HTTrack	0.15%	0.16%	0.64%	0.04%	9004	9212	15649	1572	0.1	0.0	0.83	0.11
Moz-xpcom	0.38%	0.40%	0.45%	0.03%	313	324	1933	154	0.0	0.0	0.31	0.51
Transmission	0.11%	0.20%	0.22%	0.07%	1088	1093	2123	919	0.1	0.0	0.56	0.40
zsnes	0.05%	0.11%	0.03%	0.44%	10684	10462	11737	372	0.5	0.0	0.13	0.23
HawkNL	0.09%	0.08%	0.00%	0.15%	10	10	19	16	0.0	0.0	0.0	0.07
Moz-JS79054	0.84%	0.99%	11.7%	4.20%	340	338	1325	360	9.4	2.6	0.23	0.44
SQLite1672	0.05%	0.01%	0.98%	0.50%	6	6	3	3	0.1	0.1	0.0	0.06
Avg.	0.31%	0.42%	3.08%	1.39%	-	-	-	-	-	-	-	-

red font denotes >3% overhead; #: count of dynamic instances; Abort%: percentage of aborted dynamic Tx*s.*

sites have been executed during our experiments. Their execution frequencies are reflected by the numbers of `setjmp` and `StartTx` in Table 5.

Recovery Time & Comparison with Whole-program Restart. As shown in Table 6, a successful BugTM failure recovery takes little time.

The recovery of atomicity violations, except for WAW atomicity violations, and deadlocks mostly takes less than 100 μ-seconds. For these bugs, one run of retry is sufficient to avoid unexpected interleaving, because the failing thread does not need to wait for any other threads.

The recovery of order violations and WAW atomicity violations takes slightly longer time, as it highly depends on how much `sleep` is inserted to trigger the failure, as discussed in Section 7.

The reexecution regions are always small, ranging from 5-224 instructions at byte-code level.

BugTM recovery is much faster than a system restart, which could take a few minutes or even more for complicated systems, as shown in the table. BugTM recovery 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.

Table 6 only shows the recovery details of BugTM$_{HS}$. The details of BugTM$_H$ and BugTM$_S$ are similar—they took negligible amount of time to recover from most atomicity violations and deadlocks, and slightly longer time to recover from order violations and WAW atomicity violations.

Understanding BugTM$_H$ Overhead. The overhead of BugTM$_H$ differs among benchmarks, ranging from 0.00 to 11.9 percent. As TM researchers found before, performance in TM systems is often complicated [29], [30]. An indicating metrics for our benchmarks is the frequency of dynamic `StartTx`. As shown in the #`StartTx` per 10 μs column of Table 5, BugTM$_H$ executes more than 1 `StartTx` per 10 micro second on average for 10 benchmarks, and incurs more than 1 percent overhead for 9 of them.

TABLE 6
Failure Recovery Details by BugTM$_{HS}$

	RootCause	Recovery (μ-seconds)	# Retries	Restart (μ-seconds)
MySQL2011	AV$_{RAR}$	10	1	5,069,944
MySQL3596	AV$_{RAR}$	21	1	5,982,713
MySQL38883	AV$_{RAR}$	21	1	5,982,720
Apache21287	AV$_{RAW}$	98	1	4,824,363
Moz-JS18025	AV$_{RAW}$	42	1	5,508
Moz-JS142651	AV$_{RAW}$	74	1	1,143
Bank	AV$_{WAR}$	5	1	139
Moz-ex52111	AV$_{WAW}$	183	1245	149
Moz-ex209188	AV$_{WAW}$	92	741	107
MySQL791	AV$_{WAW}$	10,732	11,234	66,724
MySQL16582	AV$_{WAW}$	24,321	23,108	7,424,224
Click	OV	1,211	7,662	900,452
FFT	OV	11,719	9,725	131,494
HTTrack	OV	4,625	18,244	11,776
Moz-xpcom	OV	37,388	124,732	217,041
Transmission	OV	22,445	1,234	553,298
zsnes	Deadlock	109	1	354,640
HawkNL	Deadlock	78	1	14,493
Moz-JS79054	Deadlock	35	1	472
SQLite1672	Deadlock	89	1	1,570

The experiments are conducted with small amount of noises inserted to help trigger the concurrency-bug failures.

