TxRace: Efficient Data Race Detection Using Commodity Hardware Transactional Memory

Tong Zhang Dongyoon Lee Changhee Jung
Virginia Tech
{ztong, dongyoon, chjung}@vt.edu

Abstract

Detecting data races is important for debugging shared-memory multithreaded programs, but the high runtime overhead prevents the wide use of dynamic data race detectors. This paper presents TxRace, a new software data race detector that leverages commodity hardware transactional memory (HTM) to speed up data race detection. TxRace instruments a multithreaded program to transform synchronization-free regions into transactions, and exploits the conflict detection mechanism of HTM for lightweight data race detection at runtime. However, the limitations of the current best-effort commodity HTMs expose several challenges in using them for data race detection: (1) lack of ability to pinpoint racy instructions, (2) false positives caused by cache line granularity of conflict detection, and (3) transactional aborts for non-conflict reasons (e.g., capacity or unknown). To overcome these challenges, TxRace performs lightweight HTM-based data race detection at first, and occasionally switches to slow yet precise data race detection only for the small fraction of execution intervals in which potential races are reported by HTM. According to the experimental results, TxRace reduces the average runtime overhead of dynamic data race detection from 11.68x to 4.65x with only a small number of false negatives.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords data race; concurrency bug detection; hardware transactional memory; dynamic program analysis

1. Introduction

Data races are an important class of concurrency errors in shared-memory multithreaded programs. A data race occurs when two threads access the same memory location, at least one of the two accesses is a write, and their relative order is not explicitly enforced by synchronization primitives such as locks [8, 49, 56].

Data races often lie at the root of other concurrency bugs such as unintended sharing, atomicity violation, and order violation [46]. There are many real-world examples showing the severity of data races, including the northeastern blackout [64], mismatched Nasdaq Facebook share prices [57], and security vulnerabilities [73]. Moreover, data races make it difficult to reason about the possible behaviors of programs. The C/C++11 standards [8, 33, 34] give no semantics to programs with data races, and the data race semantics for Java programs [49] is considered to be too complex [69].

To address this problem, a variety of dynamic data race detectors [19, 21, 32, 58, 65, 75] have been proposed to help programmers write more reliable multithreaded programs. However, such dynamic tools often add too much runtime overhead. For example, FastTrack, a state-of-the-art happens-before based detector, incurs a 8.5x slowdown for Java programs [21] and a 57x slowdown for C/C++ programs [18]. For different set of benchmarks, Intel's Inspector XE incurs a 200x slowdown [62], and Google's ThreadSanitizer incurs a 30x slowdown [65]. Such high overhead hinders the widespread use of dynamic data race detectors in practice, despite their good detection precision.

This paper presents TxRace, a new software data race detector that leverages commodity Hardware Transactional Memory (HTM) to speed up dynamic data race detection. Transactional Memory (TM) [28] was proposed to simplify concurrent programming as a new programming paradigm, and hardware support for transactional memory has recently become available in commodity processors such as Intel's Haswell processor [29, 31]. TxRace exploits the observation that the conflict detection mechanism of HTM can be repurposed for lightweight data race detection in conventional multithreaded programs not originally developed for TM.

However, naively leveraging HTM does not automatically guarantee efficient data race detection. Rather, the limitations of the current commodity HTMs expose three challenges in using them for data race detection: (1) As the

ASPLOS '16 April 02-06, 2016, Atlanta, GA, USA.

DOI: http://dx.doi.org/10.1145/2872362.2872384

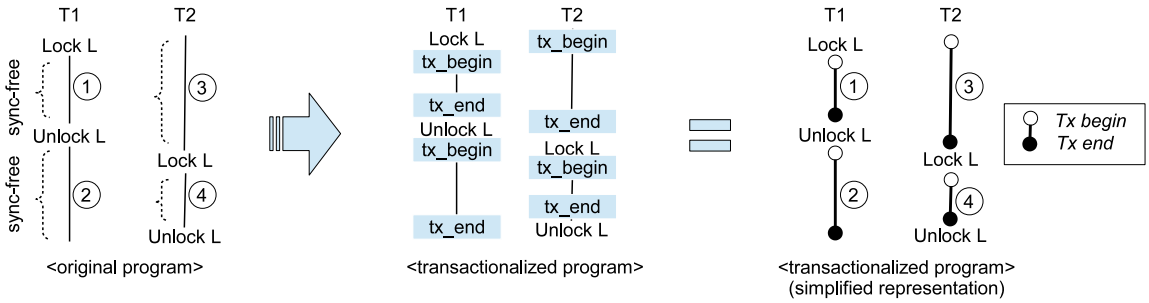

Figure 1: Transactionalization. TxRace statically instruments synchronization-free regions into transactions and leverages HTM for lightweight data race detection at runtime.

HTMs are designed to transparently guarantee atomicity and isolation between concurrent transactions, they do not provide a way to pinpoint racy instructions or conflicting addresses; (2) Since data conflicts are detected at the cache line granularity, false alarms could be reported due to the false sharing; and (3) Due to the best-effort nature of existing commodity HTMs, transactions may abort for reasons other than data conflicts, including exceeding the hardware capacity, interrupts and exceptions.

To overcome these challenges, TxRace first instruments a multithreaded program to transform all code regions between synchronizations (including critical sections) into transactions (Figure 1). At runtime, TxRace then performs a two-phase data race detection comprising fast and slow paths. During the initial fast path, TxRace detects potential data races using the low-overhead data conflict detection mechanism of HTM. In this stage, the detected races are only potential races, as the conflict might be due to false sharing in the cache line. When a data conflict is detected, the current HTMs do not identify the instruction that caused the transaction to abort, the conflicting address, or the other conflicting transaction. TxRace addresses this problem by artificially aborting all the (in-flight) concurrent transactions, rolling back them to the state before the data conflict occurred, and then performing software-based *sound*[1] (no false negative) and *complete* (no false positive) data race detection [21, 32, 65] for the concurrent code regions. This work refers to the rollback and subsequent re-execution with software-based data race detection as the slow path, which enables TxRace not only to pinpoint racy instructions but also to filter out any false positives. TxRace also relies on the slow path to cover the code regions, which cannot be monitored by transactions due to the limitations of existing commodity HTMs. This conservative approach reduces the chance of missing data races at the cost of runtime overhead, and TxRace includes an optimization technique to avoid repeated capacity aborts.

[1] In this paper, a dynamic analysis is *sound* if it does not incur false negatives; and *complete* if it does not report false positives, for the analyzed execution as in [16, 21, 22, 72]. Some [6, 9, 19] refer to a dynamic analysis to be *precise* (instead of *complete*) when there is no false positive, and others [18, 66] swap the definitions of soundness and completeness.

The experimental results show that using Intel's Restricted Transactional Memory (RTM) and Google's ThreadSanitizer (TSan) for the fast and slow paths respectively, TxRace achieves runtime overhead reduction of dynamic data race detection from 11.68x (TSan) to 4.65x (TxRace) on average. Using an HTM-based detector during the fast path may lead to missing data races if they do not overlap in concurrent transactions (and for other reasons). Nevertheless, TxRace incurs only a few false negatives (high recall of 0.95) for tested applications.

This paper makes the following contributions:

- To the best of our knowledge, TxRace is the first software scheme that demonstrates how commodity hardware transactional memory can be used to build a lightweight dynamic data race detector.
- TxRace proposes novel solutions to address the challenge in designing HTM-based data race detector. They enable TxRace to pinpoint racy instructions, remove false data race warnings caused by false sharing, and handle non-conflict transactional aborts efficiently.
- The paper presents experimental results showing cost effectiveness of TxRace compared to a state-of-the-art happens-before based data race detector and its random sampling based approach.

2. Background and Challenges

This section briefly introduces hardware transactional memory systems and discusses the challenges in using them in data race detection.

