# TABLE OF CONTENTS

LIST OF FIGURES ................................................................. vii
LIST OF TABLES ................................................................. viii
ACKNOWLEDGMENTS ............................................................. ix
ABSTRACT ................................................................. x

1 INTRODUCTION ........................................................... 1
  1.1 Motivation .......................................................... 1
  1.2 Contribution ...................................................... 3

2 BACKGROUND ............................................................. 6

3 BUGTM ................................................................. 8
  3.1 High-Level Design ................................................ 8
    3.1.1 Strawman approaches ...................................... 8
    3.1.2 Our approach ............................................... 8
  3.2 BugTM design about AbortTx ................................... 10
  3.3 BugTM design about StartTx and CommitTx ..................... 11
    3.3.1 Challenges ................................................... 11
    3.3.2 Where to XBEGIN and XEND ............................... 12
    3.3.3 How to StartTx and CommitTx ................................ 12
  3.4 BugTM design for fallback and retry ........................... 13
    3.4.1 Challenges ................................................... 13
    3.4.2 Solutions .................................................... 13
  3.5 Inter-procedural BugTM and Others ............................ 15
    3.5.1 Inter-procedural BugTM ....................................... 15
    3.5.2 Optimizations ............................................... 15

4 BUGTMS ................................................................. 16
  4.1 Exploring the Design Space ..................................... 16
  4.2 Background: ConAir .............................................. 17
  4.3 Deferred Writes for Failure-Unrelated Killing Writes ........ 17
    4.3.1 Feasibility checking ....................................... 18
    4.3.2 Moving the \(w_{\text{kill}}\) ..................................... 19
  4.4 Undo Log for Failure-Related Killing Writes ..................... 20
    4.4.1 Basic Algorithm ............................................. 20
    4.4.2 Final Algorithm ............................................ 21

5 FAILURE DIAGNOSIS .................................................. 23

6 METHODOLOGY ........................................................... 25
7 EXPERIMENTAL RESULTS ...................................................... 27
  7.1 Failure recovery capability ........................................... 27
  7.2 Performance ............................................................. 30
  7.3 Diagnosis ................................................................. 33
  7.4 Alternative designs of BugTM ....................................... 33

8 RELATED WORK ............................................................ 36

9 CONCLUSIONS .............................................................. 38
LIST OF FIGURES

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

3.1 An real-world OV bug (simplified from Transmission) .......................... 9

3.2 BugTM wrapped function (my_xend) for CommitTx ............................... 13

3.3 BugTM wrapped function (my_xbegin) for StartTx ............................... 14

3.4 BugTM code transformation for assert(e) ............................................ 14

4.1 BugTM's deferred write transformation, denoted by ‘+’ and ‘-’, makes a ConAir-unrecoverable bug recoverable ................................. 18

4.2 Moving a killing write (flag is initialized FALSE) ................................. 19

4.3 Memory-checkpoint example .............................................................. 20

5.1 Root-cause diagnosis based on failure recovery .................................... 23
### LIST OF TABLES

3.1 Common types of concurrency bugs and how BugTM, BugTM$_S$, and ConAir attempt to recover from them. (R/W: read/write to a shared variable; thick vertical line: the execution of one thread; dashed arrowed line: the re-execution region of BugTM; thin arrowed line: the re-execution region of BugTM$_S$ and ConAir; explosion symbol: a failure; -: cannot recover; ✓: sometimes can recover; ✓✓: mostly can recover.)

7.1 Overall results and comparison (CA: ConAir; *: failures partly recovered; red font denotes > 4% overhead; Moz-JS: Mozilla JavaScript Engine; Moz-ex: benchmarks extracted from Mozilla; Moz-xpcom: Mozilla COM model.)

7.2 Reasons for unrecoverable benchmarks (we use numbers to index multiple reasons)

7.3 # of static and dynamic setjmp and StartTx in BugTM$_S$ and BugTM (Dyn.Freq.: # of dynamic StartTx per 100 $\mu$-second; Abort%: percentage of aborted dynamic Txs.)

7.4 Comparing BugTM with alternative designs (The %s are the overhead over baseline execution w/o any recovery scheme applied; ✓: failure recovered; X: failure not recovered.)
ACKNOWLEDGMENTS

I am sincerely grateful to my advisor Prof. Shan Lu. Thanks for her patience and detailed guidance since 2014. I am so lucky to have such nice advisor. Without her, I would never make satisfactory progress in my research career. And it’s my honor to have Prof. Haryadi Gunawi, Prof. Andrew Chien in my master committee.

Also, I would like to thank my partner Shu Wang, and Linhai Song, Guoliang Jin, Wei Zhang, Haopeng Liu. They patiently answered every question I asked even when they were busy. And they gave me lots of useful suggestions. I learn much from them, enthusiasm, perseverance, hardworking, preciseness. I enjoy working with these guys.

In the last, I would also like to thank my parents. They encouraged me to keep focused throughout this project.
ABSTRACT

Concurrency bugs severely threaten software reliability in production runs. They are difficult to expose and eliminate during in-house testing, lead to severe production run failures, and are time-consuming to diagnose and fix correctly. Techniques that help recover software from concurrency-bug failures during production runs are highly desired.

This paper proposes using transactional memory techniques to help production-run concurrency-bug recovery. BugTM uses existing hardware transactional memory support (Intel TSX). It can recover from failures caused by all major types of concurrency bugs, with about 4% overhead on average in our thorough evaluation. For systems that lack hardware transaction support, we develop a software transactional memory system called BugTM$_S$ by targeting TM principles for concurrency-bug failure recovery. It has slightly worse recovery capability than BugTM, but out-performs the state-of-art techniques in overhead, coverage, and diagnosis capability.
CHAPTER 1
INTRODUCTION

1.1 Motivation

Concurrency bugs are caused by untimely accesses to shared variables. They are difficult to eliminate during in-house testing. They widely exist in production-run software systems [32], have caused disasters during production runs [29, 40, 49], and are extremely difficult to fix correctly after being discovered [58]. Techniques that can handle production-run failures caused by concurrency bugs are highly desired.

Several techniques have been proposed to either proactively prevent the manifestation of concurrency bugs or reactively recover software from triggered concurrency-bug failures.

The prevention approach works by perturbing the timing of the program execution, hoping that failure-triggering interleavings would not happen. This approach either relies on prior knowledge about a concurrency bug and its failure [25, 34] to prevent the same bug from manifesting again, or relies on extensive off-line training [61] to guide the production-run execution towards likely failure-free timing. It is not suitable for avoiding production-run failures caused by previously unknown concurrency bugs. Furthermore, the perturbation may cause unacceptable slowdowns [61].