9.3 Diagnosis

Table 7 shows the different diagnosis capability of BugTM.

BugTM$_S$ provides useful diagnosis information for all of the 18 benchmarks that it can help recover from. For 10 out of these benchmarks whose root causes are order violations

TABLE 7
Diagnosis Capability Comparison

	RootCause	BugTM$_S$	BugTM$_H$	BugTM$_{HS}$
MySQL2011	AV$_{RAR}$	✓	-	-
MySQL3596	AV$_{RAR}$	✓	-	✓
MySQL38883	AV$_{RAR}$	✓	-	-
Apache21287	AV$_{RAW}$	✓	-	-
Moz-JS18025	AV$_{RAW}$	✓	-	-
Moz-JS142651	AV$_{RAW}$	NA	-	-
Bank	AV$_{WAR}$	NA	-	-
Moz-ex52111	AV$_{WAW}$	✓*	✓*	✓*
Moz-ex209188	AV$_{WAW}$	✓*	✓*	✓*
MySQL791	AV$_{WAW}$	✓*	✓*	✓*
MySQL16582	AV$_{WAW}$	✓*	✓*	✓*
Click	OV	✓*	✓*	✓*
FFT	OV	✓*	✓*	✓*
HTTrack	OV	✓*	✓*	✓*
Moz-xpcom	OV	✓*	✓*	✓*
Transmission	OV	✓*	NA	✓*
zsnes	OV	✓*	✓*	✓*
HawkNL	Deadlock	✓	✓	✓
Moz-JS79054	Deadlock	✓	✓	✓
SQLite1672	Deadlock	✓	✓	✓
Total		8(18)	3(12)	4(14)

NA: can't help recover from them
**: report two possible root causes, order violation and WAW violation.*
-: no affirmative diagnosis

or WAW atomicity violations, BugTM$_S$ reports that the root cause could be either one of these two. For the other 8 benchmarks, BugTM$_S$ accurately pin-points the exact root cause.

There are many benchmarks in Table 7 that can be recovered by BugTM$_H$ and BugTM$_{HS}$, but cannot be diagnosed by them. The reason is that these bugs' manifestation leads to HTM data conflict aborts, while BugTM$_H$ and BugTM$_{HS}$ cannot decide whether such aborts are caused by software bugs or benign races.

BugTM also conducts memory-access logging during failure recovery attempts. 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.

9.4 Alternative Designs of BugTM

Table 8 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 versus Intra-procedural. BugTM$_H$ uses the inter-procedural algorithm discussed in Section 3.5. This design adds 0.00–4.3 percent overhead to its intra-procedural alternative, as shown in Table 8. In exchange, there are 4 benchmarks in Table 8 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,

TABLE 8
BugTM$_H$ versus Alternative Designs

	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% ✓

%: the overhead over baseline execution w/o recovery scheme applied; ✓: failure recovered; ×: failure not recovered.

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 percent overhead for three MySQL benchmarks shown in Table 8. Such design also causes BugTM$_{HS}$ to incur more than 20 percent 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 8. 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 8. 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 percent.

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 percent. The impact to BugTM$_{HS}$ would also be huge accordingly.

9.5 Discussion

BugTM has explored three designs that all have better failure-recovery capability than the state of the art technique ConAir, at the cost of slightly worse performance than ConAir. These three designs have different trade-off combinations between performance and failure-recovery capability. Among them, BugTM$_S$ has the best performance, as well as the best failure diagnosis capability, but has the worst recovery capability. BugTM$_H$ has better recovery capability than BugTM$_S$, but the worst performance. BugTM$_{HS}$ has the best recovery capability, and also good performance that is much better than BugTM$_H$ but slightly worse than BugTM$_S$. On commodity machines that support HTM, BugTM$_{HS}$ is likely the best BugTM design. For the multi-threaded software systems which are performance-sensitive or I/O intensive, like browser, BugTM$_S$ and BugTM$_{HS}$ are recommended instead of BugTM$_H$ considering BugTM's higher overhead and HTM constraints. For other multi-threaded software systems which emphasize availability, e.g banking/business systems, they may prefer BugTM$_H$ and BugTM$_{HS}$ for better recovery capabilities, which means fewer failure occurrences during production runs. Overall, all hardened programs of these three designs can perform as the intermediate patches before the final patches generated, which improves software availability and could facilitate the debugging and patching process.