2.1 Hardware Transactional Memory

Transactional Memory (TM) [28] provides programmers with transparent support to execute a group of memory operations in a user-defined transaction in an atomic (all or nothing) and isolated (the partial state of a transaction is hidden from others) manner. Hardware support for transactional memory has been implemented in IBM's Blue Gene/Q supercomputers [26] and System z mainframes [35]; Sun's (canceled) Rock processor [17]; Azul's Vega processor [14]. Recently, HTM has become available in commodity pro-

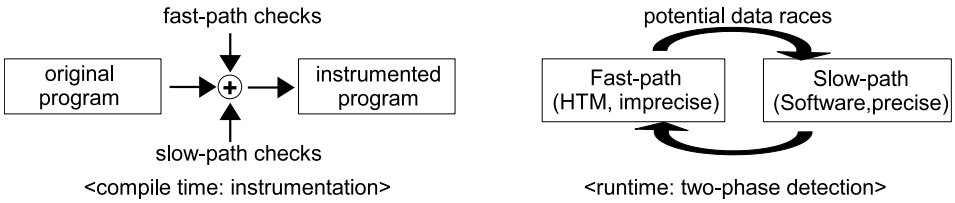

Figure 2: TxRace Overview

cessors used in desktops such as Intel's Haswell processor [29, 31].

TxRace leverages Intel's Transactional Synchronization Extensions (TSX) introduced in the Haswell processors. Intel TSX includes Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM) supports, where the latter enables general-purpose transactional memory. Intel RTM provides a new set of instructions comprising *xbegin*, *xend*, *xabort*, and *xtest* to help programmers initiate, complete, abort a transaction, and check its status, respectively. Intel RTM uses the first level (L1) data cache to track transactional states, and leverages the cache coherence protocol to detect transactional conflicts [23, 74]. Intel RTM supports *strong isolation*, which guarantees transactional semantics between transactions and non-transactional code [51]. For conflict management, Intel RTM uses a simple *requester-wins* policy in which on a conflicting request, the requester always succeeds and the conflicting transactions abort [7].

2.2 Challenges in Using HTM for Race Detection

At first glance, it might be expected that HTMs can trivially provide lightweight data race detection. However, the commercial HTMs, including Intel RTM, share limitations that hinder their adoption for data race detection.

First, though HTMs can detect the presence of data conflicts and abort, HTMs including Intel RTM do not provide programmers with the problematic instructions that caused the transaction to abort, or with the affected memory addresses. Moreover, the concurrent transactions to which the competing instructions belong may have been successfully committed according to TM semantics. This implies that programmers cannot reason about the pairs of memory accesses involved in the data race.

Second, HTMs detect data conflict by leveraging a cache coherence mechanism. Conflicts are therefore discovered at the cache-line granularity (64-bytes in the Intel Haswell processor). This may produce false warnings in data race detection due to non-conflicting operations on variables that share a cache line. By comparison, traditional dynamic data race detectors identify data races at the word (or byte) granularity, significantly reducing the likelihood of false positives.

Third, HTMs have bounded resources, and irrevocable I/O operations are not supported. Intel RTM does not support arbitrarily long transactions, simply aborting any transactions exceeding the capacity of the hardware buffer for transactional states [27, 61]. Moreover, changing privilege level always forces a transaction abort. The implication is that a transaction should not include any system call.

Finally, a transaction in best-effort (non-ideal) commodity HTMs including Intel RTM may be aborted for an unknown reason (neither due to data conflict nor due to capacity overflow). Intel's reference manuals [29, 31] illustrate some causes of unknown aborts, such as operating system context switches for interrupt or exception handling. However, neither the exact abort reason nor the problematic instruction is provided to programmers, which makes it hard to work around such a transaction abort.

3. Overview of TxRace

TxRace is a lightweight software dynamic data race detector that leverages hardware transactional memory support in modern commodity processors. The key insight is that TxRace can detect potential data races with a very low runtime overhead by initially relying on the conflict detection mechanism of HTM. The detected races are potential because the data conflict between transactions might be caused by false sharing in the same cache line. When transactions commit without data conflicts, TxRace incurs only the small runtime overhead of the transactional execution. In this sense, this HTM-based check is called *fast path* in which TxRace first takes to quickly identify potential data races. To address the aforementioned challenges of HTM-based detectors, on a data conflict, TxRace artificially aborts all concurrent (in-flight) transactions, rolls them back to the state before the data conflict occurs, and performs software-based sound and complete data race detection in an on-demand manner. Such re-execution with software-based detection is called *slow path*, which allows TxRace not only to pinpoint racy instructions but also to filter out false positives caused by false sharing.

Figure 2 shows an overview of TxRace. It consists of a compile-time instrumentation and a two-phase data race detection at runtime. TxRace inserts fast path transactional codes (e.g., *xbegin*, *xend*) and slow path sound and complete data race detection codes (e.g.,FastTrack [21], ThreadSanitizer [65]) into the original program at compile-time. Then, TxRace makes use of the two-phase data race detection at runtime. This allows TxRace to selectively perform sound and complete (but slow) data race detection for only a small fraction of the whole execution, leading to significant run-

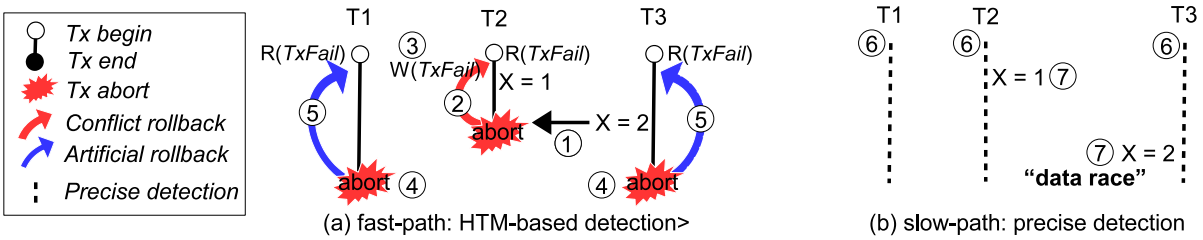

Figure 3: TxRace Runtime Example

time overhead reduction compared to a traditional dynamic data race detection.

As illustrated in Figure 1, TxRace transforms program regions between synchronizations (*synchronization-free regions*) into transactions. Then, TxRace performs HTM-based race detection between program regions that overlap in parallel at runtime. For example, for an execution where program regions ① and ③ overlap, TxRace checks potential data races between those two regions. Similarly, program regions between ② and ③; or between ② and ④ are checked when they run concurrently. On the other hand, the original synchronization lock *L* prevents the program regions ① and ④ (and corresponding transactions) from being overlapped at runtime. Therefore, the HTM will never observe conflicting memory accesses in critical sections protected by the same lock. In the following figures, a white circle corresponds to the beginning of a transaction, a black circle to its end.

After compile-time instrumentation, TxRace detects data races at runtime using the fast and slow paths as follows. Figure 3 shows an example with three threads, T1, T2, and T3, each performing one transaction. Suppose the shared variable *X* is not protected by the common lock, so that the transactions of T2 and T3 may run concurrently. During the fast path, each transaction begins by first reading a shared global flag named *TxFail* (as a part of instrumented code together with *xbegin*). Then, TxRace relies on HTM to detect data conflicts. Suppose a transaction in T3 causes another transaction in T2 to abort by accessing the shared variable *X* (step ①). Intel RTM employs a *requester-wins* conflict resolution strategy in which on a conflicting request, the requester always succeeds and the conflicting transactions abort [7]. Thus, the concurrent transactions of T1 (no conflict) and T3 (winner) may proceed further. To pinpoint the precise data race condition, TxRace immediately aborts the in-flight transactions by making the aborted transaction in T2 update *TxFail* (step ③) right after its rollback. Intel RTM supports *strong isolation* that guarantees transactional semantics between transactions and non-transactional code [51]. Together with the requester-wins policy, the strong isolation property in Intel RTM cause a transaction to abort if there is a conflicting access from a non-transaction code. Therefore, the update to *TxFail* causes all the concurrent transactions to abort artificially (step ④) as they have read *TxFail* at the beginning of the transaction. When all the concurrent transactions are rolled back (step ⑤), they resume execution on the slow path in which HTM is no longer used (step ⑥), but software-based sound and complete data race detection is performed (step ⑦) instead. When the slow path finishes for the program regions where potential data races are detected, TxRace switches back to the fast path in which HTM is used for the next program regions.

