TRADE: Precise Dynamic Race Detection for Scalable Transactional Memory Systems

GOKCEN KESTOR, Pacific Northwest National Laboratory
OSMAN S. UNSAL and ADRIAN CRISTAL, Barcelona Supercomputing Center
SERDAR TASIRAN, Koc University

As other multithreaded programs, transactional memory (TM) programs are prone to race conditions. Previous work focuses on extending existing definitions of data race for lock-based applications to TM applications, which requires all transactions to be totally ordered "as if" serialized by a global lock. This approach poses implementation constraints on the STM that severely limits TM applications' performance.

This article shows that forcing total ordering among all running transactions, while sufficient, is not necessary. We introduce an alternative data race definition, *relaxed transactional data race*, that requires ordering of *only* conflicting transactions. The advantages of our relaxed definition are twofold: First, unlike the previous definition, this definition can be applied to a wide range of TMs, including those that do not enforce transaction total ordering. Second, within a single execution, it exposes a higher number of data races, which considerably reduces debugging time. Based on this definition, we propose a novel and precise race detection tool for C/C++ TM applications (TRADE), which detects data races by tracking happens-before edges among conflicting transactions.

Our experiments reveal that TRADE precisely detects data races for STAMP applications running on modern STMs with overhead comparable to state-of-the-art race detectors for lock-based applications. Our experiments also show that in a single run, TRADE identifies several races not discovered by 10 separate runs of a race detection tool based on the previous data race definition.

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

General Terms: Languages, Verification

Additional Key Words and Phrases: Transactional memory, correctness, race detection, debugging

ACM Reference Format:
Gokcen Kestor, Osman S. Unsal, Adrian Cristal, and Serdar Tasiran. 2015. TRADE: Precise dynamic race detection for scalable transactional memory systems. ACM Trans. Parallel Comput. 2, 2, Article 11 (July 2015), 23 pages.
DOI: http://dx.doi.org/10.1145/2786021

1. INTRODUCTION

With the widespread use of multicore hardware, parallel programming has become prevalent. Transactional memory (TM) is one of the programming models that addresses ease of programmability of parallel programs while keeping up with performance expectations of multicore hardware. With TM, programmers mark sections of code they intend to execute atomically and need not worry about deadlocks. As with other multithreaded programs, TM programs are also prone to race conditions.

Authors' addresses: G. Kestor; email: gokcen.kestor@pnnl.gov; O. S. Unsal and A. Cristal; emails: {osman.unsal, adrian.cristal}@bsc.es; S. Tasiran; email: stasiran@ku.edu.tr.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2015 ACM 2329-4949/2015/07-ART11 $15.00
DOI: http://dx.doi.org/10.1145/2786021

A race condition occurs when two threads access the same shared variable, at least one of the accesses is a write, and at least one thread does not use the proper synchronization primitive. For TM programs, this means that a data race may exist between transactional and nontransactional accesses to shared data or between multiple nontransactional accesses to shared data.

While there has been significant efforts to develop hardware (HTM) [Dice et al. 2009; Haring 2011] and software (STM) [Dalessandro et al. 2010; Dice et al. 2006; Riegel et al. 2007; Kestor et al. 2011] TM systems, compilers with TM support [Christie et al. 2010; compiler support in gcc 4.7 2010; Intel Corporation 2009], and basic debuggers for TM [Herlihy and Lev 2009; Zyulkyarov et al. 2010], there is still a lack of integrated development environments and software debugging tools to detect data races in TM programs. The fundamental problem with the development of such tools is that there is no consensus on a single definition of what it means for a TM program to have a race condition. Researchers have suggested that data races for TM programs should be defined in terms of locking semantics [Boehm 2009], and that TM should be considered a technique for implementing the semantics of a single global lock, but allowing greater concurrency than would be implied by actually having a single lock [Menon et al. 2008]. This basic, pragmatic semantics is called *single global lock atomicity* (SGLA), where a program is required to behave "as if" transactions were protected by a single global lock. More precisely, a TM system is said to provide SGLA if, for every program execution, there exists some global total order on all transactions that is consistent with program order. Given that an underlying STM implements SGLA, previous works have extended the definition of data race based on a happens-before relation for lock-based applications to TM applications [Dalessandro and Scott 2009; Grossman et al. 2006; Menon et al. 2008]. In this article, we call this happens-before relation the *strict happens-before relation*.

Although SGLA simplifies the design, implementation, and testing of STM systems, many important scalable STMs do not implement SGLA semantics. The reason is that forcing a total order among *all* transactions may reduce the concurrency level, which is not desirable from a performance point of view. Performance (scalability in particular) is of paramount importance for modern multicore systems; hence, programmers have been reluctant to use STM designs that implement SGLA semantics. For example, TL2 [Dice et al. 2006], TinySTM [Riegel et al. 2007], and NORec [Dalessandro et al. 2010] are commonly used STMs but do not implement SGLA semantics. Hence, the applicability of dynamic race detection tools based on the strict happens-before relation remains limited. Another problem with such race detection tools is that the probability of detecting a data race within a single execution may be low due to the unnecessary strict happens-before orders established among transactions. From a practical point of view, this means that the programmer is required to run the race detection tool several times on the same program to cover a large number of thread interleavings. This, obviously, increases the application debugging time. More importantly, the exact number of iterations is usually unbounded and unknown and grows with the complexity of a program due to the larger number of possible thread interleavings, which makes race detection tools based on the strict happens-before relation limited for real applications.

This article shows that, while total ordering on all transactions is a sufficient condition, it is overrestrictive and unnecessary. We thus introduce an alternative happens-before relation, the *relaxed happens-before relation*, that orders *only* conflicting transactions. We denote two transactions as *conflicting* if there exists a memory location x where one transaction writes and the other either reads or writes. If no such conflicting access exists, there is no need to order those transactions (*nonconflicting*) with respect to each other. We show that this ordering is still sufficient to maintain

soundness. Based on this definition, we also propose Transactional data RAce DEtection (TRADE), a novel and precise race detection algorithm for TM applications, which computes a precise representation of the relaxed happens-before relation among conflicting transactions for the observed execution and reports an error if and only if the observed execution has a data race. Within a single execution, TRADE has a higher probability of observing a data race compared to a tool based on the strict happens-before relation.

For comparison reasons, we implement a race detection tool based on the strict happens-before relation (s-TRADE). This tool requires the underlying STM to implement SGLA [Menon et al. 2008] and cannot be used with an STM that does not enforce total ordering among all transactions, such as TinySTM. We analyze the precision and performance of TRADE on a eight-core Intel Nehalem system with STAMP applications, a benchmark suite commonly used to test TM systems [Minh et al. 2008]. TRADE detects data races in one of the STAMP applications. To further analyze the effectiveness of our tool, we inject synthetic data races into STAMP applications: our experiments demonstrate that a single run of TRADE discovers more races than 10 independent runs of s-TRADE. Moreover, we show that TRADE runtime overhead with respect to s-TRADE is generally negligible despite the fact that TRADE requires more work than s-TRADE to establish happens-before edges among only conflicting transactions. More importantly, TRADE can be used with scalable STM systems that provide higher performance compared to STMs that implement SGLA.

This article makes the following contributions:

—We propose TRADE, a novel race detection algorithm. To the best of our knowledge, TRADE is the first race detection algorithm based on the happens-before relation for real TM C/C++ applications.
—Based on our algorithm, we implement the corresponding dynamic race detection tool for real C/C++ TM applications.
—We analyze STAMP applications and discover data races for *SSCA2*.

This article is organized as follows: Section 2 reviews preliminary concepts. Sections 3 and 4 detail the TRADE race detection algorithm and implementation, respectively. Section 5 evaluates TRADE in terms of precision and performance and compares it with s-TRADE. Section 6 presents related work. Section 7 concludes this work.

2. PRELIMINARIES

This section first reviews the previous definition of a data race based on a strict transactional happens-before relation and then our proposed definition. In this work, we consider applications that only use transactions to synchronize accesses to shared memory. While other synchronization mechanisms (e.g., locks) could be used in conjunction with transactions, we are not aware of any publicly available applications programmed with transactions and locks that we can use in our experiments. We leave this for future work.

We represent an execution of a multithreaded program as a sequence of actions, such as transactional/nontransactional read/write operations, begin/end transactions, fork/join, and barriers, and we assume the following:

—Accesses from an individual thread are ordered by *program order*.
—Synchronization primitives, such as `barrier()`, `fork()`, and `join()`, introduce a *sync-primitive order*. An access a performed before a synchronization primitive is ordered before an access b performed after this synchronization primitive.