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. Even if not successful, the recovery attempts only delays the failure manifestation by a negligible amount of time (less than 40 micro-seconds in our experiments).

Several practices can help further improve the benefit of BugTM. First, as discussed in Section 9.1, 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.

10 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 [13], [31] to prevent the same bug from manifesting again, or relies on extensive off-line training [32], [33] 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 [33] 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.

Concurrency Bug Detection. Many automated detection tools have been proposed for a variety of concurrency bugs, including data races [34], [35], [36], [37], [38], [39], [40], [41], atomicity violations [42], [43], order violations [14], [44], [45], and deadlocks [46]. These tools aim to discover bugs during in-house testing and are not a good fit for production-run failure recover—they often incur large overhead (e.g., 10X slowdowns) and cannot provide the desired bug/failure coverage.

Automated Concurrency-bug Fixing. Static analysis and code transformation techniques have been proposed to automatically generate patches for concurrency bugs [12], [26], [46], [47]. They work at off-line and rely on accurate bug-detection results. A recent work [48] 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 [7], [40] and debugging [49], [50], [51], [52] 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 [53], [54]. Using it to help recover from concurrency-bug failures was recently pioneered by ConAir [9]. 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 [55], [56], [57]. 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 percent overhead [55], [57]) or lots of hardware changes to existing HTM [56], 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 [55] or only apply for single-threaded software [56], [57], which is completely different from BugTM.

Others. Diagnosing production-run failure is challenging. Sampling has been proposed to lower its run-time overhead [58], [59], [60]. Triage [52] re-executes software from previous checkpoints after failure, and applies dynamic bug detection during re-execution to diagnose production-run failures. Different from BugTM$_{HS}$, Triage requires changes to OS to support full-application checkpoint-and-replay, and relies on bug-detection tools for diagnosis. Furthermore, Triage like its predecessor Rx [7] focuses on memory bugs. BugTM$_{HS}$ focuses on concurrency bugs, and leverages software's reaction to failure-recovery attempts to diagnose failures.

Lots of research was done on HTM and STM [17], [61], [62], [63], [64], [65], [66], [67], [68]. Recent work explored using HTM to speed up distributed transaction systems [69], race detection [70], [71], etc. Previous empirical studies have examined the experience of using Txs., instead of locks, in developing parallel programs [72], [73]. They all look at different ways of using TM systems from BugTM.

11 Future Work

Recently, more processors that support TSX-NI have been released, like Intel's Xeon family processors. These processors have more cores and threads than i7-5775C processors that we used in our experiments. It will be interesting to deploy BugTM on these new processors and see the performance differences.

There is still a lot of design space to explore for concurrency bug failure recovery. For example, we could use more complicated checkpointing techniques to help BugTM$_S$ recover from more concurrency bugs, but it may lead to worse performance.

Future work can also explore mixing and switching among different recovery mechanisms, like ConAir and BugTM$_H$, at run time, applying different strategies under different dynamic contexts.

12 Conclusions

Concurrency bugs severely affect system availability. This paper presents three TM-inspired techniques to help automatically recover concurrency-bug failures during production runs. BugTM$_H$ automatically inserts HTM APIs into software. It is capable of recovering failures caused by all major types of concurrency bugs and incurs 3.08 percent overhead on average in our evaluation. BugTM$_S$ uses STM inspired techniques to enhance the recovery capability of state-of-the-art ConAir. Although it cannot recover as many failures as BugTM$_H$, it incurs less than 1 percent overhead and can provide useful diagnostic information. BugTM$_{\mathrm{HS}}$ combines ConAir and BugTM$_H$, achieving a good combination of recovery capability and performance (1.39 percent overhead). Overall, BugTM improves the state of the art of failure recovery, and presents novel ways of using TM techniques.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful feedback and comments to bring the paper to its final form. And we would also thank Wei Zhang and Linhai Song for their help in our experiments with ConAir. This project is funded by NSF (grants CNS-1563956, IIS-1546543, CNS-1514256, CNS-1764039, CCF-1514189, CCF-1439091) and CERES Center for Unstoppable Computing.