4. Fast Path HTM-based Race Detection

This section first describes how TxRace instruments original programs to detect potential data races as well as how it handles different types of transactional aborts, and then discuss optimization techniques used for reducing performance overhead.

4.1 Transactionalization

To exploit HTM for potential data race detection, TxRace transforms a code region between synchronization operations (including a critical section) into a transaction as illustrated in Figure 1. To be specific, at compile-time, TxRace inserts transaction begin instructions (*xbegin*) at thread entry points and after synchronization operations; and transaction end instructions (*xend*) at thread exit points and before synchronization operations. System calls require special consideration due to HTM limitations. Intel RTM, for example, aborts a transaction if a change in privilege level takes place. Consequently, TxRace ends the current transaction prior to each system call and begins a new transaction immediately after the system call in order to guarantee forward progress.

Furthermore, TxRace instruments each transaction to read the shared flag *TxFail* immediately after *xbegin*. As discussed in Section 3, Intel RTM uses the *requester-wins* policy that allows the requester to succeed on a conflicting request and aborts the conflicting transactions [7]; and supports *strong isolation* that guarantees transactional semantics between transactions and non-transactional code [51]. These two properties cause an in-flight transaction to abort on a conflicting access from non-transaction code. As TxRace makes transactions read *TxFail* when they start, when a data conflict is detected, the aborted transaction can artificially abort other in-flight transactions by writing to the flag *TxFail*. For example in Figure 3, when the aborted transaction in T2 writes to *TxFail*, the concurrent in-flight transactions

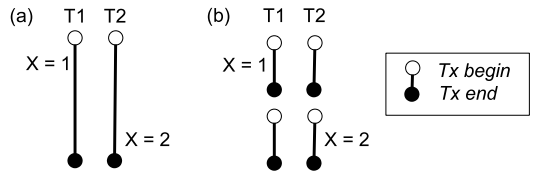

Figure 4: (a) Race detected with long transactions (b) Race missed with short transactions

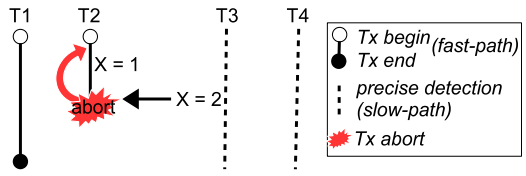

Figure 5: Detecting data races between fast and slow paths using the strong isolation property of HTM

in T1 and T3 which have read *TxFail* get aborted. Similar techniques have been used to abort in-flight transactions in hybrid TM systems [11, 15, 44] or to enable transactional lock elision [2].

HTMs can only detect those data races that result in conflicts between concurrent transactions. This suggests that maximizing the size of the transactions inserted at compile-time will minimize the likelihood of false negatives. It would be ideal to transform each synchronization-free region into a single transaction. However, as mentioned above, TxRace cuts transactions across systems calls inevitably. A performance optimization called *loop-cut* discussed further in Section 4.3 may also lead to cut a transaction originally formed for a synchronization-free region.

Figure 4 (a) and (b) show that the length of transactions can affect the detection of data races. Both (a) and (b) show unsynchronized and potentially-concurrent writes to the shared variable *X* from threads T1 and T2: a race condition. In (a), each thread executes a single lengthy transaction. The long transaction length increases the likelihood that the two transactions will overlap, and in this case the data race on *X* is detected. In (b), each thread executes two transactions, and the data race on *X* is more likely to be missed (a false negative) as a result. As shown in (b), suppose the first transaction in T1 includes *X=1* and successfully commits prior to the beginning of the second transaction in T2, which includes *X=2*. However, if the two transactions do not overlap, then neither will abort. For this reason, similar to other overlap-based data race detectors [5, 12, 18, 20], TxRace may miss data races if they happen far apart in time. Though perfect soundness is thus out of reach, the experimental results in Section 8 show that TxRace trades only a few false negatives (recall of 0.95) for excellent performance.

4.2 Handling Transactional Aborts

The best-effort Intel RTM does not guarantee that a transaction will eventually commit and make progress. In addition to data conflicts, there are many architectural and micro-architectural conditions that may cause a transaction to abort. When a transaction is aborted, Intel RTM rolls back the transaction to the point where it begins and reports the abort type(s) in the register. TxRace handles transaction aborts according to the abort reason as follows, while juggling the competing goals of reducing false negatives, decreasing overhead, and offering a forward progress guarantee.

Conflict. A transaction aborted due to a data conflict indicates a potential data race. To conduct precise data race detection, TxRace updates the shared flag (named *TxFail*) that every transaction begins by reading, forcing all the concurrent (in-flight) transactions to abort and roll back (Figure 3). TxRace then performs slow path software-based sound and complete data race detection among the code regions that overlapped with the aborted code region. Once a potential data race is detected by the fast path, the software-based slow-path detector will winnow out the false positives and to find data races if one exists.

Retry. A transaction aborted with "retry" status might succeed if retried. If this flag is set in conjunction with the conflict flag described above, TxRace treats the case as a conflict and follows the slow path. Otherwise, TxRace retries the transaction.

Capacity. When a transaction is aborted due to overflow, TxRace makes only the thread that observed the capacity abort fall back to slow path. Unlike the case for data conflicts, TxRace does not artificially abort the other concurrent transactions (by not updating the shared flag *TxFail*) since there is no indication of a potential data race with concurrent transactions. Using concurrent slow and fast path executions minimizes performance overhead while still giving TxRace high detection coverage (fewer false negatives) and the guarantee of forward progress. Figure 5 demonstrates how TxRace can detect data races when both fast and slow paths run at the same time. Here, threads T1 and T2 are on the fast path, while threads T3 and T4 are on the slow path due to capacity aborts. In this case, data races between threads T1 and T2 can be detected using HTM-based fast-path detection, and data races between thread T3 and T4 can be detected using precise slow-path detection. The interesting case is a race condition between fast path and slow path threads. If T2 is on the fast path and T3 is on the slow path as shown in the example, the strong isolation property of Intel RTM ensures that the transaction in T2 will be aborted in the event that T3 makes the conflicting access to the variable *X* that T2 has accessed. In this case, TxRace handles the conflict abort as described above. Because T3 is already in the slow path, the precise data race condition can be identified once TxRace puts T2 in the slow path.

Unknown. A transaction may abort with an unknown (unspecified) reason. As TxRace enforces that a transaction does not include a system call (Section 4.1), this is most

likely due to unexpected operating system context switches to handle interrupts, exceptions, etc. To guarantee forward progress while achieving high detection coverage (fewer false negatives), TxRace treats this case the same as the capacity abort.

Debug/Nested. The debug bit is set when a transaction aborts upon encountering a debug breakpoint, while the nested bit is set when a transaction was aborted during a nested transaction. Neither of these conditions may happen as a result of the TxRace transactionalization process; no debug breakpoints are used, and TxRace does not introduce nested transactions. TxRace simply ignores this case.

4.3 Optimization

To reduce performance overhead at runtime, TxRace applies several optimizations in the fast path. First, TxRace checks if the program is in the single-threaded mode or not (e.g., in the very beginning of program execution before spawning child threads). If so, there should be no races, thus TxRace simply does not use HTM to monitor the program execution to avoid unnecessary cost of transactions. If a function is profiled to be invoked in both single-threaded and multi-threaded modes, then at compile-time TxRace clones the function and instruments only the version that is called in multithreaded mode.

Second, Txrace reuses the same static analysis algorithm that Google TSan uses to avoid unnecessary data race checks. If a memory operation is statically proven to be data race free, then TSan does not instrument it. TxRace also does not insert transaction codes for those code regions that are not instrumented by Google TSan to hook memory accesses for data race detection.

Third, for regions containing a small number of memory operations, the overhead associated with HTMs exceeds the cost of the software-based slow path. If a code region contains fewer than K memory operations, TxRace favors the slow path. In our experiment we chose $K = 5$.