2.1. Strict Transactional Data Race

Although locks and transactions are semantically different, previous work focused on extending existing definitions of data race for lock-based applications to TM applications. Researchers have investigated forms of *single global lock atomicity* (SGLA) that map correct synchronization of TM programs into existing lock-based disciplines [Dalessandro and Scott 2009; Menon et al. 2008; Grossman et al. 2006]. In STM systems that implement SGLA semantics, transactions are considered "as if" they were executed under a single global lock. Menon et al. explore the implementation and performance implications of SGLA and the merits of various definitions for which pairs of transactions are ordered [Menon et al. 2008]. They show that a sufficient condition for SGLA is to allow concurrent execution of transactions but to linearize their execution at the commit phase in a staggered, pipelined fashion. This staggered execution of transactions provides an explicit total ordering over all transactions.

In the following, we define strict transactional data races for TM applications according to the *transactional data race free* (TDRF) model [Menon et al. 2008; Dalessandro and Scott 2009; Grossman et al. 2006]:

Definition 1. All transactions in the execution are totally ordered by a *strict transaction order* ($\longrightarrow_{sto}$). If transaction A is transactionally ordered before B, then access a in A is transactionally ordered before access b in B.

With the strict transaction order, two transactions are allowed to run in parallel but they need to commit in linear order, even if their read- and write-sets do not overlap.

Definition 2. The irreflexive transitive closure of program order, sync-primitive order, and strict transaction order defines a *strict happens-before* ($\longrightarrow_{shb}$) partial ordering on all accesses in the execution.

Definition 3. A *strict transactional data race* exists between two accesses in a given execution if and only if they access the same location, at least one is a write, they are executed by different threads, and they are not ordered by $\longrightarrow_{shb}$.

Definition 4. A program is *transactional data race free* (TDRF) if and only if no strict transactional data races exist in any execution of the program.

TDRF is a direct extension of the conventional models used for lock-based applications, for which a total ordering of operations on a lock follows naturally from their semantics, and correctness conditions can be defined based on *data race freedom* [Adve and Hill 1998]. Note that TDRF cannot be used as a correctness model if the underlying STM system does not enforce any form of transactional total ordering among *all* running transactions in the system such as TL2 [Dice et al. 2006], TinySTM [Riegel et al. 2007], and NORec [Dalessandro et al. 2010].

2.2. Relaxed Transactional Data Race

SGLA limits the nature of TM by ordering nonconflicting transactions. The result is that STMs that implement SGLA provide much lower performance than STMs that provide higher concurrency. Figure 1 shows the speedup of several popular STMs over the sequential version. The experiments are conducted with STAMP [Minh et al. 2008] applications running with eight threads on an eight-core Intel Nehalem system. We use STMs from different research groups: TL2 [Dice et al. 2006] is a lazy conflict detection, write-buffered design; TinySTM [Riegel et al. 2007] is an eager conflict detection, write-buffered design; NORec [Dalessandro et al. 2010] is a lazy conflict detection, write-buffered STM based on a single sequential lock; and finally, Pipeline [software transactional memory runtime 2010] is an STM that features a lazy conflict detection,

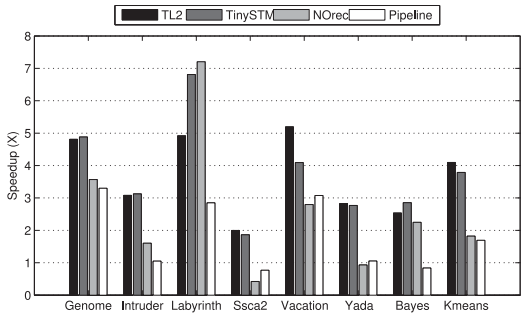

Fig. 1. Speedup of STAMP applications with various STMs.

Thread 1	Thread 2
atomic {	
x = 1;	
}	
	atomic {
	y = 42;
	}
	r1 = x

Thread 1	Thread 2
	atomic {
	y = 42;
	}
	r1 = x
atomic {	
x = 1;	
}	

Fig. 2. A racy program, as `Thread1` and `Thread2` may access `x` concurrently. Based on the strict transactional data race definition, the execution on the left does not expose the race, while the execution on the right does.

write-buffered STM and that implements SGLA as described by Menon et al. [2008]. All the STMs tested, except Pipeline, allow read-only transactions to commit in parallel; TL2 and TinySTM also allow read-write transactions that have no conflicts to commit in parallel. The graph clearly shows that the performance of Pipeline is considerably lower than the other STMs: in some cases, such as *Bayes* and *Intruder*, Pipeline performance with eight threads barely matches single-threaded execution. On the other hand, TL2 and TinySTM achieve good speedups for most of the cases. Because of these performance reasons, programmers have been averse to using STMs with SGLA semantics, while STMs with higher concurrency are widely spread (e.g., TinySTM is the reference design for GCC-TM [Schindewolf et al. 2009]). This, in turn, limits the applicability of race detection tools based on the strict transactional happens-before relation.

The strict transactional data race may produce results that do not fit with programmers' intuitive expectations for transactions' execution. Consider the example in Figure 2: this program is racy, as `Thread1` and `Thread2` may access `x` concurrently. Indeed, if we consider the program execution on the right, in which `Thread2`'s transaction (TX_2) commits before `Thread1`'s transaction (TX_1), then access to x in TX_1 and in `r1 = x` are not ordered by $\longrightarrow_{shb}$, and hence there is a data race. However, if we consider the program execution on the left, in which the thread interleaving is such that TX_1 commits before TX_2, then $TX_1 \longrightarrow_{shb} TX_2$. By program order, $TX_2 \longrightarrow_{shb}$ `r1 = x`, and hence $TX_1 \longrightarrow_{shb}$ `r1 = x`. It follows that, in this execution, no data race is detected, which contradicts the programmer's expectation. Even for this simple program and assuming that the two interleavings are equally distributed, a tool based on the strict happens-before relation has only 50% probability to expose the data race (the execution on the right in Figure 2), and hence a programmer is required to run the race detection tool at least two times with no certainty that both executions of the program will be tested. From a practical point of view, a programmer needs to check

the program a number of times N empirically determined and large enough that there is reasonable confidence in the results. For programs more complex than the one in Figure 2, the possible thread interleavings are difficult to determine and N becomes unbounded. In such case, the programmer is forced to run the race detection tool many times, which drastically increases the debugging time, and with no guarantees on the final result. Note that, since TX_1 and TX_2 do not conflict, a programmer expects the two transactions to run and commit in parallel. However, this program execution is forbidden under STMs that implement SGLA.

In order to remove the excessive constraints and limitation imposed by the strict transactional happens-before relation, we propose an alternative definition of a data race based on a new relation that weakens the strict transactional happens-before by ordering only conflicting transactions yet still maintaining soundness and correctness. This definition is more intuitive and can be used with a broader set of high-performance STMs, even those not implementing SGLA. To this extent, we provide the following definitions to include less than the full atomic order:

Definition 5. Two transactions conflict if they access the same location, at least one is a write, and they are executed by different threads.

Definition 6. *Only* conflicting transactions in the execution are totally ordered by a *relaxed transaction order* ($\longrightarrow_{rto}$). If transaction A transactionally ordered before B, then access a in A is transactionally ordered before access b in B.

Definition 7. The irreflexive transitive closure of program order, sync-primitive order, and relaxed transaction order define a *relaxed happens-before* ($\longrightarrow_{rhb}$) partial ordering on all accesses in the execution.

As opposed to $\longrightarrow_{shb}$, two nonconflicting transactions are not ordered by $\longrightarrow_{rhb}$.[1]

Definition 8. A *relaxed transactional data race* exists between two accesses in a given execution if and only if they access the same location, at least one is a write, they are executed by different threads, and they are not ordered by $\longrightarrow_{rhb}$.

Definition 9. A program is correctly synchronized if and only if no relaxed transactional data races exist in any execution of the program.

This correctness model can be used with STMs that do not implement SGLA semantics because the underlying definition of transactional data race does not assume total ordering among all transactions. In the particular example in Figure 2, `r1 = x` in `Thread2` and the access to x in `Thread1` are not ordered by $\longrightarrow_{rhb}$, and hence the data race is exposed in both the executions in the figure, regardless of the order in which transactions commit. Note that the programmer's expected execution in which TX_1 and TX_2 run and commit in parallel is also allowed by the underlying STM and that this program execution also exposes the data race.