References

[1] Y. Chen, S. Wang, S. Lu, and K. Sankaralingam, "Applying hardware transactional memory for concurrency-bug failure recovery in production runs," in *Proc. USENIX Annu. Tech. Conf.*, 2018.

[2] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes–A comprehensive study of real world concurrency bug characteristics," in *Proc. 13th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2008, pp. 329–339.

[3] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Comput.*, vol. 26, no. 7, pp. 18–41, 1993.

[4] PCWorld, "Nasdaq's Facebook glitch came from race conditions," [Online]. Available: http://www.pcworld.com/businesscenter/article/255911/nasdaqs_facebook_g litch_came_from_race_conditions.html

[5] SecurityFocus, "Software bug contributed to blackout," [Online]. Available: http://www.securityfocus.com/news/8016

[6] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram, "How do fixes become bugs?" in *Proc. 13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 26–36.

[7] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating bugs as allergies—a safe method to survive software failure," in *Proc. 20th ACM Symp. Operating Syst. Principles*, 2005, pp. 235–248.

[8] Y. Saito, "Jockey: A user-space library for record-replay debugging," in *Proc. 6th Int. Symp. Autom. Anal.-Driven Debugging*, 2005, pp. 69–76.

[9] W. Zhang, M. de Kruijf, A. Li, S. Lu, and K. Sankaralingam, "ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution," in *Proc. 18th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2013, pp. 113–126.

[10] H. Volos, A. J. Tack, M. M. Swift, and S. Lu, "Applying transactional memory to concurrency bugs," in *Proc. 17th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2012, pp. 211–222.

[11] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: Why is it only a research toy?" *ACM Queue*, vol. 6, pp. 40–46, 2008.

[12] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2011, pp. 389–400.

[13] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea, "Deadlock immunity: Enabling systems to defend against deadlocks," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, 295–308. [Online]. Available: https://code.google.com/archive/p/dimmunix/

[14] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng, "Do I use the wrong definition? DeFUse: Definition-use invariants for detecting concurrency and sequential bugs," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2010, pp. 160–174.

[15] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps, "ConSeq: Detecting concurrency bugs through sequential errors," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2011, pp. 251–264.

[16] W. Zhang, C. Sun, J. Lim, S. Lu, and T. Reps, "ConMem: Detecting crash-triggering concurrency bugs through an effect-oriented approach," *ACM Trans. Softw. Eng. Methodology*, vol. 22, 2012, Art. no. 10.

[17] T. Harris, J. R. Larus, and R. Rajwar, *Transactional Memory*, 2nd ed. San Mateo, CA, USA: Morgan & Claypool, 2010.

[18] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proc. 20th Annu. Int. Symp. Comput. Archit.*, 1993, pp. 289–300.

[19] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, et al., "The IBM blue gene/Q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, Mar./Apr. 2012.

[20] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski, "Early experience with a commercial hardware transactional memory implementation," in *Proc. 14th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2009, pp. 157–168.

[21] Intel 64 and ia-32 architectures optimization reference manual, [Online]. Available: http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf, Accessed on: Jul. 30, 2016

[22] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation Optimization: Feedback-Directed Runtime Optimization*, 2004, Art. no. 75.

[23] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, "Atom-aid: Detecting and surviving atomicity violations," in *Proc. 35th Annu. Int. Symp. Comput. Archit.*, 2008, pp. 277–288.

[24] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel, "TxLinux: Using and managing hardware transactional memory in an operating system," in *Proc. 21st ACM Symp. Operating Syst. Principles*, 2007, pp. 87–102.

[25] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, and A. Welc, "xCalls: Safe I/O in memory transactions," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, 2009, pp. 247–260.

[26] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 221–236.

[27] V. B. Lvin, G. Novark, and E. D. Berger, "Archipelago: Trading address space for reliability and security," in *Proc. 13th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2008, pp. 115–124.

[28] J. Nejati and A. Balasubramanian, "An in-depth study of mobile browser performance," in *Proc. 25th Int. Conf. World Wide Web*, 2016, pp. 1305–1315. [Online]. Available: https://doi.org/10.1145/2872427.2883014

[29] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, 2007, pp. 81–91.