Finally, TxRace leverages our so-called *loop-cut* optimization for transactions that includes loops with a large number of iterations; these loops are a frequent cause of capacity aborts. By default, TxRace falls back to the slow path when a transaction experience a capacity abort to obtain better detection capability for those code regions at the cost of some performance overhead. To reduce this overhead, the loop-cut optimization aims to end the long transaction before prior to a capacity abort. TxRace first profiles an application with representative input to identify the candidate loops for the loop-cut optimization. In this study, TxRace leverages the Last Branch Recorder (LBR), a branch tracing facility in Intel processors [30], which allows TxRace to identify the last branch taken before a transaction aborts. Then, TxRace inserts the following loop-cut logic to the end of the candidate loop body.

The high level idea is for TxRace to keep track of the number of loop iteration (called *loop-cut-threshold*). When the transaction experiences a capacity abort in the loop, TxRace takes the slow path at first (default behavior), but when the same loop is executed next time, TxRace uses this loop-cut-threshold to cut the current transaction early in the middle of loop iterations, placing the rest of the iterations into another transaction to avoid capacity aborts. As discussed above, short transactions cut by loop-cut optimization may lead to false negatives.

HTM semantics make this implementation slightly tricky. Note that as the loop is a part of transaction, it is not possible to use a counter incremented per loop iteration to obtain the precise loop-cut-threshold value; updates to the counter will not survive a transaction abort. TxRace addresses this problem by setting a small initial estimate loop-cut-threshold (two in our experiment) and by incrementing and decrementing the estimate when the transaction commits/aborts, respectively (outside the transaction). This approach enables TxRace to estimate the last largest loop-cut-threshold allowing the transaction to commit. This work calls this scheme, which dynamically learns the loop-cut-threshold at runtime, *TxRace-DynLoopcut*.

As another scheme, *TxRace-ProfLoopcut* profiles an application with representative input to figure out the initial loop-cut-threshold value beforehand. This approach allows TxRace to avoid even the very first capacity abort. Similar to *TxRace-DynLoopcut*, *TxRace-ProfLoopcut* handles misprofiling by adjusting the threshold when the transaction commits or aborts accordingly. Section 8.2 evaluates the effectiveness of the two loop-cut optimization schemes.

5. Slow Path Software-based Race Detection

For software-based data race detection during the slow path, TxRace uses Google's ThreadSanitizer (TSan) [62, 65], an open-source state-of-practice data race detector. Similar to the well-known sound and complete FastTrack algorithm [21], TSan keeps track of the happens-before order for each memory location using a shadow memory. Then, TSan detects data races when accesses to shared locations are not ordered. This process requires instrumenting (1) synchronization operations to track the happens-before order; and (2) memory operations to look up shadow memory and compare their happens-before order. By design, TSan is complete (no false positive). However, to bound memory overhead, TSan maintains N (default 4) shadow cells per 8 application bytes, and replaces one random shadow cell when all shadow cells are filled. This may affect soundness (no false negative) of data race detection. Thus, this work configured TSan to have enough number of shadow cells to be sound as well.

The potential interplay of fast and slow path threads necessitates additional overhead during the fast path. A naive fast path could rely solely on HTM to detect potential data

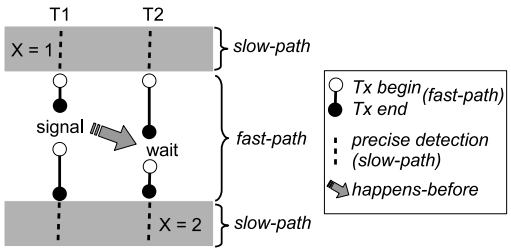

Figure 6: Tracking the happens-before order of synchronizations on the fast path eliminates false warnings on the slow path

races, obviating the need to track the happens-before order imposed by synchronization operations. However, threads may alternate between fast and slow paths for precise data race detection. When TSan finishes in the slow path, TxRace resumes the use of the fast path to monitor the next regions, achieving better performance. This design requires TxRace to keep track of the happens-before order of synchronization operations even during the fast path to remove false warnings during slow path. Figure 6 describes this feature in more detail. Suppose that threads T1 and T2 have executed the slow path, fast path, and slow path in turn for some reason other than conflicting accesses to *X*, and that there was a happens-before order between *signal* and *wait* that appeared during the fast path. If TxRace does not track this happens-before order during the fast path, the slow path data race detector would report a data race between *X=1* and *X=2*, which is a false warning. The performance overhead breakdown in Section 8.2 shows that tracking synchronization operations is not that expensive during the fast path.

6. False Negatives

TxRace is complete (no false positive) but unsound (some false negative). There are four main reasons why TxRace could miss data races. First, fast-path HTMs do not detect data conflicts between transactions that do not overlap in time, as discussed in Section 4.1. This is different from sound (happens-before based) data race detectors such as FastTrack or Google's TSan, which identify races by tracking the happens-before order of synchronization operations. In this sense, TxRace resembles overlap-based data race detectors [5, 12, 18, 20].

Second, when a transaction is aborted due to data conflict and TxRace writes the shared flag *TxFail* to abort others, there is no guarantee that some of the already-running transactions will not commit before they see the write. In this case, even though TxRace triggers the slow path, the race will not occur again and thus cannot be detected.

Third, race detection between the fast and slow paths (Figure 5) only works in one direction. If the slow path thread makes a shared memory access *before* the fast path thread makes a conflicting access, then the HTM's strong isolation guarantee does not apply. As a result, when the opposite of the situation in Figure 5 happen, TxRace will not trigger the slow path, and the race will not be detected.

Finally, TxRace by nature shares the limitations of the underlying HTM system. As of now, Intel Haswell processor does not support more concurrent transactions than the total number of hardware threads available. This implies that the number of threads that can be monitored by HTM during fast path is limited.

During evaluation, the thread count was restricted to be smaller than the hardware thread counts, ruling out the forth reason. All of the observed false negatives were due to non-overlapping transactions.

7. Implementation

TxRace instrumentation framework is implemented in the LLVM compiler framework [39]. As a very first process, we translate application source codes into LLVM IR (Intermediate Representation) and perform instrumentation as a custom transformation pass. During this process, we do not include external libraries such as standard *libc*, *libc++*, *libm*, etc., assuming that such libraries are thread safe, and users are interested in detecting data races in application codes. External libraries may be included into our scope when their LLVM IR is provided. It is worthwhile to note that we included all the internal libraries that are provided with core application codes such as *gsl*, *libjpeg*, *glib*, *libxml2*, etc. in PARSEC benchmark suite [3] into our analysis.

Instrumentation for fast path needs to intercept synchronization operations and the program points before/after system calls so that transaction begin/end codes can be inserted. For example, a new transactional region starts after a new thread starts or after each system call. As we do not include standard C/C++ libraries into our scope, we instrument system calls at the library call boundary; i.e., before and after calls to library functions that may invoke system calls such as synchronization (e.g., *PThread* library); standard I/O (e.g., *read*, *write*); and dynamic memory management library (e.g., *malloc*, *free*). For the third party libraries whose source codes are not available, dynamic binary instrumentation tools [10, 48, 55] can be used to profile the program with representative input and to identify a list of external library functions invoking system calls. Misprofiling would result in unknown aborts caused by undetected system calls. TxRace falls back to slow path in case of unknown aborts (by default), thus misprofiling only adds runtime overhead, and does not harm detection coverage.

For slow path, we use off-the-shelf Google's ThreadSanitizer (TSan) [62, 65], an open-source state-of-practice happens-before based data race detector. For each memory location and synchronization variable, TSan keeps track of happens-before order information into shadow memory. TSan supports compile-time instrumentation for data race detection using Clang frontend [68] and LLVM passes [39]. For simplicity, we instrument fast/slow path codes together

into the original program. For example, the same memory access hook is instrumented for both fast/slow paths. Depending on the fast or slow path, the hook performs TSan data race detection for slow path or it does nothing for fast path. For better performance, it would be ideal to clone the codes and have separate fast/slow path codes to remove the redundancy similar to [38, 67]. We leave this optimization as a future work.

The implementation of TxRace can be downloaded from https://github.com/lzto/TxRace

8. Evaluation

Our evaluation answers the following questions:

- What is the overhead of TxRace data race detection? Is it efficient?
- Does TxRace effectively detect data races? How many false negatives are there?
- Is TxRace cost-effective compared to other approaches? Is it better than a sampling-based approach, or a full happens-before based detector?

