An Oracle for Tolerating and Detecting Asymmetric Races

Darko Kirovski and Benjamin Zorn, Microsoft Research
Rahul Nagpal, IISc Bangalore
Karthik Pattabiraman, University of Illinois at Urbana-Champaign

Contact: {darkok,zorn}@microsoft.com

Technical Report MSR-TR-2007-122
September 2007

Microsoft Research
One Microsoft Way Redmond, WA 98052, USA
http://research.microsoft.com

An Oracle for Tolerating and Detecting Asymmetric Races

Darko Kirovski[◇], Benjamin Zorn[◇], Rahul Nagpal[†], and Karthik Pattabiraman[‡]
[◇]Microsoft Research, [†] IISc Bangalore, and [‡] University of Illinois at Urbana-Champaign

ABSTRACT

Because races represent a hard-to-manage class of errors in concurrent programs, numerous approaches to detect them have been proposed and evaluated. We consider specifically asymmetric races, a subclass of race conditions, where a programmer's thread correctly acquires and releases a lock for a specific variable, while another thread causes a race by improperly accessing this variable. We introduce ToleRace, an oracle that allows programs to either tolerate or detect asymmetric races based on local replication of shared state. ToleRace provides an approximation of atomicity in critical sections by creating local copies of shared variables when a critical section is entered and propagating the appropriate copy when the critical section is exited. We characterize the possible interleavings that can cause races and precisely describe the effect of ToleRace in each of these cases. We evaluate the theoretical aspects of the oracle and note that it could be implemented in hardware and/or software within a favorable range of overhead-to-benefit scenarios.

1. INTRODUCTION

Race conditions are memory errors that occur when multiple threads read and write a memory location in an underspecified order. Because race conditions depend on the interleaving of the memory operations of individual threads, they are notoriously difficult to reproduce and represent a major obstacle in the task of writing correct concurrent programs. The advent of multicore processors has generated significant interest in this problem [7, 8, 21, 24, 25, 26, 27, 29, 31].

Current approaches dealing with race conditions focus on the problem of race detection and face three significant obstacles. First, both static and dynamic methods for race detection can produce false positives that significantly reduce their effectiveness in practice (e.g., see [24, 27]). For example, recently published static analyzers report false positive rates of 90% [23, 27]. Because race conditions are difficult to reason about, determining if a race reported by a static detection tool is a real race can be time consuming and counter-productive. Second, dynamic race detection tools can have lower false positive rates, but also currently add significant overhead to execution time (ranging from 2x to 30x slowdown) [21, 29, 31]. Finally, even when a true race is detected, they can be difficult to correct. Whenever software errors are fixed, potential new errors may be introduced. For example, fixing a race condition may involve introducing additional synchronization constructs, and the incorrect use of such constructs can result in a deadlock. Programmers are faced with the choice between allowing a race that occurs infrequently and creating potential new synchronization errors that have broader impact. In large software systems races can require months to correct after being observed.

We present ToleRace, a runtime system that allows programs with concurrency errors to tolerate them and continue executing. Inspired by the DieHard runtime system [3], which probabilistically tolerates memory safety errors, ToleRace uses replication to deterministically or probabilistically manage asymmetric data races.[1] An asymmetric race occurs when one thread correctly protects a shared variable using a lock, while another thread accesses the same variable improperly due to a synchronization error (e.g., not taking a lock, taking the wrong lock, taking a lock late, etc.).

ToleRace provides an approximation of atomicity in critical sections by creating local copies of shared variables when a critical section is entered, detecting conflicting changes to shared data when the critical section is exited, and propagating the appropriate copy when possible to effectively hide the race. ToleRace allows a variety of implementations that range from software only, where races are only probabilistically detected and tolerated, to a combination of hardware and software, where stronger guarantees are possible.

ToleRace can be compared with transactional memory [17], which combines mechanisms for conflict detection and conflict resolution (through transaction abort and rollback). ToleRace allows a range of approaches to conflict detection that trade off precision with cost, while at the same time we demonstrate that it tolerates a number of race scenarios without requiring abort and rollback. ToleRace represents a family of possible runtime implementations that occupy the space between programming with existing locking mechanisms and using full transactional memory. In this paper, we focus on the fundamental properties of the ToleRace oracle, described below. The contributions of ToleRace include:

- **Comprehensive runtime management of races.** ToleRace allows programs with races to tolerate their existence by increasing the likelihood that races will not cause incorrect program behavior. Increasing a program's tolerance to races reduces the need for the race to be debugged/patched. In instances where ToleRace cannot tolerate races, it detects them either precisely or with high probability, depending on the implementation.
- **Precise detection.** ToleRace only identifies races that happen at runtime. It detects a race when the

[1] In the remainder of the paper, a referral to a race means a referral to an asymmetric race unless specified otherwise.