The definition of data race based on the relaxed transactional happens-before relation is more robust than the one based on the strict happens-before relation. More specifically, two transactions ordered by $\longrightarrow_{rhb}$ are also ordered by $\longrightarrow_{shb}$, that is, $TX_1 \longrightarrow_{rhb} TX_2 \Rightarrow TX_1 \longrightarrow_{shb} TX_2$, but not vice versa. Let's define E as the set of strict happens-before edges in a given execution and E' as the set of relaxed happens-before edges in the same execution. Since the set of conflicting transactions is a subset of all transactions, it follows that $E' \subseteq E$. The other direction is not true (e.g., read-only transactions): $TX_1 \longrightarrow_{shb} TX_2 \not\Rightarrow TX_1 \longrightarrow_{rhb} TX_2$ because $E \not\subseteq E'$. Summarizing, in

[1]The relaxed happens-before relation is equivalent to "Disjoint Lock Atomicity" of Menon et al. [2008] and the weakest "conflicting regions" semantics defined by Grossman et al. [2006].

a given execution, all data races detected by an algorithm based on $\longrightarrow_{shb}$ are also detected by an algorithm based on $\longrightarrow_{rhb}$ but not vice versa.

3. RACE DETECTION ALGORITHMS

In this section, we describe our race detection algorithm, TRADE. For comparison reasons we also design an algorithm, s-TRADE, that detects races based on the strict transactional happens-before relation. s-TRADE is designed for STM systems that implement SGLA, as it assumes a form of total ordering among all transactions in the system. TRADE, instead, is based on the relaxed transactional happens-before relation and does not assume SGLA, and hence it covers a broader set of STM implementations (Section 3.2).

In both algorithms, the happens-before relation is represented using *vector clocks* [Lamport 1978]. A vector clock $VC : Tid \to Clk$ records a clock for each thread $t \in Tid$, where Tid is the set of all threads in the system. Vector clocks are partially ordered ($\sqsubseteq$) in a point-wise manner, with an associated join operation ($\sqcup$) and minimal element ($\bot$). We also define the helper function inc_t, which increments the tth component of a vector clock:

$$\begin{array}{rcl} V_1 \sqsubseteq V_2 & \texttt{if and only if} & \forall t \in Tid, V_1(t) \leq V_2(t) \\ V_1 \sqcup V_2 & = & \lambda t.max(V_1(t), V_2(t)) \\ \bot_V & = & \lambda t.0 \\ inc_t(V) & = & \lambda u.\ \texttt{if}\ u = t\ \texttt{then}\ V(u) + 1\ \texttt{else}\ V(u) \end{array}$$

The semantics of TM requires transactions to be executed entirely and atomically and their effects to be permanent only after successful commits. On abort, all transactional operations appear as if they were never executed and their effects (writes to memory) should not be visible to other threads. Atomicity is achieved by two possible designs that differ in the way the STM updates modified memory locations: *in place* and *write buffered*. In-place STMs optimistically update shared memory locations when a transactional write is issued; if the transaction aborts, the modified memory locations are rolled back to their original values. The main difference between in-place and write-buffered STMs is whether aborted transactions are considered part of the execution (*visible abort semantics*) or not (*invisible abort semantics*) [Shpeisman et al. 2009]. In-place STMs usually adopt visible abort semantics, which means that transactional writes performed by an aborted transaction can still originate races. In this case, our algorithms verify the occurrence of a race when a transactional operation is issued, even if the transaction eventually aborts. With write-buffered STMs, instead, aborted transactions are generally not part of the execution; thus, transactional writes performed by aborted transactions do not originate data races. In this case, our algorithms perform race detection upon successful validation of a transaction at commit phase.

3.1. s-TRADE Race Detection Algorithm

For STM systems that implement SGLA semantics, total ordering is defined by strict transactional order. For these STMs, we propose the following algorithm based on vector clocks to detect races based on strict transactional happens-before relation. We define:

$$\begin{array}{ll} C : Tid \to VC & R : Var \to VC \\ W : Var \to VC & G : Var \to VC, \end{array}$$

where C is the vector clock of each thread $t \in Tid$. The clock entry $C_t(u)$ records the clock for the last operation of thread u that happens before the current operation of thread

t. W_x and R_x are the write and read vector clocks of a variable $x \in Var$, respectively. For any thread t, $R_x(t)$ and $W_x(t)$ record the clock of the last read and write to x by thread t. G is the global vector clock that imposes a strict transactional happens-before order between operations of different threads. Essentially, G enforces the semantics of a single global lock.

For each thread $t \in Tid$, the algorithm follows the next rules:

[**Thread Creation**]

$$\forall t \in Tid,\ i = 1..N \quad C_t(i) := 0$$
$$C_t(t) := 1$$

[***Before* Transaction Commit**]

$$C_t := C_t \sqcup G$$

[***After* Transaction Commit**]

$$G := C_t$$
$$inc_t(C_t)$$

[***TX/Non-TX* Read Shared**]

$$R_x(t) := C_t(t)$$
$$W_x \sqsubseteq C_t \Rightarrow RaceFree$$

[***TX/Non-TX* Write Shared**]

$$W_x(t) := C_t(t)$$
$$W_x \sqsubseteq C_t \quad and \quad R_x \sqsubseteq C_t \Rightarrow RaceFree$$

At thread creation, all entries in the vector clock of thread t (C_t) are initialized to 0, except the thread's clock ($C_t(t) = 1$). Strict transactional happens-before relations are established by G: at commit phase, before the STM updates the modified memory locations, the thread's vector clock is joined to the global clock G. After the transaction has been validated, the thread's vector clock is copied to G and then $C_t(t)$ is increased to record that now thread t has moved to the next clock.

For this algorithm, the rules for transactional and nontransactional read/write operations are the same. Race detection for in-place STMs that implement visible abort semantics is performed at the moment the transactional operation is issued, similarly to nontransactional operations. For write-buffered STMs that implement invisible abort semantics, the actual race detection for transactional operations is performed during the validation of the transaction at commit phase.

Read operations. For both transactional and nontransactional operations, the variable's read clock of the reading thread $R_x(t)$ is updated with the current thread's clock $C_t(t)$. Next, the algorithm searches for the occurrence of a possible *write-read* race: if all modifications to x are prior to the current thread's vector clock ($W_x \sqsubseteq C_t$), then there is no race detected.

Write operations. Transactional and nontransactional writes to shared memory locations behave similarly to their respective read operations except that the algorithm also searches for *write-write* and *write-read* races: if all reads from x ($R_x \sqsubseteq C_t$) and all modifications to x ($W_x \sqsubseteq C_t$) are prior to the thread's vector clock, then there is no strict transactional race.

Figure 3 shows the evolution of the s-TRADE race detection algorithm for the left-side execution in Figure 2, where `Thread1`'s transaction commits before `Thread2`'s transaction. In this example, we assume a write-buffered STM, and thus race detection is performed at commit phase (the read and write vector clocks for memory locations that are transactionally accessed are updated at the end of the transaction).

s-TRADE first joins C_1 with G before entering the commit phase (`end_tx`). In the commit phase, it updates W_x with the current clock of `Thread1` and checks that the write happens after all previous accesses ($W_x \sqsubseteq C_t \quad and \quad R_x \sqsubseteq C_t$). After `end_tx`, G is

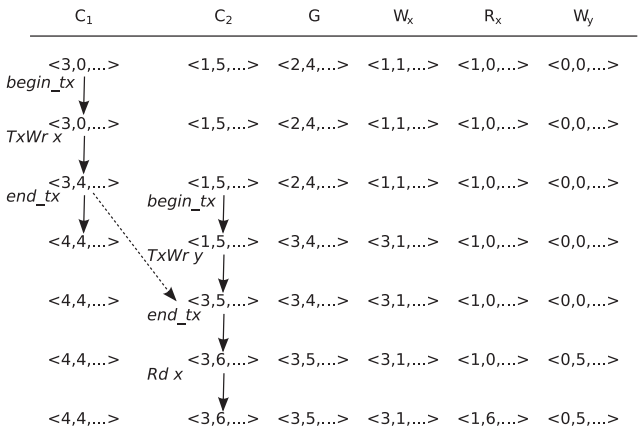

Fig. 3. Example trace for the program in Figure 2 running on an STM that implements SGLA. No data races detected.

updated with C_1 and the current vector clock of `Thread1` is incremented. `Thread2` also performs similar operations on C_2, W_y, and G when it commits. When `Thread2` accesses x nontransactionally, s-TRADE updates R_x with the current clock of `Thread2` and compares the vector clocks, $W_x =< 3, 1, \ldots >$ and $C_2 =< 3, 6, \ldots >$, $W_x \sqsubseteq C_2$. Since this check passes, this read is ordered with the previous writes to x, and no race is reported.