8.1 Methodology

We ran experiments on a 3.6GHz quad-core Intel Core™i7-4790 processor, with 16GB of RAM, running Gentoo Linux (kernel 4.0.4). Intel's Restricted Transactional Memory (RTM) is used for HTM-based fast path data race detection. Intel Haswell processor supports the same number of concurrent transactions as the hardware threads available, which is four (eight with hyperthreading) in our case. On the other hand, Google's ThreadSanitizer (TSan) is used for software-based slow path data race detection.

TxRace was evaluated using 1) PARSEC benchmark suite [3] that is designed to be representative of next-generation shared-memory programs including emerging workloads; and 2) Apache web server [1]. We used *sim-large* input for all the 13 application in PARSEC, and tested Apache using ab (ApacheBench) by sending 300,000 requests from 20 concurrent clients over a local network. Performance was reported in terms of overhead with respect to the original execution time without data race detection. We compare our system (named *TxRace* in the result) with off-the-shelf Google's ThreadSanitizer (named *TSan*). All results are the mean of five trials with four worker threads (except the scalability analysis).

8.2 Performance Overhead

Table 1 shows the TxRace execution statistic, the number of detected data races, and overall performance results. The first column provides the application name. The next four columns show transaction statistics during HTM-based fast path data race detection: the number of total committed transactions, the number of data conflict aborts, the number of capacity (overflow) aborts, and the number of unknown (unspecified) aborts. The next two columns give the number of races detected by TSan and TxRace. The applications in which TxRace cannot detect all the races reported by TSan are marked with asterisk. The next three columns show the original, TSan, and TxRace execution times. The first is the execution time of the original application (with no transactions, memory hooks, etc.); the second is the execution time when TSan is used; the third is the execution time of our system TxRace. The last two columns show TSan's and TxRace's overhead with respect to the original execution.

Examining the results, we see that TxRace's overhead is generally low. On average, TxRace reduces runtime overhead of dynamic data race detection from 11.68x to 4.65x (geometric means), showing 60% reduction ratio. For some applications such as *vips* and *streamcluster*, TxRace achieved more than 10 times speedup over TSan.

Figure 7 shows a breakdown of the overhead normalized to the original execution time (baseline) for all benchmarks. The black portion in each bar (xbegin/xend) represents the pure fast path overhead in which transactions are executed, but no slow path is taken even when they get aborted (simply run untransactionalized code). For most applications, this overhead is pretty low (the geometric mean of 17%), except *swaptions* and *streamcluster*. Upon further investigation, we found out that these two applications have tight loops that have system calls in the loop body. In this case, TxRace ends and begins new transactions around the system calls. This results in tight short transactions whose management cost now becomes dominant. The next overhead comes from handling aborts due to data conflicts (157%), which includes running slow path software-based data race detection. This low-overhead result shows the benefits of using HTM-based potential data race detection beforehand, which allows TxRace to selectively perform the software-based dynamic data race detection only for small fraction of execution intervals. Finally, the remaining performance overhead comes from handling capacity and unknown aborts (126% and 66%, respectively). To achieve small false negatives, TxRace takes the conservative approach of using slow path software-based detector to monitor program regions that fast path HTM cannot cover. We envision that if there is an ideal HTM such that a transaction aborts only if there is a data conflict and do not have capacity/unknown aborts, then the runtime overhead of TxRace would be improved significantly as TxRace only falls back to slow path when necessary (only when a data conflict occurs).

Figure 8 shows the scalability of TxRace. We varied the number of worker threads from 2, 4 to 8, and measured the runtime overhead normalized to the original execution time with 2, 4, and 8 worker threads, respectively. For comparison of two and four thread cases, some applications such as *swaptions* and *streamcluster* show lower runtime overhead for four thread cases, but most of the remaining applications show small differences in normalized runtime over-

application	committed transactions	conflict aborts	capacity aborts	unknown aborts	TSan races	TxRace races	original time(ms)	TSan time(ms)	TxRace time(ms)	TSan overhead	TxRace overhead
blackscholes	131105	2	0	7	0	0	253	467	460	1.85x	1.82x
fluidanimate	17778944	696789	10321	36614	1	1	539	8217	3724	15.23x	6.9x
swaptions	160640076	2599	557497	54317	0	0	868	5875	3446	6.77x	3.97x
freqmine	84	0	3	26	0	0	3973	55611	4569	14x	1.15x
vips	707547	16793	23403	14985	112	79(*)	953	1139087	60320	1195x	63.28x
raytrace	143	12	0	14	2	2	4546	23130	12203	5.09x	2.68x
ferret	208052	379	2413	4263	1	1	1060	11390	5852	10.74x	5.52x
x264	36808	245	423	5358	64	64	595	3837	3332	6.45x	5.6x
bodytrack	9950991	36004	47050	2004723	8	6(*)	503	6429	4479	12.78x	8.9x
facesim	12827334	1611	3372	38563	9	8(*)	2439	89242	28027	36.59x	11.49x
streamcluster	756908	170805	230	832	4	4	1430	39042	4253	25.9x	2.97x
dedup	2185219	106618	13889	40177	0	0	2748	13292	11513	4.84x	4.19x
canneal	3200570	25187	2896	106419	1	1	3499	15367	10375	4.39x	2.97x
apache	310781	227	446	9793	0	0	6916	21089	13600	3.05x	1.97x
geo.mean										11.68x	4.65x

Table 1: TxRace Execution Statistics and Performance.

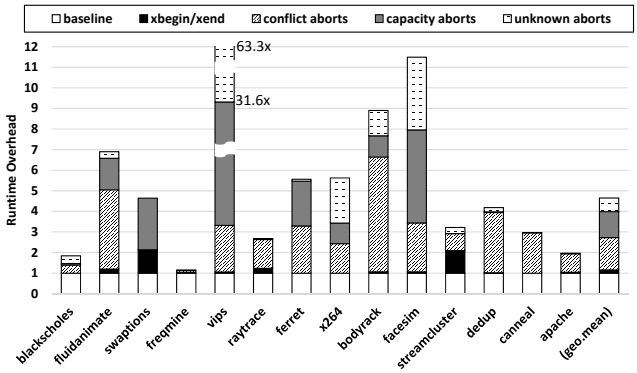

Figure 7: Breakdown of runtime overhead.

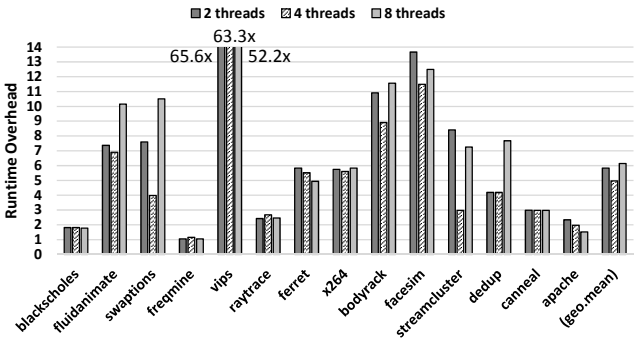

Figure 8: Scalability of TxRace

head. Upon further investigation, we found out that the number of capacity aborts (geometric mean of 644 vs. 474) and unknown aborts (geometric mean of 8377 vs. 4651) decreases from two to four thread cases. The reduction in capacity aborts makes sense as many applications in PARSEC benchmark take advantage of data parallelism, and thus each worker thread is likely to have smaller dataset with more worker threads. On the other hand, the number of conflict aborts (geometric mean of 244 vs. 1242) increases from two to four thread cases (likely due to increased concurrency). After all, the mixture of the increase and the decrease in transactional aborts causes different applications to show variation in performance overhead.

Another interesting result of this experiment is the high overhead incurred for some applications with eight threads (e.g., *fluidanimate*, *swaptions*, *streamcluster*, and *dedup*). Examining the results, we found that the number of unknown aborts increases significantly for eight thread case (geometric mean of 42251, which is 5x and 9x more than two and four thread cases, respectively). As Intel Haswell processor does not provide additional information regarding unknown aborts, further investigation was not possible, but we suspect that eight concurrent transactions enabled by hyperthreading might lead to increased unkonwn aborts.