The recovery approach works through record and re-execution, hoping that failure-triggering interleavings would not happen again during re-execution. This approach requires frequent checkpoints to achieve fast and correct failure recovery. Past work of this approach faces the design tradeoff of run-time overhead versus recovery capability. On one hand, full-blown checkpoint and re-execution can help recover almost all concurrency-bug failures. However, it incurs too large overhead to be deployed in production runs without changes to operating systems or hardware [47]. On the other hand, feather-weight checkpoint and re-execution schemes sacrifice re-execution capability to achieve low run-time overhead. For
example, a recently proposed tool ConAir [63] incurs less than 1% run-time overhead, by re-executing only one thread and requiring the re-execution region to be idempotent. It is suitable for production-run deployment, but is limited in terms of failure-recovery capability.

We elaborate on a motivating example from real-world applications to highlight the limitations of the state-of-art techniques. Figure 1.1 illustrates a read-after-write (RAW) atomicity violation from Mozilla: the write and read of $s \rightarrow \text{table}$ in Thread-1 are expected to execute atomically, but are unfortunately interleaved by the NULL assignment from Thread-2. At the first glance, re-executing the write and read to $s \rightarrow \text{table}$ in Thread-1 would recover the failure. However, correct and efficient re-execution is non-trivial. If we re-execute both threads, there is a good chance that the failure would happen again. More importantly, efficient and consistent re-execution of multiple threads is difficult to achieve without OS/hardware support. If we re-execute only Thread-1, like what ConAir does, the re-execution correctness cannot be guaranteed: another thread $t'$ could have read the old value of $s \rightarrow \text{table}$, saved it to a local pointer, the re-execution then gave $s \rightarrow \text{table}$ a new value, causing inconsistency between $t'$ and $t$ and deviation from the original program semantics. Because of this correctness concern, ConAir does not support re-executing any writes to shared variables, and hence cannot recover many concurrency-bug failures.

```c
1 //Thread-1
2 s \rightarrow \text{table} = \text{newTable}(...);
3
4 if(!s \rightarrow \text{table}){
5     //fatal-error message; software fails
6 }  
1 //Thread-2
2 s \rightarrow \text{table} = \text{NULL};
```

Figure 1.1: A real-world concurrency bug from Mozilla
1.2 Contribution

This paper presents BugTM, a transactional-memory (TM) inspired approach that efficiently and effectively recovers software from concurrency-bug failures at production runs.

The design of BugTM is motivated by the hardware transactional memory (HTM) technique already existing in modern CPU, particularly Intel TSX. Instead of using transactions to replace existing lock synchronization, BugTM explores a new way of using HTM — automatically inserting 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. While this paper’s implementation is based on Intel’s TSX, the mechanism and principles apply to other vendors’ HTM implementations.

Since HTM 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. While, the opportunity is obvious, challenges are abundant\(^1\).

- Performance challenges. High frequency of transaction uses would cause large overhead unacceptable for production runs. Unsuitable content of transactions, like illegal instructions\(^2\), high levels of transaction nesting, and long loops, would also cause performance degradation due to repeated and unnecessary transaction aborts.

- Correctness challenges (ensure semantics remains unmodified when concurrency bugs are not triggered at run time). Unpaired transaction-start and transaction-commit could cause software to crash. Deterministic aborts, such as those caused by illegal instructions, could cause software to hang if not well handled.

\(^1\) In Chapter 3 we show why brute-force and naive applications of Intel TSX will not work.

\(^2\) Certain instructions such as system calls will deterministically cause HTM abort and are referred to as illegal instructions
• Failure recovery challenges. In order for HTM to help failure recovery, the software has to be executing inside a transaction when failure happens and the corresponding transaction abort has to be carefully handled.

BugTM is carefully designed to address these challenges. It is implemented as a compiler pass that operates on source-code (multithreaded software) instrumenting it with TM mechanisms to allow concurrency bug recovery and remain transparent to software developers.

First, BugTM automatically inserts transactions around potential failure sites, such as where assert is called, to avoid excessive use of transactions, while maintaining good failure recovery capability.

Second, BugTM uses both static program analysis and run-time checking to carefully place transaction-starts and transaction-commits, so that BugTM transactions are guaranteed to contain no system calls, loops, or transaction nesting, and also guaranteed to always have perfectly paired transaction-starts and transaction-commits.

Third, BugTM inserts carefully designed transaction abort-fallback code so that aborts that might be caused by concurrency bugs get recovery chances, while aborts caused by illegal instructions and other reasons would not cause unnecessary re-executions and excessive performance degradations.

To further explore the design space of failure recovery, we also build a software version of BugTM, called BugTM$_S$. Comparing with (software) transactional memory techniques, BugTM$_S$ gives up the expensive conflict-detection functionality, which is often unnecessary for failure recovery, and maintains some version-management functionality, which is crucial to correct re-execution. BugTM$_S$ achieves better performance than BugTM, with some sacrifice in failure recovery capability and some extra capability in failure diagnosis.

We have conducted a thorough evaluation for BugTM and BugTM$_S$ 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 [22, 25, 50, 63, 64, 65]. Our evaluation shows
that BugTM and BugTM$_S$ can recover from more concurrency-bug failures than previous state of the art, ConAir, while still keeping good run-time performance (i.e., about 0.4% and 4.0% overhead on average for BugTM$_S$ and BugTM). Overall, BugTM and BugTM$_S$ greatly improve the state of art in production-run failure recovery for concurrency bugs. They present a novel way of using HTM, and provide two valuable points in the design space of record-and-replay for production-run failure recovery.
CHAPTER 2
BACKGROUND

Transactional Memory (TM) is a widely studied parallel programming construct [19]. Developers can wrap a code region in a transaction (Tx), and the underlying TM system guarantees its atomicity, consistency, and isolation. It is an interesting alternative to lock-based synchronization.

Most TM systems provide a set of operations to manage Tx. StartTx starts a Tx. CommitTx attempts to commit the current Tx. The commit attempt may succeed or fail, with the latter causing Tx abort. AbortTx explicitly aborts the current Tx. Transaction abort usually leads to the re-execution of the Tx, unless special fallback code is provided.

There are two main categories of TM systems, software transactional memory (STM) and hardware transactional memory (HTM) [17]. HTM has less overhead than STM but requires high implementation and verification cost. HTM has been implemented in IBM [16], AMD [2] and Sun [11] processors. Recently, Intel’s Transactional Synchronization Extensions (TSX) is available in commercial processors [1].

Intel TSX (RTM) provides a set of new instructions: XBEGIN, XEND, XABORT, and XTEST. We will denote the first three as StartTx, CommitTx, and AbortTx for generality. XTEST, referred to as TestTx by us for generality, checks whether the current execution is under an active Tx.