[30] D. E. Porter and E. Witchel, "Understanding transactional memory performance," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2010, pp. 97–108.

[31] B. Lucia and L. Ceze, "Cooperative empirical failure avoidance for multithreaded programs," in *Proc. 18th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2013, pp. 39–50.

[32] M. Zhang, Y. Wu, S. Lu, S. Qi, J. Ren, and W. Zheng, "AI: A lightweight system for tolerating concurrency bugs," in *Proc. 22nd Eur. Conf. Found. Softw. Eng.*, 2014, pp. 330–340.

[33] J. Yu and S. Narayanasamy, "Tolerating concurrency bugs using transactions as lifeguards," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2010, pp. 263–274.

[34] M. D. Bond, K. E. Coons, and K. S. McKinley, "PACER: Proportional detection of data races," in *Proc. 31st ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2010, pp. 255–268.

[35] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *Proc. 30th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2009, pp. 121–133.

[36] C. Hsiao, C. Pereira, J. Yu, G. Pokam, S. Narayanasamy, P. M. Chen, Z. Kong, and J. Flinn, "Race detection for event-driven mobile applications," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 326–336.

[37] B. Kasikci, C. Zamfir, and G. Candea, "RaceMob: Crowdsourced data race detection," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 406–422.

[38] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: Effective sampling for lightweight data-race detection," in *Proc. 30th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2009, pp. 134–143.

[39] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, pp. 391–411, 1997. [Online]. Available: citeseer.nj.nec.com/savage97eraser.html

[40] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy, "Detecting and surviving data races using complementary schedules," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 369–384.

[41] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: Efficient detection of data race conditions via adaptive tracking," in *Proc. 20th ACM Symp. Operating Syst. Principles*, 2005, pp. 221–234.

[42] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting atomicity violations via access interleaving invariants," in *Proc. 12th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2006, pp. 37–48.

[43] B. Lucia, L. Ceze, and K. Strauss, "ColorSafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 222–233.

[44] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin, "2ndStrike: Toward manifesting hidden concurrency typestate bugs," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2011, pp. 239–250.

[45] W. Zhang, C. Sun, and S. Lu, "ConMem: Detecting severe concurrency bugs through an effect-oriented approach," in *Proc. 15th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2010, pp. 179–192.

[46] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlk, "Gadara: Dynamic deadlock avoidance for mult-threaded programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 281–294.

[47] P. Liu, O. Tripp, and C. Zhang, "Grail: Context-aware fixing of concurrency bugs," in *Proc. 22nd Eur. Conf. Found. Softw. Eng.*, 2014, pp. 318–329.

[48] J. Huang and C. Zhang, "Execution privatization for scheduler-oblivious concurrent programs," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2012, pp. 737–752.

[49] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic systems," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 525–540.

[50] S. King, G. Dunlap, and P. Chen, "Debugging operating systems with time-traveling virtual machines," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2005, pp. 1–1.

[51] G. Pokam, K. Danne, C. Pereira, R. Kassa, T. Kranich, S. Hu, J. E. Gottschlich, N. Honarmand, N. Dautenhahn, S. T. King, and J. Torrellas, "QuickRec: Prototyping an intel architecture extension for record and replay of multithreaded programs," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 643–654.

[52] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou, "Triage: Diagnosing production run failures at the user's site," in *Proc. 21st ACM Symp. Operating Syst. Principles*, 2007, pp. 131–144.

[53] M. de Kruijf and K. Sankaralingam, "Idempotent processor architecture," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2011, pp. 140–151.

[54] S. Feng, S. Gupta, A. Ansari, S. Mahlke, and D. August, "Encore: Low-cost, fine-grained transient fault recovery," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2011, pp. 398–409.

[55] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, "HAFT: Hardware-assisted fault tolerance," in *Proc. 11th ACM Eur. Conf. Comput. Syst.*, 2016, Art. no. 25.

[56] J. Li, Q. Tan, and L. Tan, "EnHTM: Exploiting hardware transaction memory for achieving low-cost fault tolerance," in *Proc. 4th Int. Conf. Digital Manuf. Autom.*, 2013, 550–554.