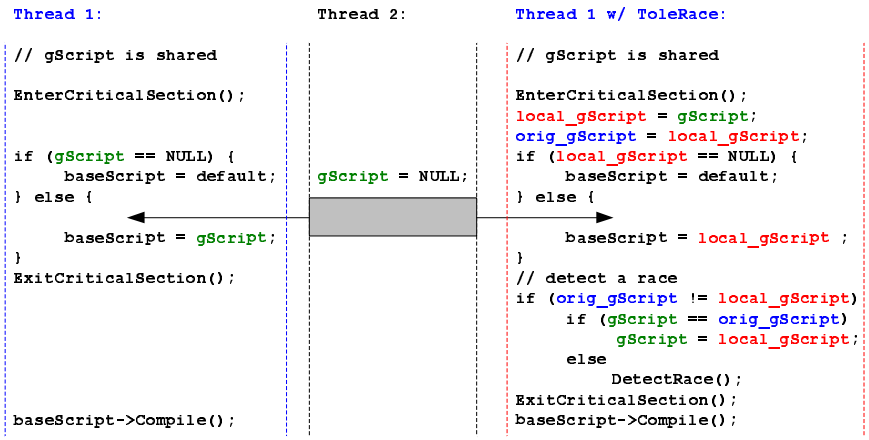

Figure 1: An example of an RwR race.

critical section in which the race takes place exits.

- **Programmer-centric local reasoning.** ToleRace enables programmer tools that allow local reasoning about correctness and that facilitate a structured means of detecting and tolerating errors that are caused by code outside a programmer's control. With ToleRace, a programmer can add compensation code at the exit of a critical section to create a custom resolution of a detected race. It can be used to detect/patch a specific race condition in the release version of the program without identifying the exact source of the bug.

The example in Figure 1, inspired by a real race detected in the Mozilla application suite [21], illustrates how ToleRace works. In this example, we see that Thread 1 correctly uses a critical section to protect its read accesses to the shared variable `gScript`. Thread 2 incorrectly updates `gScript` without a lock, creating a race. The race occurs infrequently, when Thread 2's update (**w**) is interleaved between the test for NULL (**R**) and the `else` part of the conditional in Thread 1 (**R**).

On the right side of the figure, we see how the original program is transformed by the ToleRace runtime system. ToleRace eliminates the **RwR** conflict by creating a local copy (`local_gSript`) of the shared variable, `gScript`, when the critical section is entered, operating on that copy in the body of the critical section, and copying back the updated value when the critical section is exited. With ToleRace, Thread 2 never sees the memory state in which `gScript` is set to NULL, and hence the race never occurs.

1.1 Why Asymmetric Races?

ToleRace allows programmers to reason locally about the correctness of their critical sections. Normally local reasoning cannot be applied when considering the correctness of programs with shared variables. Components that are locally correct (e.g., use locks to protect a shared variable) are made incorrect by arbitrary code somewhere else in the application. With large development teams, it is typical for most of the code in an application to be outside the direct control of a particular programmer. What is worse, the source code of a library that contains a concurrency error may not be available at all. In such cases, the client of an incorrect library may be forced to program around the error in an ad hoc way. ToleRace allows programmers to detect and respond to external concurrency errors in a structured and principled way with no changes to external code.

Races occur infrequently. As a result, attempts to detect races dynamically induce significant overhead on all executions, partly because they attempt to precisely pinpoint *both* threads that are interacting incorrectly. Surprisingly, ToleRace manages concurrency errors without requiring knowledge of the thread that caused the error, reducing the execution overhead of maintaining precise information.

ToleRace detects *asymmetric races*, a class of races caused by two threads accessing a shared variable, one that correctly acquires and releases a lock (thus creating a critical section) and another that does not. While this prevents ToleRace from addressing symmetric races where neither thread uses a lock to protect the shared variable, asymmetric races are common in software development for the following reasons. First, most code is written correctly—in many cases local reasoning about concurrency, including taking proper locks, has been done correctly, leaving many remaining concurrency errors as asymmetric races.