There are multiple causes for Tx aborts under RTM, each with their own unique abort code. Unknown abort is mainly caused by illegal instructions, such as exceptions and interrupts. The abort code is 0x00. Data conflict abort is caused by conflicting accesses from another thread — another thread accesses (writes) the write (read) set of the current Tx. Intel TSX leverages cache coherence protocol to detect these conflicts [1]. The abort code is 0x06. Capacity abort is due to out of cache capacity. The abort code is 0x08. Nested transaction abort happens when there are more than 7 levels nested transactions. The abort
code is 0x20. Finally, *manual abort* is caused by explicit AbortTx operation. Its abort code can be specified by programmers.
CHAPTER 3
BUGTM

3.1 High-Level Design

We first discuss the high-level idea about where to put Txs. We will compare and assess our idea and some strawman ideas based on performance and failure-recovery capability.

3.1.1 Strawman approaches

One approach is to chunk software to many segments and put every segment inside a hardware Tx [36]. This approach can potentially avoid many 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.

3.1.2 Our approach

In BugTM, we selectively put hardware Txs around places where failures are mostly likely to happen, like the invocation of an _assert_fail, an error-reporting function, the dereference of a shared pointer, etc.

Intuitively, this design would provide both a good performance and a good chance of re-execution and recovery for most concurrency-bug failures. In fact, it can indeed use a combination of proactive prevention and reactive recovery to handle failures caused by all common types of concurrency bugs, as shown in Table 3.1 and explained below.

---

1. In fact, BugTM can not only recover software failures, but also prevent software failures. We sometimes use the term failure recovery for both.
An atomicity violation (AV) happens when the atomicity of a code region $C$ is unexpectedly violated, such as the bug shown in Figure 1.1. It contributes to more than 70% of non-deadlock concurrency bugs based on empirical studies [32], and can be further categorized into sub-types depending on the nature of $C$, as demonstrated in Table 3.1. Conflicting accesses would usually trigger a rollback recovery before the failure occurs, shown by the dashed arrow lines in Table 3.1(a)(b)(c), benefitting from the strong atomicity guarantee of Intel TSX — a Tx will abort even if the conflicting access comes from non-Tx code. For the bug shown in Figure 1.1, 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. The re-execution of the Thread-1 Tx will then successfully avoid the failure.

An order violation (OV) happens when an instruction $A$ unexpectedly executes after, instead of before, instruction $B$, such as the bug shown in Figure 3.1. Different from AVs,
conflicting memory accesses related to OVs may not all happen inside a small window. In
fact, A may not have executed when a failure occurs in the thread of B. Consequently, the
Tx abort probably will be triggered by a software failure, instead of a conflicting access,
depicted by the dashed arrow line in Table 3.1(e). Fortunately, the rollback reexecution
will still give the software a chance to correct the unexpected ordering and recover from
the failure. Take the bug shown in Figure 3.1 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, the Tx will successfully commit.

Deadlock bugs occur when different threads each holds resources and circularly waits for
each other for other resources. As shown in Table 3.1(f), it can be recovered by Tx rollback
and re-execution too, as long as deadlocks are detected.

Of course, we do not expect BugTM to recover from all concurrency-bug failures. In
practice, some order violations cannot be recovered, if the failure thread is already too slow
and hence cannot recover by re-executing\(^2\). Some failures cause silent data corruption. As
a result, their failure sites cannot be pre-identified and cannot be surrounded by Tx. Some
failures require re-executing a long code region to recover, which cannot fit into one Tx.

Next, we will discuss in detail how BugTM surrounds failure sites with hardware Txs.
Specifically, we will present how BugTM automatically inserts StartTx, CommitTx, AbortTx,
and fallback (recovery) code into software, while targeting three goals: (1) good recovery
capability; (2) good run-time performance; (3) not changing the original program semantics.

### 3.2 BugTM design about AbortTx

BugTM considers the invocation of assertion-failure function `assert_fail` and the invocation
of error-reporting functions as failure instructions. BugTM puts an AbortTx wrapper func-

\(^2\) Most of them are caused by using already-freed memory resources, which can be tackled by traditional
memory-bug recovery/prevention tools [37, 43].
tion my\_xabort right before every failure instruction, so that a Tx abort and re-execution will be triggered right before a failure manifests. my\_xabort uses a unique abort code 0xFF for its AbortTx operation, so that BugTM can differentiate different causes of Tx aborts and handle them differently.

BugTM automatically inserts assertion checkings like checking whether a pointer parameter of a string-library function is null or not and whether a shared pointer variable is null or not right before its dereference. BugTM also automatically turns lock functions into time-out locks, with a long timeout indicating a potential deadlock failure. This is similar with how previous bug-detection [64] and failure-recovery [63] techniques identify failure instructions.

### 3.3 BugTM design about StartTx and CommitTx

#### 3.3.1 Challenges

We elaborate two key challenges associated with placing StartTx and CommitTx. First, poor placements could cause frequently-abort Txs. Illegal instructions (e.g., system calls) and high-level of TM nesting (>7 level) will deterministically cause aborts, while long Txs will abort more likely than short ones due to timer-interrupts and memory-footprint threshold. These aborts will not only hurt performance, but also hurt recovery — deterministic aborts of a Tx will eventually force us to execute the Tx region\(^3\) in non-transaction mode, leaving no hope for failure recovery. Second, poor placements could cause unpaired execution of StartTx and CommitTx, which affects both correctness and performance. When CommitTx executes without a pairing StartTx, the program will crash; when StartTx executes without a pairing CommitTx, the corresponding Tx will repeatedly abort.

We address the first challenge by carefully selecting locations for XBEGIN, XEND. We address the second challenge mainly through our design of my\_xbegin and my\_xend wrappers.

---

\(^3\) 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.
3.3.2 Where to XBEGIN and XEND

The key design principle is to minimize the chance of aborts that are unrelated to concurrency bugs. BugTM achieves this by making sure that its Txs do not contain function calls, which avoids system calls and many illegal instructions, or loops, which avoids large memory footprints. The constraint of not containing function calls will be relaxed in Section 3.5.

To place StartTx, for every failure instruction \( f \) inside a function \( F \), BugTM traverses backward through every path that connects \( f \) with the entrance of \( F \) on control-flow graph (CFG), and puts a my_xbegin, the BugTM wrapper of XBEGIN, right after the first function call instruction or loop-exit instruction or the entrance of \( F \), whichever encountered first.

To place CommitTx, BugTM puts my_xend 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 the re-execution region for failures that may happen 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.