[57] G. Yalcin, O. S. Unsal, A. Cristal, I. Hur, and M. Valero, "SymptomTM: Symptom-based error detection and recovery using hardware transactional memory," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2011, pp. 199–200.

[58] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu, "Production-run software failure diagnosis via hardware performance counters," in *Proc. 18th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2013, pp. 101–112.

[59] G. Jin, A. Thakur, B. Liblit, and S. Lu, "Instrumentation and sampling strategies for cooperative concurrency bug isolation," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2010, pp. 241–255.

[60] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2003, pp. 141–154.

[61] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proc. 11th Int. Symp. High-Perform. Comput. Archit.*, 2005, pp. 316–327.

[62] T. Bai, X. Shen, C. Zhang, W. N. Scherer, C. Ding, and M. L. Scott, "A key-based adaptive transactional memory executor," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2007, pp. 1–8.

[63] D. R. Chakrabarti, P. Banerjee, H.-J. Boehm, P. G. Joisha, and R. S. Schreiber, "The runtime abort graph and its application to software transactional memory optimization," in *Proc. Int. Symp. Code Generation Optimization: Feedback-Directed Runtime Optimization*, 2011, pp. 42–53.

[64] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 2004, Art. no. 102.

[65] T. Harris and K. Fraser, "Language support for lightweight transactions," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2003, pp. 388–402.

[66] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based transactional memory," in *Proc. 12th Int. Symp. High-Perform. Comput. Archit.*, 2006, pp. 254–265.

[67] R. Rajwar and J. R. Goodman, "Transactional lock-free execution," in *Proc. 10th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2002, pp. 5–17.

[68] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott, "An integrated hardware-software approach to flexible transactional memory," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, 2007, pp. 104–115.

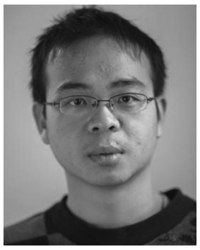
[69] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," in *Proc. 25th ACM Symp. Operating Syst. Principles*, 2015, pp. 87–104.

[70] S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Rotteler, "Using hardware transactional memory for data race detection," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2009, pp. 1–11.

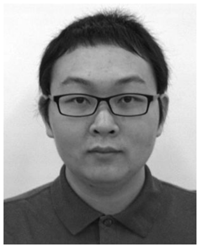
[71] T. Zhang, D. Lee, and C. Jung, "TxRace: Efficient data race detection using commodity hardware transactional memory," in *Proc. 21st Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2016, pp. 159–173.

[72] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" in *Proc. 15th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2010, pp. 47–56.

[73] J. Zhang, W. Chen, X. Tian, and W. Zheng, "Exploring the emerging applications for transactional memory," in *Proc. 9th Int. Conf. Parallel Distrib. Comput. Appl. Technol.*, 2008, pp. 474–480.



Yuxi Chen received the BE degree from Wuhan University, China, in 2013, and the MS degree from the University of Chicago, in 2016. He is currently working toward the PhD degree in the Department of Computer Science, the University of Chicago. His research interests include system reliability, program analysis.



Shu Wang received the BE degree from the Harbin Institute of Technology, China, in 2013, and the MS degree from the University of Wisconsin-Madison, in 2015. He is currently working toward the PhD degree in the Department of Computer Science, the University of Chicago. His research interests include distributed system, system reliability, and networking.

Shan Lu received the PhD degree at the University of Illinois, Urbana-Champaign, in 2008. She is an associate professor with the Department of Computer Science, University of Chicago. Her research focuses on software reliability and efficiency. Her co-authored papers have won awards at OSDI, FAST, PLDI, FSE, and ICSE conferences. She currently serves as the vice chair of ACM-SIGOPS and the associate editor for IEEE Computer Architecture Letters. She is a member of the IEEE.

Karu Sankaralingam received the PhD degree from the University of Texas at Austin, in December 2006. He is a professor with the Computer Sciences Department, University of Wisconsin-Madison. His research interests include microarchitecture, architecture, and software issues for massively parallel computation systems. His co-authored papers have won awards at PLDI, IEEE Micro. He currently serves as the associate editor for IEEE Computer Architecture Letters. He is a member of the IEEE.