3.2. TRADE Race Detection Algorithm

Unlike the previous algorithm, where producing strict happens-before edges only requires a single global vector clock, in this algorithm, there are relaxed happens-before edges between any two conflicting transactions, one edge for each variable. We say that a transaction TX_i is ordered with a transaction TX_j on a shared variable $x \in Var$ if they both access x and at least one access is a write. It follows that TX_i and TX_j can be ordered on x but not on y, if they do not conflict on y. The correctness proof of the algorithm is in the appendix.

The relaxed transactional happens-before relation is established as follows: for each transactional read operation, a transaction TX has an incoming edge from previous transactions that wrote the same memory location. Transactional write operations induce incoming edges between a transaction TX and previous transactions that read or wrote the same memory location. Conversely, there are outgoing edges from a transaction TX to subsequent transactions that read or write the memory locations accessed by TX. For each transactional read operation, there is an outgoing edge to subsequent transactions that write the same memory location. Similarly, each transactional write operation generates an outgoing edge to transactions that read or write the same variable. It follows that nonconflicting transactions (e.g., read-only transactions) are not ordered by the relaxed transactional happens-before relation.

For this algorithm, we define:

$$C : Tid \to VC$$
$$W : Var \to VC \qquad R : Var \to VC$$
$$TW : Var \to VC \qquad TR : Var \to VC.$$

In addition to C, W, and R defined previously, $TW_x(t)$ and $TR_x(t)$ record the clock of the last transactional write and read to x by thread t, respectively. These vector clocks

are used to establish relaxed transactional happens-before orders between operations performed by different threads. $\forall t \in Tid$, the algorithm follows the next rules:

[THREAD CREATION]

$$\forall t \in Tid, i = 1..N \quad C_t(i) := 0$$
$$C_t(t) := 1$$

[TRANSACTION COMMIT]

$$inc_t(C_t)$$

[*Non-TX* READ SHARED]

$$R_x(t) := C_t(t)$$
$$W_x \sqsubseteq C_t \Rightarrow RaceFree$$

[*TX* READ SHARED]

$$C_t := C_t \sqcup TW_x$$
$$R_x(t) := C_t(t)$$
$$TR_x(t) := C_t(t)$$
$$W_x \sqsubseteq C_t \Rightarrow RaceFree$$

[*Non-TX* WRITE SHARED]

$$W_x(t) := C_t(t)$$
$$W_x \sqsubseteq C_t \quad and \quad R_x \sqsubseteq C_t \Rightarrow RaceFree$$

[*TX* WRITE SHARED]

$$C_t := C_t \sqcup TW_x \sqcup TR_x$$
$$W_x(t) := C_t(t)$$
$$TW_x(t) := C_t(t)$$
$$W_x \sqsubseteq C_t \quad and \quad R_x \sqsubseteq C_t \Rightarrow RaceFree$$

At the beginning of the application, the vector clock of each thread $t \in Tid$ is initialized to 0. When a thread t is created, its clock $C_t(t)$ is set to the value 1. $C_t(t)$ is updated at the end of every transaction. The read and write vector clocks of a variable x, R and W, are initialized to 0 the first time x is accessed for reading or writing, respectively. TR and TW are also initialized to 0 the first time a variable is accessed transactionally. To guarantee atomicity, for write-buffered STMs with invisible abort semantics, the algorithm updates transactional read and write vector clocks and performs race detection at commit phase.

Read operations. Nontransactional read operations update the tth entry of a variable x read clock with the current thread clock ($R_x(t) := C_t(t)$) and then check whether a *write-read* race has occurred: if all writes to x are precedent to the last observed vector clock of thread t ($W_x \sqsubseteq C_t$), then there is no race. Transactional read operations first update the thread vector clock with the write transactional dependency clock ($C_t := C_t \sqcup TW_x$). This operation builds incoming edges between the current read operation and previous transactional write operations to x. Next, TR_x is updated to build outgoing edges between the current transaction and any subsequent transaction that writes x. As for nontransactional read operations, transactional read operations update the read vector clock of the variable x (R_x) and perform race detection.

Write operations. Nontransactional write operations update the variable's write clock W_x with the value of the thread's clock ($W_x(t) := C_t(t)$) and then check for a possible occurrence of a data race. For write operations, both *read-write* and *write-write* transactional race conditions must be checked. On transactional writes, the thread vector clock is updated with both read and write transactional dependency vector clocks ($C_t := C_t \sqcup TW_x \sqcup TR_x$): this operation builds incoming edges from previous transactions that read or wrote x and the current transaction. Next, TW_x is updated with the current thread clock: this operation builds outgoing edges between the current transaction and any subsequent transaction that reads or writes x. Finally, transactional write

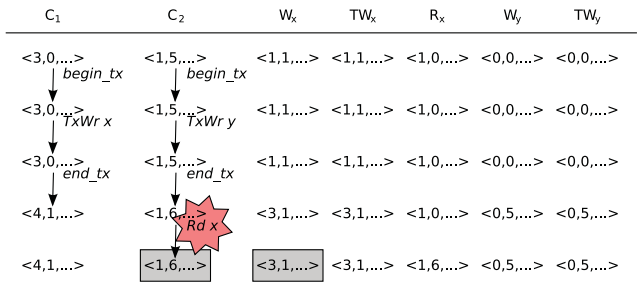

Fig. 4. Example trace for the program in Figure 2 running on an STM that does not implement SGLA semantics. TRADE detects a data race in a given execution.

operations update the write vector clock W_x and check if the current transactional write has generated any *write-write* or *write-read* data race.

Figure 4 shows the evolution of the TRADE algorithm for an execution of the example in Figure 2 in which the two transactions commit in parallel (as discussed in Section 2, this execution is forbidden in STM that implements SGLA). Similarly to the example presented in Figure 3, we show the evolution of the TRADE algorithm for a write-buffered STM with invisible abort semantics. It follows that transactional read and write vector clocks are updated at commit phase as well as race detection. In this example, `Thread1`'s and `Thread2`'s transactions do not conflict and are thus allowed to run and commit concurrently. Since the two transactions do not conflict, there are no relaxed happens-before edges between them. It follows that when `Thread1` commits, it first applies the rules for transactional write, which updates C_1 to be $C_1 \sqcup TW_x \sqcup TR_x$, modifies Wx and TW_x with the current clock of `Thread1` and checks the race freedom condition for the transactional write to variable x. Then `Thread1` increments its own clock, which switches to the vector clock representation $< 4, 1, \ldots >$ for C_1. Similarly, `Thread2` follows the rules for transactional write and commit at commit phase. When `Thread2` accesses x nontransactionally, the algorithm checks if this read happens after the last write of each thread to variable x, that is, $W_x \sqsubseteq C_2$. Since the comparison of two vector clocks fails, $W_X =< 3, 1, \ldots >$ while $C_2 =< 1, 6, \ldots >$, the algorithm detects a data race.

3.3. Extensions

Besides transactions and read/write accesses, our algorithms also track common operations that induce a partial/global ordering, such as thread creation/destruction and global barriers. Our algorithms only need to track barrier releases, which indicate that all threads have reached the barrier.[2] The vector clock of a thread is updated with the current vector clock of each thread (maximum across all threads' vector clocks) and then all threads move to the next clock by adding one to each entry of their vector clock:

[**Barrier Release**]

$$\forall t \in Tid,\ C_t = inc_t\left(\bigsqcup_{u \in Tid} C_u\right)$$

Fork and join operations also introduce a partial ordering between parents and children. Thread fork/join operations follow the next rules, where we assume that thread

[2]In case only a subset of threads participate in the barrier, we also need to track barrier entries and record which thread is involved in the barrier and should update its clock.

t is the parent of thread u:

[**FORK**]

$$C_u := C_u \sqcup C_t$$
$$C_t := inc_t(C_t)$$

[**JOIN**]

$$C_t := C_t \sqcup C_u$$
$$C_u := inc_u(C_u)$$

4. DESIGN AND IMPLEMENTATION

This section presents the design and implementation of the TRADE algorithm for transactional C/C++ applications. For simplicity and clarity of presentation, we present the implementation of TRADE for write-buffered STMs (e.g., TL2), where transactional writes are buffered into a local data structure and the effects of transactional writes happen after successful validation of the transaction at commit phase. The implementation for in-place systems with visible abort semantics is straightforward from the modifications discussed in Section 3. We also omit the s-TRADE implementation description for the same reasons. Our prototype implementation of this algorithm checks if a TM program has a race based on the relaxed transactional data race definition in a particular execution. If the tool detects data races, it reports the address and the instruction of each detected race.