Note that, one might think that we should just put my_xend right after \( f \). Unfortunately, this does not work, because correct execution will not touch \( f \) and it is difficult to know whether the execution still has chance to hit \( f \) or not.

3.3.3 How to StartTx and CommitTx

The above algorithm does not guarantee one-to-one pairing of the execution of StartTx and CommitTx. BugTM addresses this challenge through run-time TestTx checkings conducted in my_xbegin and my_xend. That is, StartTx will execute only when there is no active Txs; CommitTx will execute only when there exists an active Tx, as shown in Figure 3.3 and Figure 3.2.

Overall, our design so far satisfies performance, correctness, and failure-recovery goals
if(xtest())
    _xend(0xFF); //terminate an active transaction

Figure 3.2: BugTM wrapped function (my_xend) for CommitTx

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

3.4 BugTM design for fallback and retry

3.4.1 Challenges

It is not trivial to automatically and correctly generate fallback code for all Txs inserted
by BugTM. Inappropriate abort handling could lead to performance degradation, hang, and
lose failure-recovery opportunities.

3.4.2 Solutions

BugTM will check the abort code and react to different types of aborts differently. Specifi-
cally, BugTM implements the following fallback strategy through its my_xbegin wrapper, as
shown in Figure 3.3.

Aborts caused by explicit AbortTx inserted by BugTM indicates software failures. We
should re-execute the Tx under HTM mode, hoping that the failure will dissappear in the
retry (Line 12–15). To avoid endless retry, BugTM keeps a retry-counter Retrytimes, as
shown in Figure 3.3.

Data conflict aborts (Line 12–15) are caused by conflicting accesses from another thread.
They are handled the same way as above, because they could be related to concurrency bugs.
if(_xtest() == 0){ //no active Tx
    Retrytimes = 0;
    prev_status = -1;
retry: if((status = _xbegin()) == _XBEGIN_STARTED){
    //Tx starts
    }else{ //abort fallback handler, no active Tx at this point
        Retrytimes++;
        if(status==0x00 || status==0x08){ //unknown or capacity abort
            if(!(prev_status==0x00 && status==0x00) &&
                !(prev_status==0x08 && status==0x08))
                prev_status=status; goto retry;
        }else if(status==0x06 || status==0xFF000001){
            if(Retrytimes < RetryThreshold)
                {prev_status=status; goto retry;}
        }
    }
    //continue execution in non-Tx mode
}

Figure 3.3: BugTM wrapped function (my_xbegin) for StartTx

if(!e){
    my_xbegin();
    ...
    if(!e){
        my_xabort();
        ___assert_fail(...);
    }
    ...
    my_xend();

Figure 3.4: BugTM code transformation for assert(e)

Unknown aborts and capacity aborts (Line 8–11) 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, because BugTM Txs are non-nested.

Note that, the above wrapper function not only implements the fallback strategy, but also allows easy integration into the target software, as demonstrated in Figure 3.4.
3.5 Inter-procedural BugTM and Others

3.5.1 Inter-procedural BugTM

The above algorithm allows no function calls or returns in Txs, keeping the whole recovery attempt within one function $F$. This is too conservative as there are many functions that contain no illegal 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 extention 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.

Finally, we adjust our strategy about when to finish a BugTM Tx. The basic BugTM 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 strategy is to change the my_xend wrapper inserted at function exits, and make it take effect only when the function is the one which starts the active Tx.

3.5.2 Optimizations

A necessary condition for a successful failure recovery is that the failure instruction has control or data dependency on a shared-variable read that is re-executed during recovery attempts. If there exists no such read instruction, the execution of the failure instruction will be deterministic during re-execution, and hence the failure will not be recovered. As an optimization, we eliminate Txs that contain no failure-dependent shared-variable reads.
CHAPTER 4

BUGTM$\_5$

4.1 Exploring the Design Space

HTM in BugTM and software-based setjmp/longjmp in previous state-of-the-art ConAir [63] 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 [45, 54].

To achieve the best of both approaches, one can take the three TM principles of conflict detection, conflict resolution, and version management, and implement them in software targeted for concurrency bug-recovery only. Conflict detection can be completely jettisoned, as it is expensive to implement in software and conflicts often do not lead to failures. Conflict-resolution needs to be applied only for shared-variable reads. Reading the latest copy means delaying the current Tx (thread), whereas reading an earlier one using an undo log means delaying the conflicting one. Finally, version management for shared-variable writes, which does not exist in ConAir, can extend the types of regions that can be reexecuted for recovery.

Orthogonally, the software-based setjmp/longjmp can be composed with HTM to handle the re-execution of some illegal instructions. Since Intel TSX allows setjmp/longjmp to execute inside Txs, we can easily get BugTM$^+$ by applying BugTM to a program already hardened by ConAir or any setjmp/longjmp recovery scheme and obtain the union of each component’s recovery capability.

Our plan The remainder of 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 by offering better performance and more design flexibility at the cost of losing some recovery capability owned by BugTM. We will first give some backgrounds about ConAir, and then present our two extensions.

### 4.2 Background: ConAir

ConAir is a static code transformation tool built upon LLVM compiler infrastructure [28]. ConAir first identifies failure instructions and then inserts setjmp and longjmp into software, so that a longjmp will be executed right before a failure instruction is executed. The execution of longjmp will initiate a re-execution starting from an earlier setjmp. ConAir conducts little version management, and hence does not allow its re-execution region to contain any writes to shared variables, referred to as killing writes or $w_{\text{kill}}$.

This constraint severely affects the recovery capability of ConAir. As shown in Table 3.1, it fundamentally cannot handle any RAW violations (e.g., the bug in Figure 1.1) and WAR violations because it cannot re-execute shared variable writes. Even for those root-cause types that it can handle, its recovery capability is limited. For example, Figure 4.1 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 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.

### 4.3 Deferred Writes for Failure-Unrelated Killing Writes

Some killing writes are not related to potential failures, such as the $*\text{buf} = ' ' \text{statement}$ in Figure 4.1. BugTM$_S$ tries moving them to after the failure instruction, emulating the
Figure 4.1: BugTM$_S$ deferred write transformation, denoted by ‘+’ and ‘-’, makes a ConAir-unrecoverable bug recoverable.

defered write version-management technique in TM, so that the re-execution region can go beyond these killing writes.

4.3.1 Feasibility checking

For each $w_{\text{kill}}$ and the corresponding failure site $f$, BugTM$_S$ checks two things: (1) whether moving $w_{\text{kill}}$ would change program semantics; and (2) whether the moving will cut short other failure sites’ re-execution regions. If $w_{\text{kill}}$ 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 $w_{\text{kill}}$ 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 $w_{\text{kill}}$.