Second, asymmetric races can be caused when software evolves and assumptions are invalidated. For example, code might be developed with the assumption that application initialization never occurs in a multi-threaded context. However, new code might be introduced (e.g., a second start-up thread) that violates the original invariant. Another example occurs when a library is written assuming a single-threaded environment, and later the requirements change and multiple threads are used. An expedient response to this change in requirements is to demand that clients of the library wrap calls to the library API, acquiring locks before entry and releasing them on exit. Because this solution requires that all clients of the library be changed, races can be introduced when clients are inadvertently left unmodified.

2. PRELIMINARIES

A lock x_V is typically associated with a group of shared variables V. The purpose of x_V is to enforce exclusive access by a specific thread to any variable in V.

DEFINITION 1. ***Critical section*** *is a subgraph G of the control-flow graph $G \subset \mathcal{G}(T)$ of a specific thread T such that*

all entry points to G acquire a specific lock x_V and all exit points out of G release x_V.

Here, T enters the critical section to gain exclusive access to V. We allow arbitrary overlap or nesting of critical sections over distinct locks within the same thread. We denote as $l()$ and $u()$ the atomic functions that acquire and release a specific lock respectively. Next, we denote as $r()$ and $w()$ two functions that read from and write to a specific variable. We consider cases when a single variable is protected and accessed, unless stated otherwise. We use **l**, **u**, **r**, and **w** to denote the fundamental functions over that specific variable. Finally, we use **x** to denote a "*don't care*" function which can be either a read or a write.

The control-flow graph of a critical section may create numerous different sequences of reads and writes depending on the machine state. With no loss of generality, we use a notation where a sequence of reads and writes is presented using an unfolded format (sequential, non-conditional). We first denote a sequence of at least one read as $\mathbf{r}^+$ and a sequence of a non-negative number of reads as $\mathbf{r}^*$, thus we have $\mathbf{r}^+ \equiv \mathbf{r}\mathbf{r}^*$. Operators $*$ and $+$ are equally defined for writes. For a specific thread T_1 we define the sequence of critical operations using the above operators and fundamental functions. For example $T_1 = [\mathbf{l}_1\mathbf{r}_{11}\mathbf{w}_{11}\mathbf{r}^+_{12}\mathbf{u}_1]$ denotes a thread that first locks in a variable then reads and writes exactly once, followed by at least one read before it unlocks this variable. The first digit in the operation index denotes the thread index, i.e., T_1, and the second distinguishes between sequences of operations of the same type. We denote one possible interleaved execution of critical operations of two threads $T_1 = [\mathbf{l}_1\mathbf{r}_{11}\mathbf{w}_{11}\mathbf{r}^+_{12}\mathbf{u}_1]$ and $T_2 = [\mathbf{w}_2]$ as the following sequence $S = \{\mathbf{l}_1\mathbf{r}_{11}\mathbf{w}_{11}\mathbf{w}_2\mathbf{r}^+_{12}\mathbf{u}_1\}$. Sequence S specifies that the write from the second thread occurred after the write in the first thread to result in a race condition.

2.1 Characterizing Asymmetric Races

To characterize asymmetric races, we exhaustively consider all interleavings between operations in a correctly synchronized thread and a second, unsynchronized thread. We then reduce the interleavings that result in races into four classes and consider how ToleRace handles each class. We assume a programming model with two types of threads:

- a **safe thread** consists of a single critical section, and
- a **malicious thread** does not have a critical section although it could access a shared variable.

DEFINITION 2. *A **race condition** represents any one of all possible execution interleavings of a set of threads $\mathbb{T} = \{T_1 \ldots T_N\}$ where at least one of the threads in $\mathbb{T}$ is malicious and at least one is safe, such that the final computation state after all threads have executed does not correspond to the case when all safe threads in $\mathbb{T}$ have executed atomically with sequential memory consistency.*

Our definition is agnostic to the execution order of individual threads. Thus, we assume that the programmer intended that the threads can be executed in any order, as long as they execute atomically with respect to their critical sections.

A thread (safe or malicious) in $\mathbb{T}$ could execute but not affect the program computation state. In this case, we informally relax Definition 2 to accept execution schedules where a subset of threads from $\mathbb{T}$ does not execute as correct.

Using this definition, the sequence S presented earlier in Section 2, represents a race because T_1 does not execute atomically due to the interleaving of $\mathbf{w}_2$. Sequences $\{\mathbf{w}_2\mathbf{l}_1\mathbf{r}_{11}\mathbf{w}_{11}\mathbf{r}^+_{12}\mathbf{u}_1\}$ or $\{\mathbf{l}_1\mathbf{r}_{11}\mathbf{w}_{11}\mathbf{r}^+_{12}\mathbf{w}_2\mathbf{u}_1\}$ are not considered races as they correspond to the atomic execution of the safe thread T_1.