Dynamic race detection tools need to intercept accesses to shared memory locations. For Java applications, this can be done by the JVM; however, for applications written with unmanaged languages (e.g., C/C++), there is no such virtual machine, and thus we need alternative methods. Current TM compilers only provide automatic instrumentation of transactional accesses [Adl-Tabatabai et al. 2012; compiler support in gcc 4.7 2010]. None of them, to the best of our knowledge, instruments nontransactional accesses or provides hooks for dynamic checking tools. Finally, manual instrumentation is prohibitive for large, real applications such as the ones tested in this work. We thus implemented TRADE on top of Pin [Luk et al. 2005], a dynamic instrumentation tool that allows programmers to instrument transactional and nontransactional accesses, as well as functions' entry and exit points.

In order to distinguish transactional code from nontransactional code, we instrument a begin/end transaction. `PIN_TM_BEGIN()` is invoked before the execution of the transaction: the function sets a flag indicating that the thread has now entered a transaction. At commit stage, the STM library checks whether there are unresolved conflicts, and if not, the transaction commits and `PIN_TM_END()` is invoked. This function checks for the occurrence of race conditions for each memory location in the read-/write-set. Note that, on abort, the transaction restarts from the beginning; thus, `PIN_TM_END()` is not invoked.

TRADE associates a `ThreadState` structure to each thread (see Figure 5). `ThreadState` contains a unique thread identifier `tid` and a vector clock `C`. Since a thread vector clock `C` is private to each thread and there are no concurrent accesses to the particular thread vector clock, there is no need to protect thread vector clocks with any lock. Each memory access has an association with `VarState` containing read and write vector clocks, `R` and `W`, respectively. Besides `R` and `W`, TRADE requires `TxR` and `TxW` to be able to implement the relaxed transactional happens-before edges. Unlike thread vector clocks, transactional and nontransactional read and write vector clocks are accessed by many threads concurrently and must be protected by locks: we use fine-grained read/write locking (each read/write vector clock is protected by a specific lock) so that multiple threads can access disjoint vector clocks in parallel.

```
class ThreadState {
  int C[];
  int tid;
  List TxRead_list;
  List TxWrite_list;
}
class VarState {
  int R[];
  int W[];
  int TxR[];
  int TxW[];
}
void TRADE_TXread(VarState x, ThreadState t){
  t->TxRead_list.insert(x);
}
void TRADE_TXwrite(VarState x, ThreadState t){
  t->TxWrite_list.insert(x);
}
void TRADE_commit(ThreadState t){
  while t->TxRead_list not empty
    lazy_TXread(t->TxRead_list.remove(), t)

  while t->TxWrite_list not empty
    lazy_TXwrite(t->TxWrite_list.remove(), t)
}
void TRADE_lazy_TXread(VarState x, ThreadState t){
  vc_cup(t.C, x.TxW);
  x.R[t.tid] = t.C[t.tid];
  x.TxR[t.tid] = t.C[t.tid];
  //write-read race?
  if (x.W[u] > t.C[u] for any u) Race!;
}
void TRADE_lazy_TXwrite(VarState x, ThreadState t){
  vc_cup(t.C, x.TxW);
  vc_cup(t.C, x.TxR);
  x.W[t.tid] = t.C[t.tid];
  x.TxW[t.tid] = t.C[t.tid];
  //read-write race?
  if (x.R[u] > t.C[u] for any u) Race!;
  //write-write race?
  if (x.W[u] > t.C[u] for any u) Race!;
}
```

Fig. 5. Implementation of the TRADE algorithm.

Figure 5 shows the most important TRADE event handlers, such as `TRADE_TXread()` and `TRADE_TXwrite()`, used to track transactional read and write accesses, respectively. TRADE is transparent to the underlying STM design and does not rely on the particular STM implementation or data structures (e.g., the read- and write-set). When a transactional operation is issued, we use shadow temporal read/write data structures (linked list without duplicates) to record memory locations accessed within a transaction. The semantics of transactional memory implies that modified memory locations are permanently visible to other threads only once the transaction has committed; thus, we perform lazy race detection at commit phase. In more detail, for each transactional read/write operations, the algorithm establishes relaxed transactional happens-before edges between the current and preceding transactions: `TRADE_lazy_TXread()` and `TRADE_lazy_TXwrite()`) update the thread vector clock with the maximum clock values between the thread vector clock (`C`) and `TxR` and/or `TxW`. Moreover, `TxR` and `TxW` are updated with the clock of the thread to establish relaxed transactional happens-before edges with the following conflicting transactions. If the tool detects a write-read race (`W>C`) for transactional reads or read-write/write-write races (`R>C`/`W>C`) for transactional writes, it raises an alarm. Note that it is not necessary to check race freedom conditions for repeated accesses to a variable within a transaction: because of the TM

Table I. Number of Existing and Injected Data Races Detected by s-TRADE and TRADE During 10 and 1 Executions, Respectively

	# Detected Races s-TRADE		# Detected Races TRADE	
Appl.	Existing Races	Injected Races (10 Runs)	Existing Races	Injected Races (1 Run)
Intruder	0	1,221*	0	1,705
Ssca2	99	102	99	102
Kmeans	0	238,457	0	253,044
Vacation	0	68	0	6,733
Genome	0	10	0	73
Yada	0	6	0	6
Bayes	0	0	0	3
Labyrinth	0	5	0	5

**Intruder* crashed because of the injected bug.

atomicity property, all accesses to the same variable in a transaction have the same thread vector clock ($C_t(t)$) value. For example, only the last transactional write to a variable x is checked regardless of how many transactional writes to x have been performed within the transaction.

The pseudocode for nontransactional read/write operations is exactly the same as their equivalent lazy transactional reads/writes, except for the operations that require `TxR` and `TxW` modification.

5. EVALUATION

To assess the effectiveness of TRADE and compare with s-TRADE, we run applications from the STAMP benchmark suite, widely used to test TM systems. Note that s-TRADE can only be used with STMs that implement SGLA semantics; TRADE, instead, does not have this constraint. We thus choose Pipeline and LLT from the RSTM library to have STMs with and without SGLA semantics, respectively. The STAMP applications and the tested STMs are compiled with gcc 4.4.5 with optimization level O3 for 64-bit architectures and run on an Intel Nehalem system (eight cores, 16GB of RAM). In all the experiments, we run eight threads. To ensure a fair comparison, both tools are implemented on top of Pin and as similarly as possible.

Race detection analysis. Table I reports the number of existing and injected data races detected by s-TRADE and TRADE when using Pipeline and LLT as an underlying STM, respectively. Note that the number of races detected by TRADE in a given execution is the same even with Pipeline. The second and fourth columns in the table show the number of data races for the unmodified version of STAMP applications. Even though STAMP benchmarks are mature applications, both our tools detect the same data races for *SSCA2*.

SSCA2 consists of four kernels that work on a large, directed, weighted multigraph and includes a scalable data generator that produces edge tuples containing a start and end vertex and the weight of each directed edge. The transactional implementation focuses on kernel 1 (which builds the graph) and presents short transactions with small read-/write-sets. The large number of nodes in the graph leads to infrequent conflicts. Moreover, in several parts, each thread works on a disjoint partition of the graph, without synchronizing the accesses to the nodes. Both s-TRADE and TRADE detect data races in the process of creating the inner vertex list (`inVerList`). This code is enclosed between two global barriers, and no transactions are used to protect accesses to the graph's nodes; hence, the accesses to the nodes are ordered neither by $\longrightarrow_{shb}$

```
thread_barrier_wait();
for (i = i_start; i < i_stop; i++) {
    for (j = inVerId[i]; j < (inVerId[i] + Degree[i]); j++) {
            if ((j - inVerId[i]) < MAX_SIZE) {
             inVerList[j] = List[i*MAX_SIZE+j-inVerId[i]];
        } else {
             inVerList[j] = auxArr[i][(j-inVerId[i]) % MAX_SIZE];
        }
    }
}
thread_barrier_wait();
```

Fig. 6. SSCA2 code snapshot.

nor by $\longrightarrow_{rhb}$. Figure 6 shows the snapshot of the code where the data races occur. During the creation of the inner vertex list, each thread accesses the next `Degree[i] > 0` vertexes of a vertex with Id `inVertId[i]`, where `Degree[i]` is the degree of vertex `i`. Since the indexes `j` in different threads may overlap, thread t_k may access the first nodes in the thread t_{k+1} partition when analyzing the last nodes (`i = i_stop-1`) in its partition without synchronization, which creates a write-write data race.