Finally, we evaluate the effectiveness of the loop-cut optimization discussed in the Section 4.3. Figure 9 presents the normalized runtime overhead of TSan and three different types of TxRace. They differ from each other based on how they handle a transaction that includes a loop with a large number of iterations, causing capacity aborts frequently. *TxRace-NoOpt* stands for the basic scheme without optimization that TxRace simply falls back to slow path every time when a transaction gets aborted for the capacity reason. *TxRace-DynLoopcut* represents the optimized scheme that for a transaction including a loop, TxRace dynamically learns the loop iteration count (called *loop-cut-threshold*) that do not cause a capacity abort at runtime. When the transaction gets aborted, TxRace falls back to the slow path at first. However, when the same loop is executed next time, TxRace uses the loop-cut-threshold to terminate the transaction early in the middle of loop iterations and starts a new transaction to avoid capacity aborts. *TxRace-ProfLoopcut* is similar to the above dynamic scheme, but it profiles the program with representative input to collect the initial loop-cut-threshold, and avoids even the very first capacity aborts. Fig-

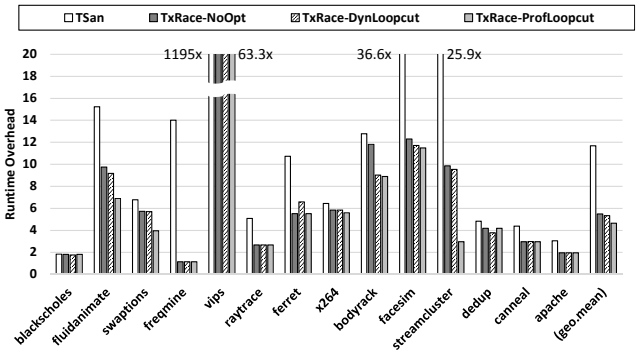

Figure 9: Effectiveness of loop-cut optimization

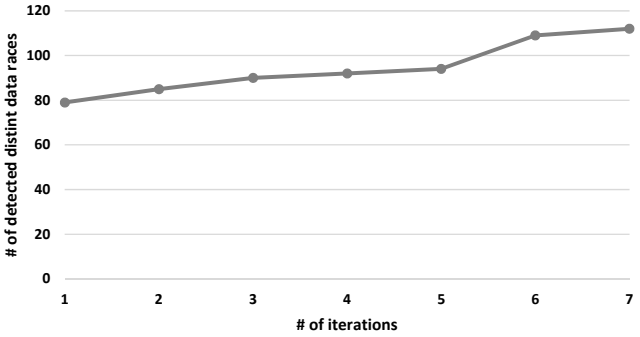

Figure 10: The number of detected distinct data races across multiple runs for *vips*

ure 9 shows the benefits of leveraging the loop-cut optimization. In all cases, TxRace is more efficient than TSan. On average, *TxRace-ProfLoopcut* shows the best result (4.65x) in terms of performance overhead, and *TxRace-DynLoopcut* which does not require profiling the threshold also performs better than TSan (5.34x).

8.3 False Negatives

In this section, we study false negatives of TxRace. HTMs detect data conflicts between transactions that are concurrently overlapped in time. As a result, similar to other overlap-based data race detectors [5, 12, 18, 20], TxRace may miss data races if they happen far apart in time. The sixth and seventh columns of Table 1 represent the average number of data races reported by happens-before based TSan and our overlap based TxRace. Here, each race is in a form of racy instruction pair, and we count the number of static instances.

There are three applications (*vips*, *bodytrack*, and *facesim*) that TxRace detects less data races than TSan. It turns out that the missed three cases of *bodytrack* and *facesim* are due to the common *initialization* idiom, in which a data structure is allocated within a thread and initialized without any synchronization while the structure is still local to the thread, and then it becomes accessible to other threads, by adding it to a global data structure. For example in *facesim*, a structure is initialized when a thread pool is created at the beginning of program execution, then it becomes shared at a later time. TxRace missed such races because conflicting accesses do not overlap.

On the other hand, for *vips*, though the number of data race found for each test run remains about the same (average of 79), we observed that TxRace actually finds different sets of data races across different runs. This makes sense because TxRace's nature of the overlap-based detection makes it sensitive to underlying OS scheduler. Figure 10 shows that when we accumulate the distinct data races detected, TxRace can find all the data races (112) found by TSan after seven runs. Note that for *vips*, TxRace (63.3x) is order of magnitude faster than TSan (1195x).

8.4 Cost-Effectiveness of Data Race Detection

TxRace is complete (no false positive) but unsound (some false negative). In essence, TxRace aims to be a cost-effective solution that exploits a critical tradeoff of soundness for performance. To quantitatively evaluate how cost-effective TxRace is, we rely on a popular economic analysis term called *cost-effectiveness ratio* where the denominator is the effectiveness and the numerator is the cost. The original ratio is inverted and redefined for the context of data race detection to quantify how cost-effective it is as follows:

$$CostEffectiveness = \frac{Race\_Detection\_Effectiveness}{Race\_Detection\_Cost}$$

As a metric to evaluate the data race detection effectiveness, we use *recall* that is commonly used to measure the quality of classifiers in information retrieval and bug detection communities [4, 36, 43, 71]. Intuitively, high recall leads to less false negatives (undetected data races). In the context of data race detection, *recall* is defined as follows:

$$recall = \frac{|Reported\_Data\_Races \cap Real\_Data\_Races|}{|Real\_Data\_Races|}$$

For comparison to TSan, $Real\_Data\_Races$ is defined as the data races reported by TSan. To calculate the cost effectiveness (CE), we use TxRace's runtime overhead normalized to TSan's. Table 2 summarizes how much more cost-effective TxRace is compared to TSan for each benchmark application (here TSan's CE is 1). TxRace turns out to be 2.38x (geometric mean) more cost-effective than TSan across the benchmark applications. This is mainly because in TxRace, only small portion of memory accesses are investigated for software-based data race detection (slow path). On the other hand, the majority of memory accesses are dealt with by transactional execution (fast path) at a very low runtime cost.

To justify such a high cost-effectiveness of TxRace, we also compare TxRace with TSan with sampling. Sampling memory operations is an intuitive way to reduce runtime overhead of dynamic data race detection. However, it also comes with false negative issues because some data races

application	overhead	recall	cost-effectiveness
blackscholes	0.99	1	1.02
fluidanimate	0.45	1	2.21
swaptions	0.59	1	1.7
freqmine	0.08	1	12.17
vips	0.05	0.71	13.32
raytrace	0.53	1	1.9
ferret	0.51	1	1.95
x264	0.87	1	1.15
bodytrack	0.7	0.75	1.08
facesim	0.31	0.89	2.83
streamcluster	0.11	1	8.71
dedup	0.87	1	1.15
canneal	0.68	1	1.48
apache	0.65	1	1.55
geo.mean	0.38	0.95	2.38