If there is no such dependency, moving $w_{\text{kill}}$ is guaranteed not to change program semantics$^1$. 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 Figure 4.1, we try code transformation to eliminate the dependency. Note that, since $w_{\text{kill}}$ writes to a shared variable, the stack variable dependency here must be a write-after-read dependency as the one between

1. 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 undependent loads.
1 - W.KILL
2 + flag = TRUE;
3  ...
4 + if(flag){
5 +   W.KILL //new location
6 +   flag = FALSE;
7 + }

Figure 4.2: Moving a killing write (flag is initialized FALSE)

Line 4 and Line 5 in Figure 4.1.

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 Figure 4.1.

### 4.3.2 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 function \( F \), 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 Figure 4.2.

Now BugTM\(_S\) can recover from some ConAir-unrecoverable failures, like the one shown in Figure 4.1. It has almost no performance impact to the original ConAir, and guarantees
1 g1 = 1;
2 setjmp;
3 ret = setjmp;
4
5 tmp = g1;
6 if (!tmp)
7 {
8     ASSERTFAIL;
9     //failure site
10    }
11 else
12 {
13     tmp = ckpt_g1;
14     if (!tmp)
15         ASSERTFAIL;
16         //failure site
17     }
18 }

(a) Base  (b) ConAir  (c) BugTM

Figure 4.3: Memory-checkpoint example

to preserve program semantics.

4.4 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 the undo log technique in TM. This extra option can help recover from some Read-After-Write (RAW) atomicity violations, while preserving program semantics and introducing little overhead.

4.4.1 Basic Algorithm

Figure 4.3a shows a toy example of RAW atomicity violation: if another thread changes the value of \(g1\) 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 \(w_{\text{kill}}\) in Line 1 and can never change the failure-triggering value returned by Line 6 in
Figure 4.3b. However, if the value of g1 could be checkpointed right at Line 1, as shown in Figure 4.3c, the failure could be recovered.

In general, taking a memory checkpoint is straightforward: simply create a local variable \texttt{ckpt\_g1} and copy the right hand side of the g1-assignment to \texttt{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 Figure 4.3c Line 5–8, BugTM$_S$ makes the read of g1 conditional on this return value: the read will access the checkpointed value in \texttt{ckpt\_g1} during re-execution and up-to-date value in g1 during regular execution.

The above BugTM$_S$ transformation can successfully recover from the failure on Line 11 in Figure 4.3c, because the checkpointed-reexecution essentially guarantees the RAW atomicity between Line 1 and Line 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.

4.4.2 Final Algorithm

When encounters a \texttt{w\_kill} which the failure site \texttt{f} depends upon, BugTM$_S$ checks whether there exists a read \texttt{r} that satisfies all of the following conditions: (1) \texttt{r} may read from the same memory location written by \texttt{w\_kill}; (2) \texttt{f} depends on \texttt{r}; (3) \texttt{r} and \texttt{w\_kill} are inside the same basic block. If such a read \texttt{r} is found, BugTM$_S$ transforms the code region between \texttt{w\_kill} and \texttt{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 \texttt{w\_kill} and \texttt{r} including \texttt{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). Therefore, BugTM$_S$ conservatively makes checkpoints to preserve program semantics. For a similar reason, we only handle $r$ and $w_{\text{kill}}$ inside the same basic block, because otherwise there could be inconsistent load orders among different paths from $\text{killw}$ 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 $w_{\text{kill}}$ 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$_{\text{ret}}$.

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 still 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 shown in Figure 1.1 as an example, by using the check-pointed value of $s \rightarrow \text{table}$ at Line 4, BugTM$_S$ will recover from the original failure on Line 5. However, after the re-execution ends at Line 4, the regular program execution will continue using the update-to-date value of $s \rightarrow \text{table}$, which is NULL. Software probably will still fail, just later than the one originally recovered by BugTM$_S$. To fundamentally recover from this failure, we will need BugTM.
CHAPTER 5

FAILURE DIAGNOSIS

BugTM₅ supports failure diagnosis through the root-cause inference routine shown in Figure 5.1 and extra logging during recovery. The root-cause inference shown in Figure 5.1 is mostly straightforward. We can obviously make inference based on the failure symptom (Line 2) and the rollback scheme (Line 4). The rationale of diagnosis based on the number of re-executions (Line 6 and 8) is the following. If the recovery success relies on a code region $C$ in the failure thread to re-execute atomically, probably one re-execution attempt is sufficient, because another unserializable interleaving during re-execution is very rare. This case applies to RAW and RAR violation, as shown in Table 3.1. If the recovery success relies on something to happen in another thread, multiple re-executions are probably needed. This applies to WAW violations and order violations, as also shown in Table 3.1.

BugTM₅ enhances ConAir to log memory accesses’ read/write types, addresses, values, and synchronization operations during re-execution. This log will help failure diagnosis, with no run-time overhead and only slight delay to recovery.

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

```
1 Input: information from a successful recovery
2 if (failure site was a lock-timeout) 
3   output: deadlock 
4 else if (first re-execution with checkpoint succeeds) 
5   output: RAW atomicity violation 
6 else if (first re-execution without checkpoint succeeds) 
7   output: RAR atomicity violation 
8 else if (re-execution succeeds after multiple attempts) 
9   output: Order Violation or WAW atomicity violation
```

Figure 5.1: Root-cause diagnosis based on failure recovery
BugTM offers much less diagnostic information than BugTM$_S$, because there are a wide variety of reasons behind its transaction aborts and we skip the discussion here.
CHAPTER 6
METHODOLOGY

Implementation  BugTM and BugTM$_S$ are both implemented using LLVM compiler infrastructure (v3.6.1). We also use the LLVM default pointer-alias analysis. We have obtained the source code of ConAir, also built upon LLVM, from its authors, and built BugTM$_S$ upon it. We did not make any other changes to ConAir, except for the two extensions discussed in Section 4. All the experiments are conducted on 4-core Intel Core i7-5775C (Broadwell) machines with 6MB cache, 8GB memory running Linux version 2.6.32.

Benchmark suite  Our benchmark suite includes 29 bugs, including all the real-world bug benchmarks in a set of previous papers on concurrency-bug detection, fixing, and avoidance [22, 25, 50, 63, 64, 65]. They cover all common types of concurrency-bug root causes and failure symptoms.

Our benchmark suite includes 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 modular router), and a variety of desktop applications (e.g., zsnes game simulator, PBZIP2 file compressor, Mozilla JavaScript Engine and XPCOM). The sizes of these applications (modules) range between around 50K to almost 1 million lines of code. Finally, our benchmark suite contains 3 extracted benchmarks: Moz52111, Moz209188, and Bank.