To further investigate the effectiveness of our tools, we inject synthetic data races into STAMP applications in the form of removing transactions. The third and fifth columns in Table I show the number of injected data races detected by 10 iterations of s-TRADE running with Pipeline and one iteration of TRADE running with LLT. As described in Section 2, TRADE has a higher probability of exposing data races than s-TRADE in a single execution due to the lower number of transaction orderings established by TRADE. In more detail, all happens-before edges produced by TRADE are also produced by s-TRADE but not vice versa, which means that all races detected by s-TRADE are also detected by TRADE, but not vice versa. The result is that TRADE detects several races that were not exposed by any of the 10 runs of s-TRADE, confirming that TRADE race detection is more robust than s-TRADE. This experiment also shows how difficult it is to empirically estimate the number of iterations for s-TRADE to detect the same number of races detected by TRADE with a single execution. In some cases (e.g., *Yada*), 10 iterations may be sufficient, while in other cases (e.g., *Vacation*), performing 10 iterations still leaves 6,665 data races unexposed. Note that, as with other race detection tools based on different forms of happens-before relations, TRADE and s-TRADE may not detect some races that require a particular thread interleaving to be exposed. Nevertheless, TRADE always detects a superset of the races detected by s-TRADE and only gives warnings for true races.

Performance analysis. To make a fair comparison between s-TRADE and TRADE and evaluate the performance impact of tracking relaxed happens-before edges, we run both race detection tools on STAMP applications using Pipeline as the underlying STM. Figure 7 shows the ratio of the execution time with TRADE to the execution time with s-TRADE for each application. Intuitively, TRADE is expected to introduce larger overhead than s-TRADE because the tool needs to track potential relaxed happens-before edges between transactional read and write operations (nontransactional operations behave similarly). However, as we can see from Figure 7, TRADE generally introduces lower overhead than s-TRADE because there are other factors that also account for the total runtime overhead when running STAMP applications.

STMs that implement SGLA do not allow multiple threads to commit in parallel; s-TRADE poses the additional constraint that race detection must also be carried on as part of the commit operation. This is necessary to ensure that no other thread updates any read/write vector clocks while a thread is still performing race detection. On the other hand, the serialization time at commit phase increases and eventually worsens performance. The serialization time depends on two parameters: (1) the total

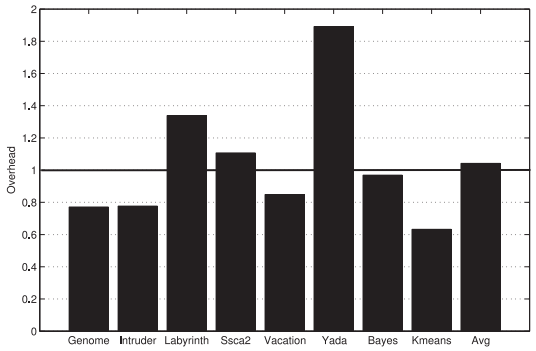

Fig. 7. Ratio of the application execution time with TRADE to the application execution time with s-TRADE. For both tools, we use Pipeline as the underlying STM.

number of transactions (second column in Table II) and (2) the size of each transaction, computed as the average number of transactional accesses per transaction (eighth column in Table II). Applications with a large number of transactions (such as *Kmeans* and *Intruder*) show, in general, better performance with TRADE than with s-TRADE. Although *Intruder* and *SSCA2* have a comparable number of transactions, s-TRADE performs slightly better than TRADE for the latter. This is because *SSCA2*'s transactions are smaller than *Intruder*'s, which means that *SSCA2*'s serialization time is shorter.

For applications with a limited number of transactions (*Yada*, *Labyrinth*, and *Bayes*), the serialization overhead at commit phase is negligible. In these scenarios, establishing relaxed happens-before edges penalizes TRADE with respect to s-TRADE. This effect can be seen especially with *Labyrinth* and *Yada*, while for *Bayes* the two tools perform similarly. In more detail, for *Bayes*, the ratio between transactional and nontransactional accesses (0.15) is much lower than *Labyrinth* and *Yada* (15.09 and 26.14, respectively). This low ratio indicates that nontransactional operations are the primary factors in determining the overall execution time. On the other hand, transactional accesses are predominant for both *Labyrinth* and *Yada*. *Labyrinth* mainly accesses individual addresses transactionally, which only requires vector clock allocation and initialization. Nonunique transactional accesses, instead, require TRADE to determine the relaxed happens-before edges: the high overhead for *Yada* is caused by the large number of nonunique transactional accesses performed.

We also analyze the overhead introduced by TRADE and s-TRADE over native executions of STAMP applications instrumented with Pin running on top of LLT and Pipeline, respectively. Table III reports the runtime overhead and the total execution time of running an application with TRADE and s-TRADE. As shown in the table, TRADE's overhead is very contained, 3.17 times on average. This overhead is in line with state-of-the-art race detectors for lock-based applications. *SSCA2* is the only exception: for this application, both tools continuously evict data from the cache when managing vector clocks, which significantly affects performance. Our experiments show that, while the slowdown over the native execution for both tools is comparable, the total execution time with TRADE is considerably shorter than with s-TRADE (e.g., *Kmeans*, *Bayes*, *Labyrinth*), which increases programmers' productivity. Even for the cases in which the slowdown of the applications over native execution is higher for TRADE (e.g., *Intruder*), the total execution time for TRADE using LTT is lower than the corresponding execution time for s-TRADE using Pipeline. This is because STAMP applications perform better with LLT than with Pipeline. Additionally, given that a

Table II. STAMP Applications' Characteristics

Apps.	Transactions		#Transactional Accesses				Per Tx Accesses	#Nontransactional Accesses		Ratio between TX and Non-TX Accesses
			Total		Unique					
	Total	Read Only	Read	Write	Read	Write		Read	Write	
Intruder	6,045K	31.10%	55,752K	3,164K	9.85%	40.09%	9.74K	25,699K	10,338K	1.63
Ssca2	5,558K	0.00%	2,780K	5,560K	9.38%	54.69%	1.50K	157,486K	34,243K	0.04
Kmeans	10,207K	0.21%	13,113K	6,588K	0.01%	0.01%	1.93K	513,365K	1,507K	0.03
Vacation	2,097K	0.15%	288,957K	7,099K	4.68%	62.14%	141.18K	37,639K	14,645K	5.66
Genome	2,489K	56.55%	58,288K	1,638K	2.21%	80.24%	24.07K	10,392K	6,259K	3.59
Yada	30K	35.71%	1,647K	240K	18.71%	51.19%	62.92K	68K	57K	15.09
Bayes	2K	26.92%	33K	3K	2.57%	14.25%	18.00K	218K	19K	0.15
Labyrinth	1K	0.76%	92K	91K	98.05%	99.71%	183.00K	4K	3K	26.14

Table III. Performance Comparison Between TRADE Running on LLT and s-TRADE Running on Pipeline (Time in Seconds; Means for Overhead Are Geometric, and Means for Total Execution Time Are Arithmetic)

	TRADE with *LLT*		s-TRADE with *Pipeline*	
Apps.	Overhead	Execution Time (sec)	Overhead	Execution Time (sec)
Intruder	4.25x	270.95	3.46x	358.36
SSCA2	44.52x	566.86	32.93x	507.21
Kmeans	1.20x	178.75	1.14x	308.34
Vacation	4.45x	437.10	7.18x	518.78
Genome	1.88x	140.78	3.21x	191.21
Yada	4.11x	6.79	2.01x	3.65
Bayes	1.14x	41.23	1.16x	81.58
Labyrinth	1.15x	78.85	1.16x	111.57
Mean	3.17x	117.88	3.08x	143.98

programmer might need to run the same application with s-TRADE several times to be able to detect the same number of races detected by TRADE, the debugging time of an application with s-TRADE could be considered longer.