Table 2: Cost-Effectiveness of TxRace vs. TSan

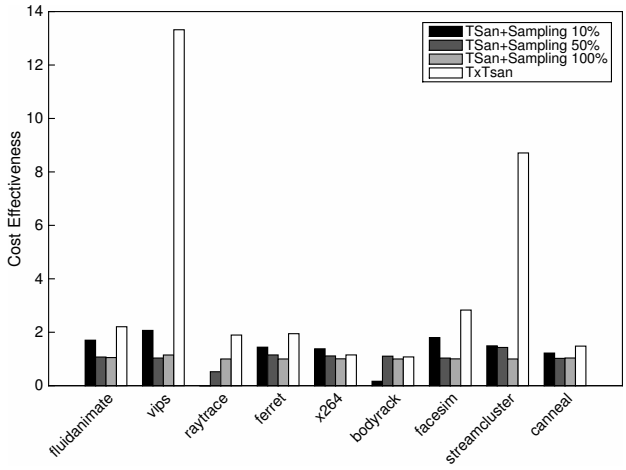

Figure 11: Cost-Effectiveness of TxRace vs. Sampling

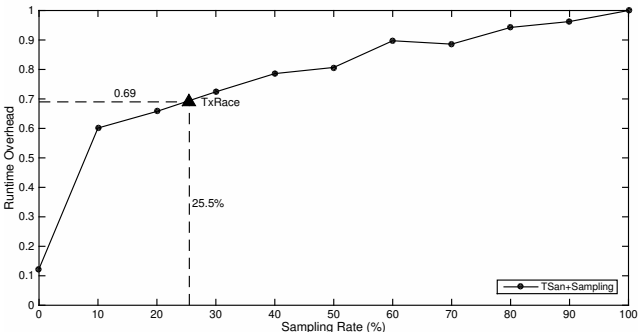

Figure 12: Runtime overhead for *bodytrack*

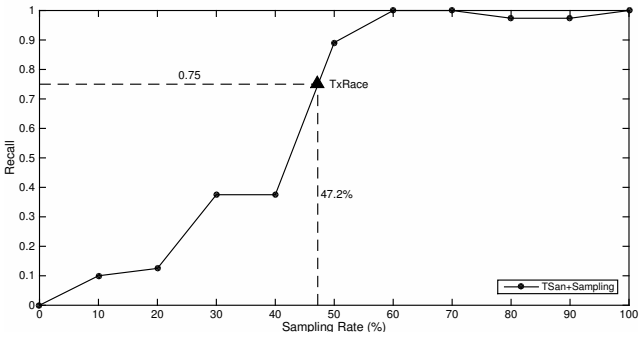

Figure 13: Recall for *bodytrack*

might be missed at a low sampling rate. To study if TxRace is more cost-effective than sampling, we vary the sampling rate and measure the resulting runtime overhead and the recall of TSan for every benchmark. As a represetative application, we present the result of *bodytrack* in detail. Figure 12 shows the runtime overhead normalized to 100% sampling (full coverage), and Figure 13 shows the recall at different sampling rates while treating 100% as an oracle. As expected, both the runtime overhead and recall increase as the sampling rate increases. On the other hand, the normalized runtime overhead and the recall of TxRace are 0.69 and 0.75, respectively. This implies that TxRace adds an overhead equivalent to sampling about 25.5% of memory operations, but its recall is equivalent to 47.2% sampling, which shows its cost effectiveness.

Lastly, Figure 11 presents the cost-effectiveness of TxRace compared to TSan with sampling across nine applications in which TxRace/TSan detect at least one data race. For some applications such as *fluidanimate*, *vips*, and *facesim*, 10% sampling turns out to be more efficient than 50% or 100% sampling cases. It turns out that the data race in such applications manifest often at runtime, thus they get detected even at the low sampling frequency. In other words, the number of dynamic instances of the data race is quite high even though the number of static instance (unique race condition) is small (e.g., one for *fluidanimate*), thus reaching the recall of almost 1 (no false negative) at the low frequency. After all, for almost all applications execpt *x264*, TxRace outperforms TSan with sampling in terms of the cost-effectiveness.

9. Related Work

This section discusses closely related work comparing TxRace with existing data race detection techniques. To the best of our knowledge, TxRace is the first approach to leverage a commodity HTM for speeding up data race detection.

Eraser [63] introduced lockset-based approach, which infers data race through violation of locking discipline. As lockset-based algorithms do not consider non-mutex synchronization operations such as conditional variables, they are incomplete (generate false positives) compared to the approach that tracks happens-before order of synchronization operations using vector clocks. FastTrack [21] is known to be the most optimized algorithm in this category, which reduced runtime overhead significantly compared to prior works such as MultiRace [58]. Google's ThreadSanitizer, which TxRace used for its slow path, also tracks happens-before order similar to FastTrack. However, high runtime overhead is still a major concern.

Several strategies have been explored to reduce the overhead of dynamic data race detection. LiteRace [50] and Pacer [9] use sampling; RaceMob [37] crowdsources (distributes) runtime checks across different users; Wester et al. [70] parallelizes data race detection; Lee et al. [40, 41] uses offline symbolic analysis; and Goldilocks [19], Choi et al. [13], and Chimera [42] leverage static analysis to remove checks for statically proven race-free memory operations. Matar et al. [52] exploit Intel TSX (same as ours) to speed up data race detection, but they leveraged HTM simply to replace locks that are used to provide atomicity in metadata updates/checks, which is different from TxRace. We believe that TxRace can use these techniques to further reduce the runtime cost of the slow path.

Greathouse et al. [24] presents a demand-driven race detector. They use hardware performance counters in modern processors to monitor cache events indicating data sharing to turn on race detection. Due to limitation in current hardware, they could identify W→R data sharing events only, and though all are presumably possible, not all cache sharing causes data races. On the other hand, TxRace uses data conflict detection mechanism in HTM to identify potential data races and to trigger on-demand race detection.

The recent advances in overlap-based data race detectors [5, 12, 18, 20] have shown their cost-effectiveness and practical benefits by trading soundness for better performance. DataCollider [20] and Kivati [12] use hardware code/data breakpoint mechanism in processors to detect data races. They set a data breakpoint to trap conflicting accesses by other threads. This is similar to conflict detection mechanism in HTM, which detects concurrent conflicting accesses. After setting up the breakpoint, they insert a short amount of time delay to the thread to increase detection probability. The detection window in HTM spans the whole length of a transaction, so the detection probability is likely to be higher than the breakpoint based approach at the cost of false positive. Moreover, the small number of breakpoints (four for x86 hardware) limits its coverage. As another overlap-based detector, IFRit [18] exploits compiler analysis to form interference-free regions where data races can be detected when they overlap. The scope of IFRit may be longer than TxRace's transactional region in some cases, but it could be very short (e.g., basic block) in other cases since each region may start after the variable is defined in the SSA form. For performance reasons, IFRit gives up data race detection for those short-scope regions, which TxRace may cover.

Data race detection also has been the subject of intense research by hardware community. In general, hardware-assisted data-race detectors store metadata (e.g., locksets, vector-clocks) in the cache, piggyback them on coherence protocol messages, and compare them to detect data races.

HARD [76] is a hardware-based implementation of the lockset algorithm, whereas ReEnact [60] and CORD [59] implement happens-before based algorithm in hardware. RADISH [16] proposes hardware-software co-design to enable always-on sound and complete data race detections in which hardware performs the vast majority of race checks and software backs up hardware resource limitations. Conflict Exceptions [47] extends a standard coherence protocol and caches to detect data conflicts between synchronization-free regions, as TxRace does. SigRace [54] employs custom hardware address signature where the memory addresses accessed by a processor are hash-encoded and accumulated, and uses it do detect the outcome of potential data races rather than the race itself. Then SigRace relies on checkpointing/rollback to identify actual racing instructions. Similarly, TxRace uses the same insight to use lightweight check first, then do expensive one later only if necessary.

Perhaps RaceTM [25] is the most closely related to our work, as it also proposes to use hardware transactional memory in detecting data races. However, their approach is hardware-only solution that requires additional hardware extension to conventional HTMs like LogTM [53]. For example, RaceTM requires two additional bits (debug read/write bits) for every cache line. They added support to provide the conflict address and responsible racy instructions as well.

Finally, even if TxIntro [45] is not a data race detector, it combines hardware performance counters (such as HITM, cache miss, and PEBS) with Intel's TSX to infer the conflicting data linear address. We envision that TxRace can leverage such supports to design more efficient slow path if the future generation of commodity processor provides such information to the users.

10. Conclusion

The spread of shared-memory multiprocessor architectures has spurred development of multithreaded programs. However, such programs are subject to concurrency bugs including data races. Unfortunately, traditional dynamic data race detectors are too slow to use in many cases. This paper describes TxRace, a new software dynamic data race detector that exploits commodity hardware transactional memory support to enable dynamic data race detection with a low runtime overhead. Leveraging existing HTM support allows TxRace to use a precise but expensive dynamic data race detector only for a small fraction of the whole execution in an on-demand manner, leading to performance improvement. The experiment results show that TxRace achieves runtime overhead reduction of dynamic data race detection by 60% on average with only a few false negatives (high recall of 0.95).

Acknowledgments

We thank Jaeheon Yi, Jungwoo Ha, and Richard M. Yoo for helpful discussion and their feedback; James Davis for his careful proofreading; and the anonymous reviewers for insightful comments. The work is supported in part by an award from Google.