Note that, the goal of BugTM is to recover production-run failures, not to detect previously unknown bugs. Therefore, our bug benchmarks use previously known bugs that we know how to repeat. Although the bugs are all previously known, in all our experiments, the evaluated failure-recovery tools do not rely on any knowledge about specific bugs. They simply harden code around all asserts, error-reporting functions, dereferences of shared pointers, lock invocations, and outputs.
**Evaluation setups and metrics** We will mainly measure the recovery capability and overhead of BugTM and BugTM$_S$. We will also evaluate and compare with ConAir [63], the state of the art concurrency-bug recovery technique.

To measure recovery capability, we follow the methodology taken by previous work [24, 63]. We 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 recovery capability. To measure the run-time overhead. We run the original software, **without** any sleeps, under the bug-triggering workload with each tool applied. We report the average overhead measured during 100 failure-free runs.

In addition, we also evaluate alternative designs BugTM, such as not conducting interprocedural 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 and representative client and server applications.
CHAPTER 7
EXPERIMENTAL RESULTS

Overall, as shown in Table 7.1, BugTM and BugTM$_S$ can both recover from a wide variety of concurrency-bug failures with good performance, achieving better recovery capability than state-of-the-art ConAir. In the following, we explain recovery capability, performance, and other results in details.

7.1 Failure recovery capability

Among all three techniques, BugTM has the best recovery capability, successfully recovering from 19 out of 29 concurrency-bug failures$^1$. BugTM$_S$ can completely recover from 16 benchmarks and partly recover from another two benchmarks, 18 in total. State of the art ConAir can only recover from 14 benchmarks.

Recoverable benchmarks  We first compare BugTM, BugTM$_S$, and ConAir among the 20 benchmarks that at least one of them can help recover.

ConAir can only recover from 14 failures, mainly because it does not allow shared-variable writes in its re-execution region. As a result, it cannot recover from any RAW bugs or WAR bugs (there are 4 such bugs in Table 7.1). It also cannot recover from two RAR bugs including the one shown in Figure 4.1, because there are failure-unrelated shared-variable writes in their buggy code regions.

BugTM$_S$ can recover from all failures that ConAir can recover from. In addition, its deferred write technique helps it to successfully recover from the two RAR violation failures ConAir cannot handle. 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

---

1. Our experiments show that BugTM$^+$ discussed in Section 4.1 can recover from 20 failures with similar performance as BugTM.
<table>
<thead>
<tr>
<th>Root Cause</th>
<th>Recovered?</th>
<th>Run-time Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL2011</td>
<td>RAR</td>
<td>✓  ✓</td>
</tr>
<tr>
<td>MySQL3596</td>
<td>RAR</td>
<td>✓  ✓</td>
</tr>
<tr>
<td>MySQL38883</td>
<td>RAR</td>
<td>✓  ✓</td>
</tr>
<tr>
<td>Apache21287</td>
<td>RAW</td>
<td>✓  ✓*</td>
</tr>
<tr>
<td>Moz-JS18025</td>
<td>RAW</td>
<td>✓  ✓*</td>
</tr>
<tr>
<td>Moz-JS142651</td>
<td>RAW</td>
<td>–</td>
</tr>
<tr>
<td>Bank</td>
<td>WAR</td>
<td>–</td>
</tr>
<tr>
<td>Moz-ex52111</td>
<td>WAW</td>
<td>✓  ✓*</td>
</tr>
<tr>
<td>Moz-ex209188</td>
<td>WAW</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>MySQL791</td>
<td>WAW</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>MySQL16582</td>
<td>WAW</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>Click</td>
<td>OV</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>FFT</td>
<td>OV</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>HTTrack</td>
<td>OV</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>Moz-xpcom61369</td>
<td>OV</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>Transmission</td>
<td>OV</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>zsnes</td>
<td>OV</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>HawkNL</td>
<td>D.D.</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>Moz-JS79054</td>
<td>D.D.</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>SQLite1672</td>
<td>D.D.</td>
<td>✓ ✓</td>
</tr>
</tbody>
</table>

| Tot./Avg   | 14 18* 19 | 0.31% 0.42% 4.04% |

Table 7.1: Overall results and comparison (CA: ConAir; *: failures partly recovered; red font denotes > 4% overhead; Moz-JS: Mozilla JavaScript Engine; Moz-ex: benchmarks extracted from Mozilla; Moz-xpcom: Mozilla COM model.)
because the bug involves complicated control flows. Moz-JS18025 is demonstrated in Figure 1.1. As discussed earlier, BugTM$_S$ can help software recover from the failure shown in the figure, but cannot prevent subsequent failures caused by the NULL value of s$\rightarrow$ table. Apache21287 can be recovered by BugTM$_S$ with about 50% probability, depending on which bug-related thread fails first. Finally, BugTM$_S$ fundamentally cannot handle WAR violations, as discussed at the end of Section 4.

BugTM can successfully recover from all the 6 concurrency-bug failures that ConAir cannot help in Table 7.1. BugTM cannot recover from the Transmission bug, because recovering this bug requires re-executing malloc, an illegal operation for Intel TSX. In fact, malloc is allowed in some more sophisticated TM designs [45, 54].

**Unrecoverable benchmarks** There are 9 benchmarks that none of the three tools can help recover. As shown in Table 7.2, there are mainly three reasons for the failed recovery attempts. Some of these issues go beyond the scope of failure recovery, yet others are promising to address in the future.

First, some order violations cause failures when the failure thread is unexpectedly slow. In these cases, re-executing the failure thread would not help correct the timing. Fortunately, both failures in PBZIP2 and x264 can be prevented by delaying resource deallocation, a prevention approach proposed before for memory-bug failures [37, 43].

Second, some failures are difficult to detect, not to mention recovering. For example, Cherokee326, Apache25520, and MySQL169 lead to silent buffer and log corruption that are very difficult to detect. Tackling them goes beyond the scope of failure recovery.