6. RELATED WORK

Dynamic race detectors for lock-based applications are often based on lockset [Savage et al. 1997; Von Praun and Gross 2001] or on happens-before relation [Schonberg 1989; Dinning and Schonberg 1991]. In lockset algorithms, each shared variable is associated with a lockset that keeps track of all locks held during accesses, and a race is reported when the lockset becomes empty. Happens-before algorithms are based on Lamport's happens-before relation [Lamport 1978], which combines program order and synchronization events to establish a partial temporal ordering of instructions. Smaragdakis et al. [2012] observe that the original happens-before relation for lock-based applications is excessively strict: they follow our same reasoning and introduce the concept of causally precedes (CP), where two critical sections are ordered only if they contain conflicting accesses. As we mentioned, transactions and locks are semantically different and we need TM-specific algorithms to detect races in TM programs.

There are only a few works on race detection that leverage TM. Gupta et al. [2009] present a system that modifies an HTM implementation to perform dynamic race detection (RaceTM) by using the conflict detection mechanisms of TM to detect data races for lock-based applications. RaceTM introduces a lower instrumentation overhead than TRADE, as shared memory accesses are tracked by the hardware. On the other hand, RaceTM is sensitive to thread migration, a higher number of false positives caused by false sharing inside cache lines, and a higher number of false negatives caused by cache eviction. In contrast, TRADE is based on a formal definition of data races for TM applications and it is not affected by many of these issues. Finally, TRADE does not require extra hardware.

Teixeira et al. [2010] detect data races in TM applications by converting transactions into lock-protected critical sections and applying an existing lock-oriented data race detector. The authors implemented their approach with AJEX [Dias and Teixeira 2009] to parse atomic blocks and use the JChord [Naik et al. 2006] lock-based data race detector. Goldilocks [Elmas et al. 2007] is a lockset-based algorithm for precisely computing the happens-before relation and detecting data races at runtime. The algorithm is implemented in the Kaffe Java Virtual Machine and evaluated with Java benchmarks and a few microbenchmarks that combine lock-based and

transaction-based synchronization. The authors use the source-to-source translation and protect all shared objects accessed in a transaction with per-object lock. The problem with these two approaches is that directly transforming transactions into a single global lock serializes the execution of transactions and modifies the runtime characteristics of the application, enforcing an artificial total sequential order.

Besides the Transactional Data Race Free (TDRF) model [Dalessandro and Scott 2009] based on the strict happens-before relation among transactions, other correctness models have been proposed in the literature. Static Separation (SS) in STM-Haskell [Harris et al. 2005] restricts how memory locations can be accessed during the execution of a program—the same location cannot be written both transactionally and nontransactionally. SS does not permit common programming techniques, such as initializing shared memory locations before the main thread creates secondary threads. *Dynamic Separation* (DS) in AME [Abadi et al. 2010] overcomes some of the limitations of SS by providing explicit operations, invoked by the programmer, to indicate when a location changes from being available for use inside transactions to being available for use directly. Both SS and DS require language support and extensions available only in STM-Haskell and AME, respectively, which limits their applicability. Kai Huang [2003] analyzes correct parallelizations of transactions-everywhere Cilk programs. In transactions-everywhere programs, every instruction is enclosed in a transaction, and thus there are no nontransactional memory accesses. A program can be transformed in a transactions-everywhere program by inserting cut points that delimit the beginning/end of each transaction. Cut points can be added by the programmer or by an automatic compiler (*atomization*). In both cases, the programmer needs to use a semantic checker, such as the one presented in Kai Huang [2003], to find possible error locations caused by erroneous insert of cut points. Our previous work, T-Rex [Kestor et al. 2014], introduces a precise dynamic race detection tool for C/C++ TM programs based on an intuitive definition of data race: a transactional data race occurs between accesses to a memory location from different threads, where at least one access is a write and at least one access is nontransactional.

Comparing data race detectors for different correctness models is difficult because each model poses different restrictions on the programmer and relies on a different definition of what it means for a program to be correctly synchronized. For example, a race detection tool for SS only needs to check whether a memory location is written both inside and outside transactional code to declare a data race. Such a tool would only require checking static conditions and would introduce minimum runtime overhead and would not be sensitive to thread interleaving compared to a tool based on happens-before, like TRADE. At the same time, however, SS poses considerable restrictions on the programmer. We compared T-Rex and TRADE using LLT as the underlying STM and the STAMP benchmarks. T-Rex does not need to track all happens-before edges and thus generally introduces lower overhead than TRADE: T-Rex is faster than TRADE for *Genome* (three times), *Intruder* (7.9 times), *SSCA2* (11.6 times), *Vacation* (9.8 times), and *Yada* (4.9 times), while *Bayes* and *Labyrinth* show comparable runtime overhead and TRADE is 1.5 times faster than T-Rex for *Kmeans*. However, T-Rex does not recognize some programming idioms such as privatization/publication. To avoid false positives detected by T-Rex in case an STM supports safe privatization/publication, the programmer needs to annotate privatization/publication idioms in the code.

7. CONCLUSIONS

TM is one of the programming models that helps programmers expose a higher level of parallelism in their applications. TM has reached a maturity level with many hardware (HTM) and software (STM) implementations available to programmers. However, there

is still a lack of debugging tools that automatically detect data races in TM programs, especially those written with unmanaged programming languages, such as C/C++.

In this article, we proposed TRADE, a novel algorithm that precisely detects data races for C/C++ TM applications. Our algorithm is based on the relaxed transactional happens-before relation that only orders conflicting transactions. This definition is closer to transactional memory semantics and more intuitive than the previously defined strict transactional happens-before relation.

We implemented a dynamic race detection tool based on TRADE: our tool precisely detects transactional races (real or injected) for STAMP applications. Thanks to our tool, we were able to identify data races in *SSCA2*. We also compared the TRADE tool with an equivalent race detection tool that implements an algorithm based on the strict happens-before relation (s-TRADE). The TRADE tool shows negligible overhead over s-TRADE but can be used with popular, high-performance STMs, which increases its practicability.

A. APPENDIX

In this appendix, we provide the logical reasoning of the algorithms' correctness proofs. For brevity, we prove the correctness of TRADE; the proof for s-TRADE is a simplified version of this.

We begin the correctness proofs with the formal definition of program state σ. At any given moment, a program is in a state σ represented by a tuple $<C, R, W, TR, TW>$.

Definition 10. A state $\sigma = <C, R, W, TR, TW>$ is *well formed* if and only if:

(1) for all $t, u \in Tid$, if $t \neq u$, then $C_u(t) < C_t(t)$;
(2) for all $x \in Var$, $t \in Tid$, $R_x(t) \leq C_t(t)$;
(3) for all $x \in Var$, $t \in Tid$, $W_x(t) \leq C_t(t)$;
(4) for all $x \in Var$, $t \in Tid$, $TR_x(t) < C_t(t)$;
(5) for all $x \in Var$, $t \in Tid$, $TW_x(t) < C_t(t)$.

Let σ_0 be the initial state where all vector clocks in the tuple are empty; then σ_0 is well formed. An execution trace α is a sequence of operations performed by all threads in a multithreaded program. The operations that a thread $t \in Tid$ can perform are the following:

—$rd(t, x)$ and $wr(t, x)$, which read and write a value from x nontransactionally, respectively;
—$txrd(t, x)$ and $txwr(t, x)$, which read and write a value x transactionally, respectively;
—$begin(t)$ and $end(t)$, which start and end a transaction, respectively;
—$fork(t, u)$, which forks a thread u;
—$join(t, u)$, which blocks until a thread u terminates; and
—*barrier*, which blocks until all threads $u \neq t$ reach the barrier.

When performing an operation a, a program moves from an initial state σ to a final state σ'. We indicate this transition with $\Rightarrow^a$. If the operation a is race free and σ is well formed, then σ' is also well formed.

THEOREM 1 (SOUNDNESS). *Given a well-formed state σ and $\sigma \Rightarrow^\alpha \sigma'$, if* TRADE *detects a data race ϕ, then ϕ is a race in α.*

PROOF. If TRADE detects a race between two operations a and b, then

(i) a and b conflict, and
(ii) $a \not\longrightarrow_{rhb} b$.