References

[1] The apache http server. http://httpd.apache.org.

[2] Y. Afek, A. Levy, and A. Morrison. Software-improved hardware lock elision. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 212–221, 2014. ISBN 978-1-4503-2944-6.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. of the 17th PACT*, Oct. 2008.

[4] C. Bishop et al. *Pattern recognition and machine learning*. Springer New York:, 2006.

[5] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Efficient, software-only data race exceptions. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '15, 2015.

[6] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, software-only region conflict exceptions. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 241–259, 2015. ISBN 978-1-4503-3689-5.

[7] 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 *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 81–91, 2007. ISBN 978-1-59593-706-3.

[8] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, 2008.

[9] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 255–268, 2010. ISBN 978-1-4503-0019-3.

[10] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, 2003. ISBN 0-7695-1913-X.

[11] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: A hybrid transactional memory for haswell's restricted transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 187–200, 2014.

[12] L. Chew and D. Lie. Kivati: Fast detection and prevention of atomicity violations. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 307–320, 2010. ISBN 978-1-60558-577-2.

[13] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 258–269, 2002. ISBN 1-58113-463-0.

[14] C. Click. Azuls experiences with hardware transactional memory. In *In HP Labs - Bay Area Workshop on Transactional Memory*, 2009.

[15] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 39–52, 2011. ISBN 978-1-4503-0266-1.

[16] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. Radish: Always-on sound and complete ra detection in software and hardware. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 201–212, 2012. ISBN 978-1-4503-1642-2.

[17] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 157–168, 2009. ISBN 978-1-60558-406-5.

[18] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. Ifrit: Interference-free regions for dynamic data-race detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 467–484, 2012. ISBN 978-1-4503-1561-6.

[19] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 245–255, 2007. ISBN 978-1-59593-633-2.

[20] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *In Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI '10, 2010.

[21] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, 2009. ISBN 978-1-60558-392-1.

[22] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 293–303, 2008. ISBN 978-1-59593-860-2.

[23] B. Goel, R. Titos-Gil, A. Negi, S. A. McKee, and P. Stenstrom. Performance and energy analysis of the restricted transactional memory implementation on haswell. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 615–624, Washington, DC, USA, 2014. ISBN 978-1-4799-3800-1.

[24] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin. Demand-driven software race detection using hardware performance counters. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 165–176, 2011. ISBN 978-1-4503-0472-6.

[25] S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Roetteler. Racetm: Detecting data races using transactional memory. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 104–106, 2008. ISBN 978-1-59593-973-9.

[26] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim. The ibm blue gene/q compute chip. *IEEE Micro*, 32 (2):48–60, Mar. 2012. ISSN 0272-1732.

[27] W. Hasenplaugh, A. Nguyen, and N. Shavit. Quantifying the capacity limitations of hardware transactional memory. In *WTTM '15: 7th Workshop on the Theory of Transactional Memory*, July 2015.

[28] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, 1993. ISBN 0-8186-3810-9.

[29] Intel. Intel architecture instruction set extensions programming reference. chapter 8: Intel transactional synchronization extensions, 2012. https://software.intel.com/sites/default/files/m/9/2/3/41604.

[30] Intel. Intel 64 and ia-32 architectures software developers manual, 2013. http://download.intel.com/products/processor/manual/325462.pdf.

[31] Intel. Intel 64 and ia-32 architectures optimization reference manual. chapter 12: Intel tsx recommendations, 2014. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[32] Intel. Intel inspector xe, 2015. http://software.intel.com/en-us/intel-inspector-xe.

[33] International Organization for Standardization. ISO/IEC 14882:2011: Information technology – Programming languages – C++, 2011.

[34] International Organization for Standardization. ISO/IEC 9899:2011: Information technology – Programming languages – C, 2011.

[35] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for ibm system z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 25–36, 2012. ISBN 978-0-7695-4924-8.

[36] C. Jung, S. Lee, E. Raman, and S. Pande. Automated memory leak detection for production use. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[37] B. Kasikci, C. Zamfir, and G. Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 406–422, 2013. ISBN 978-1-4503-2388-8.

[38] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast track: A software system for speculative program optimization. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 157–168, 2009. ISBN 978-0-7695-3576-0.

[39] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, 2004. ISBN 0-7695-2102-9.

[40] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira. Offline symbolic analysis for multi-processor execution replay. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 564–575, 2009. ISBN 978-1-60558-798-1.

[41] D. Lee, M. Said, S. Narayanasamy, and Z. Yang. Offline symbolic analysis to infer total store order. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 357–358, 2011. ISBN 978-1-4244-9432-3.

[42] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid program analysis for determinism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 463–474, 2012. ISBN 978-1-4503-1205-9.

[43] S. Lee, C. Jung, and S. Pande. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[44] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, aug 2007.

[45] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA '14, 2014.

[46] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, 2008. ISBN 978-1-59593-958-6.

[47] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 210–221, 2010. ISBN 978-1-4503-0053-7.

[48] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005. ISBN 1-59593-056-6.

[49] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, 2005. ISBN 1-58113-830-X.

[50] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Pro-*

ceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, pages 134–143, 2009. ISBN 978-1-60558-392-1.

[51] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17–17, July 2006. ISSN 1556-6056.

[52] H. S. Matar, I. Kuru, S. Tasiran, and R. Dementiev. Accelerating precise race detection using commercially available hardware transactional memory support. In *5th Workshop on Determinism and Correctness in Parallel Programming*, WoDet '14, 2014.

[53] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 2006 IEEE 12th International Symposium on High Performance Computer Architecture*, pages 254–265, Feb. 2006.

[54] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. Sigrace: Signature-based data race detection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 337–348, 2009.

[55] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, 2007. ISBN 978-1-59593-633-2.

[56] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, Mar. 1992. ISSN 1057-4514.

[57] PCWorld. Nasdaq's facebook glitch came from race conditions, May 2012. http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html.

[58] E. Pozniansky and A. Schuster. Multirace: Efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, Mar. 2007. ISSN 1532-0626.

[59] M. Prvulovic. Cord: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *Proceedings of the 2006 IEEE 12th International Symposium on High Performance Computer Architecture*, HPCA '06, 2006.

[60] M. Prvulovic and J. Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 110–121, 2003. ISBN 0-7695-1945-8.

[61] C. G. Ritson and F. R. Barnes. An evaluation of intel's restricted transactional memory for cpas, 2013.

[62] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 34–41, 2006. ISBN 1-59593-576-2.

[63] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997. ISSN 0734-2071.

[64] SecurityFocus. Software bug contributed to blackout, Feb. 2004. http://www.securityfocus.com/news/8016.

[65] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, 2009. ISBN 978-1-60558-793-6.

[66] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 387–400, 2012. ISBN 978-1-4503-1083-3.

[67] M. Susskraut, T. Knauth, S. Weigert, U. Schiffel, M. Meinhold, and C. Fetzer. Prospect: A compiler framework for speculative parallelization. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 131–140, 2010.

[68] T. C. Team. Clang 3.8 threadsanitizer, 2015. http://clang.llvm.org/docs/ThreadSanitizer.html.

[69] J. Ševčík and D. Aspinall. On validity of program transformations in the java memory model. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 27–51, 2008. ISBN 978-3-540-70591-8.

[70] B. Wester, D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy. Parallelizing data race detection. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 27–38, 2013. ISBN 978-1-4503-1870-9.

[71] R. Wilcox. *Introduction to Robust Estimation and Hypothesis Testing*. Elsevier Science & Technology, 2012. ISBN 9780123869838.

[72] Y. Xie, M. Naik, B. Hackett, and A. Aiken. Soundness and its role in bug detection systems. In *In Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[73] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In *The 4th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2012. USENIX. URL `https://www.usenix.org/conference/hotpar12/concurrency-attacks`.

[74] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, 2013. ISBN 978-1-4503-2378-9.

[75] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 221–234, 2005. ISBN 1-59593-079-5.

[76] P. Zhou, R. Teodorescu, and Y. Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 121–132, 2007. ISBN 1-4244-0804-0.