Third, some failures cannot be recovered due to un-reexecutable instructions. These issues are promising to address. For example, Intel TSX does not support putting memcpy, cond_wait, or I/O into its Txs. More sophisticated TM designs [45, 54] would be able to help recover these failures.
Table 7.2: Reasons for unrecoverable benchmarks (we use numbers to index multiple reasons)

<table>
<thead>
<tr>
<th></th>
<th>BugTM$_S$</th>
<th>BugTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenOffice44627</td>
<td>Failure thread is too slow in order violation</td>
<td></td>
</tr>
<tr>
<td>PBZIP2x264</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache25520</td>
<td>1. Failure difficult to detect</td>
<td></td>
</tr>
<tr>
<td>Cherokee326</td>
<td>2. memcpy, I/O 2. memcpy</td>
<td></td>
</tr>
<tr>
<td>MySQL169</td>
<td>1. Failure difficult to detect</td>
<td></td>
</tr>
<tr>
<td>Apache42031</td>
<td>2. w$_{kill}$, I/O 2. I/O</td>
<td></td>
</tr>
<tr>
<td>MySQL29560</td>
<td>w$_{kill}$</td>
<td>cond_wait</td>
</tr>
<tr>
<td>Aget</td>
<td>w$_{kill}$, I/O</td>
<td>I/O</td>
</tr>
</tbody>
</table>

7.2 Performance

As shown in Table 7.1, BugTM$_S$ incurs less than 1% overhead for all benchmarks at run time, almost a free lunch for production failure recovery. BugTM incurs 4.04% overhead on average, lower than 1% for half of the benchmarks and lower than 4% for three quarters of the benchmarks.

To better understand the performance of BugTM and BugTM$_S$, Table 7.3 presents more details.

BugTM$_S$ and BugTM both insert many static re-execution regions into software and start many dynamic re-execution regions at run time, through setjmp and StartTx respectively. They did not use any knowledge about specific bugs and instead provide a broad failure-recovery support for software.

There are more dynamic setjmp in 8 benchmarks, while more dynamic StartTx in 12 benchmarks. Even for those that encounter fewer StartTx under BugTM than setjmp under BugTM$_S$, BugTM still incurs much larger overhead. The main reason is that Tx execution incurs more overhead than setjmp. The number of dynamic setjmp executed by BugTM$_S$ and the number of dynamic StartTx executed by BugTM could sometimes differ a lot (e.g., MySQL3596 and Moz-xpcom61369), because global/heap writes caused BugTM
<table>
<thead>
<tr>
<th>#setjmp</th>
<th>#StartTx</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL2011</td>
<td>5183</td>
</tr>
<tr>
<td>MySQL3596</td>
<td>4363</td>
</tr>
<tr>
<td>MySQL38883</td>
<td>4350</td>
</tr>
<tr>
<td>Apache21287</td>
<td>1265</td>
</tr>
<tr>
<td>Moz-JS18025</td>
<td>596</td>
</tr>
<tr>
<td>Moz-JS142651</td>
<td>605</td>
</tr>
<tr>
<td>Bank</td>
<td>2</td>
</tr>
<tr>
<td>Moz-ex52111</td>
<td>4</td>
</tr>
<tr>
<td>Moz-ex209188</td>
<td>2</td>
</tr>
<tr>
<td>MySQL791</td>
<td>4239</td>
</tr>
<tr>
<td>MySQL16582</td>
<td>9362</td>
</tr>
<tr>
<td>Click</td>
<td>7430</td>
</tr>
<tr>
<td>FFT</td>
<td>41</td>
</tr>
<tr>
<td>HTTrack</td>
<td>2884</td>
</tr>
<tr>
<td>Moz-xpcom61369</td>
<td>50</td>
</tr>
<tr>
<td>Transmission</td>
<td>1442</td>
</tr>
<tr>
<td>zsnes</td>
<td>602</td>
</tr>
<tr>
<td>HawkNL</td>
<td>29</td>
</tr>
<tr>
<td>Moz-JS79054</td>
<td>812</td>
</tr>
<tr>
<td>SQLite1672</td>
<td>489</td>
</tr>
</tbody>
</table>

Table 7.3: # of static and dynamic setjmp and StartTx in BugTM₅ and BugTM (Dyn.Freq.: # of dynamic StartTx per 100 μ-second; Abort%: percentage of aborted dynamic Txs.)
and BugTM$_S$ to place StartTx and setjmp in different basic blocks that were executed with hugely different frequencies.

The overhead of BugTM differs among benchmarks, ranging from 0.0% to 15%. As TM researchers found before, performance in TM systems is often complicated [6, 42]. We found the most indicating metrics for our benchmarks to be the frequency of StartTx at run time. As shown in the Dyn.Freq. column of Table 7.3, BugTM executes more than 1 StartTx per 100 micro second on average for 10 benchmarks, and incurs more than 1% overhead for 9 of them.

Moz-JS142651, Moz-JS18025, Moz-JS79054 are the only three benchmarks that incur more than 10% overhead for BugTM. Their StartTx frequency ranks 1st, 2nd, and 5th among all benchmarks. Their baseline runs less than 5 milli-seconds. In comparison, we have six benchmarks that run for more than one second in baseline (FFT, HawkNL, HTTrack, Moz-xpcom 61369, MySQL2011, Transmission). BugTM incurs lower than 0.7% overhead for all of them. For these three benchmarks, our investigation shows BugTM inserted Txs into some frequently executed and short-running utility functions in Moz-JSEngine. For example, js.AtomicAdd is invoked 1788 times; the Tx in it contributes to 16% of all dynamic Txs. Note that, these three benchmarks are all from Mozilla JavaScript Engine, which involves little I/Os and is just a component of Mozilla web-browser. If we apply BugTM to the whole browser, the overhead should be much smaller.

As shown in Table 7.3, the Tx abort rate is less than 1% for all benchmarks, benefitting from our careful BugTM design. Across all benchmarks, 95% of all aborts are unknown aborts (timer interrupts, etc.), 3% are data-conflict aborts; 2% are capacity aborts. As we will see in Section 7.4, the abort rates and overhead are much worse in alternative designs.
Table 7.4: Comparing BugTM with alternative designs (The %s are the overhead over baseline execution w/o any recovery scheme applied; ✓: failure recovered; ✗: failure not recovered.)