Let's assume that a and b do not race in α; then either a and b do not conflict, which contradicts (i), or $a \longrightarrow_{rhb} b$, which contradicts (ii). Hence, a races with b in α. □

LEMMA 1. *Let σ_a be a well-formed state and $a.\alpha$ be an execution trace such that $\sigma_a \Rightarrow^{a.\alpha} \sigma_b \Rightarrow^{b} \sigma_b'$. Let's also assume $t = tid(a)$ and $u = tid(b)$. If $C_t^a(t) \leq C_u^b(t)$, then $a \longrightarrow_{a.\alpha.b} b$.*[3]

THEOREM 2 (COMPLETENESS). *Given a well-formed state σ and $\sigma \Rightarrow^{\alpha} \sigma'$, all data races in α are detected by* TRADE.

PROOF. The theorem can be proved by contradiction. Suppose that there is a data race in α not detected by TRADE. This means that there are two operations in α, say, a and b, that conflict and $a \not\longrightarrow_{rhb} b$. The proof can be constructed by induction on $\alpha = a.\beta.b$, assuming that the data race between a and b is the first in the trace and that there are no other races between β and b. The process can then be repeated with $\alpha = \beta$. Finally, let's assume $t = tid(a)$ and $u = tid(b)$; by definition of data race $t \neq u$ and by Lemma 1 $C_t^a(t) > C_u^b(t)$ because $a \not\longrightarrow_{rhb} b$.

Without loss of generality, let's assume that a is a write operation to $x \in Var$ and b is a read operation from $x \in Var$. C_t^a and C_u^b denote the thread vector clocks at the time the operations a and b are executed.

—$a = txwr(t, x)$, $b = txrd(u, x)$. If $begin(u)$ occurs after $end(t)$ then:[4]
  $TW_x^a(t) = C_t^a(t)$ by $txwr(t, x)$
  $C_u^b = C_u \sqcup TW_x^a$ by $txrd(u, x)$
  It follows that $C_t^a(t) = C_u^b(t)$, which contradicts the initial hypothesis $C_t^a(t) > C_u^b(t)$, and hence $a \longrightarrow_{rhb} b$.
—All other cases (at least one operation is nontransactional):
  $W_x^a(t) = C_t^a(t)$ by $txwr(t, x)$, or $wr(t, x)$
  $W_x^a \sqsubseteq C_u^b$ is false in the $rd(u, x)$, or $txrd(u, x)$, because $W_x^a(t) > C_u^b(t)$
  It follows that TRADE detects a race between a and b, which contradicts the initial hypothesis that there was a race in α not detected by TRADE. □

REFERENCES

M. Abadi, T. Harris, and K. F. Moore. 2010. A model of dynamic separation for transactional memory. *Inf. Comput.* 1093–1117.

A. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Feb. 2012. Draft Specification for Transactional Features in C++. Version 1.1.

S. V. Adve and M. D. Hill. 1998. Weak ordering a new definition. In *Proc. of the International Symposia on Computer Architecture (ISCA'98)*. 363–375.

H.-J. Boehm. 2009. Transactional memory should be an implementation technique, not a programming interface. In *Proc. of the 1st USENIX Conference on Hot Topics in Parallelism*. 15–15.

D. Christie, J. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. 2010. Dresden TM compiler (DTMC). In *Proc. of the 5th ACM European Conference on Computer Systems*.

TM compiler support in gcc 4.7. 2010. Project Web Site. Retrieved from http://gcc.gnu.org/gcc-4.7/changes.html.

L. Dalessandro and M. L. Scott. 2009. Strong isolation is a weak idea. In *Proc. of the 4th Workshop on Transactional Computing*.

L. Dalessandro, M. F. Spear, and M. L. Scott. 2010. NOrec: Streamlining STM by abolishing ownership records. In *Proc. of the 15th Symposium on Principles and Practice of Parallel Programming*.

[3]The proof of this lemma can be found in Flanagan and Freund [2009]. We omit it here.

[4]If the two transactions run in parallel and there is a conflict between a and b, the TM system will abort one of the transactions, and hence a data race cannot occur, which contradicts the initial hypothesis.

R. Dias and B. C. Teixeira. 2009. *Ajex: A source-to-source Java STM framework compiler*. Technical Report. DI-FCT/UNL.

D. Dice, Y. Lev, M. Moir, and D. Nussbaum. 2009. Early experience with a commercial hardware transactional memory implementation. In *Proc. of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*.

D. Dice, O. Shalev, and N. Shavit. 2006. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing*.

A. Dinning and E. Schonberg. 1991. Detecting access anomalies in programs with critical sections. In *Proc. of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*.

T. Elmas, S. Qadeer, and S. Tasiran. 2007. Goldilocks: A race and transaction-aware java runtime. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

C. Flanagan and S. N. Freund. 2009. FastTrack: Efficient and precise dynamic race detection. In *Proc. of the Conference on Programming Language Design and Implementation*. 121–133.

D. Grossman, J. Manson, and W. Pugh. 2006. What do high-level memory models mean for transactions? In *Proc. of the Workshop on Memory System Performance and Correctness*. 62–69.

S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Rotteler. 2009. Using hardware transactional memory for data race detection. In *Proc. 23rd International Parallel and Distributed Processing Symposium*.

R. Haring. 2011. The Blue gene/Q compute chip. In *The 23rd Symposium on High Performance Chips*.

T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. 2005. Composable memory transactions. In *Proc. of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*. 48–60.

M. Herlihy and Y. Lev. 2009. tm db: A generic debugging library for transactional programs. In *Proc. 18th International Conference on Parallel Architectures and Compilation Techniques*.

Intel Corporation. May 2009. Intel Transactional Memory Compiler and Runtime Application Binary Interface. Document No. 318523-002US, revision 1.1.

K. Huang. 2003. Data-Race Detection in Transactions-Everywhere Parallel Programming. M.S Thesis, the Massachusetts Institute of Technology.

G. Kestor, R. Gioiosa, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. 2011. STM2: A parallel STM for high performance simultaneous multithreading systems. In *Proc. of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. 221–231.

G. Kestor, O. S. Unsal, A. Cristal, and S. Tasiran. 2014. T-Rex: A dynamic race detection tool for C/C++ transactional memory applications. In *Proc. of the 9th European Conference on Computer Systems (EuroSys'14)*. ACM, New York, NY, Article 20, 12 pages. DOI: http://dx.doi.org/10.1145/2592798.2592809

L. Lamport. 1978. Time clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7.

C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the Conference on Programming Language Design and Implementation*.

V. Menon, S. Balensiefer, T. Shpeisman, A. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. 2008. Practical weak-atomicity semantics for Java STM. In *Proc. of the Annual Symposium on Parallelism in Algorithms and Architectures*. 314–325.

C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proc. of the International Symposium on Workload Characterization*.

M. Naik, A. Aiken, and J. Whaley. 2006. Effective static race detection for Java. In *Proc. of the ACM Conference on Programming Language Design and Implementation*.

T. Riegel, C. Fetzer, and P. Felber. 2007. Time-based transactional memory with scalable time bases. In *19th ACM Symposium on Parallelism in Algorithms and Architectures*.

S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* 15, 4.

M. Schindewolf, A. Cohen, W. Karl, A. Marongiu, and L. Benini. 2009. Towards transactional memory support for GCC. In *GCC Research Opportunities Workshop (GROW'09)*.

D. Schonberg. 1989. On-the-fly detection of access anomalies. In *Proc. of the ACM Conference on Programming Language Design and Implementation*.

T. Shpeisman, A. Adl-Tabatabai, R. Geva, Y. Ni, and A. Welc. 2009. Towards transactional memory semantics for C++. In *Proc. of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*.

Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. 2012. Sound predictive race detection in polynomial time. In *Proc. of the Symposium on Principles of Programming Languages*. 387–400.

Rochester software transactional memory runtime. 2010. www.cs.rochester.edu/research/synchronization/rstm/.

B. C. Teixeira, J. Lourenco, and D. Sousa. 2010. A static approach for detecting concurrency anomalies in transactional memory. In *Proc. of InForum 2010*.

C. Von Praun and T. R. Gross. 2001. Object race detection. In *Proc. of the 16th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

F. Zyulkyarov, T. Harris, O. S. Unsal, A. Cristal, and M. Valero. 2010. Debugging programs that use atomic blocks and transactional memory. In *Proc. of the Symposium on Principles and Practice of Parallel Computing*.

Received March 2013; revised April 2014; accepted May 2015