<table>
<thead>
<tr>
<th></th>
<th>BugTM</th>
<th>Intra-proc</th>
<th>Illegal-Ins</th>
<th>Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moz-JS18025</td>
<td>11.9%</td>
<td>✓</td>
<td>9.10%</td>
<td>✓</td>
</tr>
<tr>
<td>Moz-xpcom61369</td>
<td>0.45%</td>
<td>✓</td>
<td>0.44% ✗</td>
<td>0.54%</td>
</tr>
<tr>
<td>Moz-JS79054</td>
<td>14.8%</td>
<td>✓</td>
<td>14.4% ✓</td>
<td>17.5%</td>
</tr>
<tr>
<td>Moz-JS142651</td>
<td>15.1%</td>
<td>✓</td>
<td>10.0% ✗</td>
<td>24.0%</td>
</tr>
<tr>
<td>MySQL791</td>
<td>2.04%</td>
<td>✓</td>
<td>1.50% ✓</td>
<td>11.4%</td>
</tr>
<tr>
<td>MySQL2011</td>
<td>0.14%</td>
<td>✓</td>
<td>0.13% ✓</td>
<td>1.50%</td>
</tr>
<tr>
<td>MySQL3596</td>
<td>8.09%</td>
<td>✓</td>
<td>7.01% ✓</td>
<td>107%</td>
</tr>
<tr>
<td>MySQL38883</td>
<td>8.07%</td>
<td>✓</td>
<td>7.00% ✓</td>
<td>126%</td>
</tr>
<tr>
<td>MySQL16582</td>
<td>3.23%</td>
<td>✓</td>
<td>0.16% ✓</td>
<td>93.1%</td>
</tr>
</tbody>
</table>

7.3 Diagnosis

Our evaluation shows that BugTM_S can indeed provide useful diagnosis information for all the 18 benchmarks that it can help recover from. Particularly, for 10 out of these benchmarks whose root causes are order violations 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.

BugTM_S 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.

7.4 Alternative designs of BugTM

Table 7.4 shows the performance and recovery capability of three alternative designs of BugTM. Due to space constraints, we only show results on benchmarks in MySQL database server and Mozilla browser suite (non-extracted).
**Inter-procedural vs. Intra-procedural** BugTM uses the inter-procedural algorithm discussed in Section 3.5 by default. As shown by Table 7.4, the inter-procedural design adds 0.01 – 5.1 % overhead to its intra-procedural alternative. In exchange, there are 4 benchmarks in Table 7.4 that require inter-procedural re-execution to recover from. Specifically, recovering MySQL2011, Moz-xpcom61369, Moz-JS79054 have to re-execute not only the function $F$ where the failure occurs, but also part of the caller of $F$. As for Moz-JS142651, we need to re-execute a callee function of $F$ where a memory access involved in the atomicity violation resides.

**Including illegal instructions in Txs** Clearly, if BugTM did not intentionally exclude system calls from its Tx, more Txs will abort. Although BugTM only re-retries twice for such aborts, it may still hurt performance. Furthermore, these aborts may hurt recovery capability: if an illegal instruction is executed before a failure instruction in a Tx region $R$, $R$ will eventually execute in non-transaction mode to avoid endless aborts and lose the opportunity of failure recovery.

Table 7.4 confirms the above reasoning. This alternative design would incur much larger overhead. In fact, it incurs around 100% overhead for three MySQL benchmarks. It will also fail to recover from two benchmarks in the table.

**Including loops in Txs** Including loops in Txs could lead to more capacity aborts. As shown in Table 7.4, its impact is not as negative as including illegal instructions in Txs. However, it still raises the overhead of MySQL791 from about 2% to almost 12%. In fact, we have observed more Tx aborts for all benchmarks, although the overhead does not change much for most benchmarks.

**More Txs** We also tried randomly inserting more StartTx into software. The overhead increases significantly. For benchmark 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%.
CHAPTER 8
RELATED WORK

Concurrency bug detection Many automated detection tools have been proposed for a wide variety of concurrency bugs, including data races [7, 13, 20, 26, 38, 48, 53, 59], atomicity violations [33, 35], order violations [14, 50, 66], and deadlocks [56]. These tools aim to discover bugs during in-house testing and are not a good fit for production-run failure recovery — they often incur large overhead (e.g., 10X slowdowns) and cannot provide the desired bug/failure coverage.

Automated concurrency-bug fixing Static program analysis and code transformation techniques have been proposed to automatically generate patches for concurrency bugs [22, 24, 31, 56]. They work at off-line and rely on accurate bug-detection results. A recent work [21] 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. Its high-level idea is similar with the undo log extension in BugTM S. BugTM S only offers this as a re-execution option after software fails. Instead, this previous work aggressively applies this without knowing whether there are concurrency bugs in software. This different usage context leads to different design details.

General failure recovery Rollback and re-execution have long been a valuable recovery [43, 53] and debugging [10, 27, 41, 52] technique. Many rollback-and-re execution techniques target full system or full application replay and hence are much more complicated and expensive than that in BugTM and BugTM S. Feather-weight re-execution based on idempotency has been used before for recovering hardware faults [9, 12]. Using it to help recover from concurrency-bug failures was recently pioneered by ConAir [63]. BugTM and BugTM S
provide new design points in re-execution based failure recovery, and greatly improved the state-of-the-art ConAir.

Note that, BugTM not only achieves much better failure recovery capability and uses different re-execution mechanisms from ConAir, but also completely differs from ConAir in terms of its static code transformation design. The setjmp and longjmp used by ConAir have completely different performance and correctness implications from StartTx, CommitTx, and AbortTx, which naturally led to completely different designs in BugTM and ConAir.

Production-run failure diagnosis Diagnosing production-run failure is challenging. Sampling techniques have been proposed to lower its run-time overhead [4, 23, 30]. Triage [52] re-executes software from previous checkpoints when software fails, and applies dynamic bug detection during re-execution to diagnose production-run failures. Different from BugTM, Triage requires changes to operating systems to support full-application checkpoint-and-replay, and relies on bug-detection tools to help diagnose failures. Furthermore, Triage like its predecessor Rx [43] focuses on memory bugs. BugTM focuses on concurrency bugs, and leverages software’s reaction to failure-recovery attempts to diagnose failures.

Using TM techniques Lots of research has been done on HTM and STM [3, 5, 8, 15, 17, 18, 39, 44, 51]. Recent work has explored using HTM to speed up distributed transaction systems [57], race detection [62], etc. The HTM/software hybrid race detector [62] achieves much better performance than pure software race detectors, but still incurs about 4X slowdowns. Previous empirical studies have examined how to use Txs to manually patch concurrency bugs [55], and the experience of using Txs, instead of locks, in developing parallel programs [46, 60]. They all look at different ways of using TM systems from BugTM and BugTM.
CHAPTER 9
CONCLUSIONS

Concurrency bugs severely affect the availability of production-run software. This paper presents two TM-inspired techniques to help automatically recover concurrency-bug failures from production runs. BugTM automatically places HTM instructions into multi-threaded software and leverages HTM to prevent and recover concurrency-bug failures. It is capable of recovering failures caused by all major types of concurrency bugs and only incurs about 4% overhead on average in our evaluation. BugTMS uses STM inspired techniques to enhance the recovery capability of previous state-of-the-art ConAir. Although it cannot recover as many failures as BugTM, it incurs less than 1% overhead and can provide useful failure diagnosis information. Altogether, BugTM and BugTMS improve the state of the art of production run failure recovery, and present novel ways of using TM techniques.
REFERENCES


[50] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and