For the purposes of this discussion, we assume the hardware supports sequential memory consistency semantics [20] and that ToleRace preserves these semantics. We relax these restrictions in Subsection 3.3 by considering the general case of races occurring within critical sections that protect multiple variables as well as critical sections that are nested and/or overlapped.

To understand the ways in which the safe and malicious threads can interact, we exhaustively explore all interleavings where the malicious thread T_2 executes between operations in the safe thread T_1. To simplify the analysis, we note there are only three ways in which a sequence of operations by a single thread can interact with a single variable: by reading it only ($\mathbf{r}^+$), by setting its value regardless of its prior ($\mathbf{w}\mathbf{x}^*$), and by setting its value based upon its prior ($\mathbf{r}^+\mathbf{w}\mathbf{x}^*$). Operations that follow a write by a particular thread are important semantically but do not affect the inter-thread interactions. Also note that **rw** could occur in two versions: (i) **w** is dependant upon the value retrieved by **r** and (ii) **w** is not dependant upon the value retrieved by **r**. Sequences where (ii) is true, could be analyzed as independent manifestations of two sequences of type $\mathbf{r}^+$ and $\mathbf{w}\mathbf{x}^*$. Sequences where (i) is true, demand special attention; thus, in the remainder of this paper, when we specify a sequence **rw** issued by the same thread we assume (i).

Table 1 tabulates all possible interactions between two threads: a safe thread T_1 and a malicious thread T_2. The safe thread is improperly intercepted by T_2 at a position that slices the operations of T_1 into two parts T'_1 and T''_1. The table evaluates the outcome of this interaction exhaustively. Symbols T and F specify that a race has and has not occurred respectively. Symbol CT points to the fact that a race occurs conditionally – the condition is that T_2 alters at least one more live variable in the program computation state. This variable could be local to the thread. We derive the following classification theorem from Table 1.

THEOREM 1. ***Race condition cases.*** *A race between two threads occurs due to one of the following conditions:*

I $\mathbf{XwR} \equiv \{\mathbf{l}_1\mathbf{x}^+_1\mathbf{w}_2\mathbf{x}^*_2\mathbf{r}_1\mathbf{u}_1\}$. *This case specifies that for any sequence of operations by T_2 that starts with a write, is followed by a read by T_1, a race will occur.*

II $\mathbf{WrW} \equiv \{\mathbf{l}_1\mathbf{r}^*_{11}\mathbf{w}_{11}\mathbf{x}^*_1\mathbf{r}^+_2\mathbf{r}^*_{12}\mathbf{w}_{12}\mathbf{u}_1\}$. *This case specifies that any sequence of reads by T_2 when placed in-between two writes by T_1 results in a race.*

III $\mathbf{RXwW} \equiv \{\mathbf{l}_1\mathbf{r}_1\mathbf{x}^*_1\mathbf{w}_2\mathbf{x}^*_2\mathbf{w}_1\mathbf{u}_1\}$. *When T_1 starts with a read followed by an arbitrary sequence of operations, and T_2 executes any sequence of operations that starts with a write just before T_1 writes back to this variable, a race will occur. An additional constraint is that T_2 alters at least one more live variable in the program's computation state besides the variable protected by T_1. This variable does not need to be shared.*

IV $\mathbf{XrwX} \equiv \{\mathbf{l}_1\mathbf{x}^+_{11}\mathbf{r}^+_2\mathbf{w}_2\mathbf{x}^*_2\mathbf{x}_{12}\mathbf{u}_1\}$. *This case specifies that any sequence that starts by a write based upon a*

Operation interleaving				Operation interleaving				Operation interleaving			
T_1'	T_2	T_1''	Race	T_1'	T_2	T_1''	Race	T_1'	T_2	T_1''	Race
$\mathbf{r}^+$	$\mathbf{r}^+$	$\mathbf{r}^+$	F	$\mathbf{r}^+$	$\mathbf{wx}^*$	$\mathbf{r}^+$	T ✓	$\mathbf{r}^+$	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{r}^+$	T ✓
$\mathbf{r}^+$	$\mathbf{r}^+$	$\mathbf{wx}^*$	F	$\mathbf{r}^+$	$\mathbf{wx}^*$	$\mathbf{wx}^*$	CT ✓	$\mathbf{r}^+$	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{wx}^*$	CT
$\mathbf{r}^+$	$\mathbf{r}^+$	$\mathbf{r}^+\mathbf{wx}^*$	F	$\mathbf{r}^+$	$\mathbf{wx}^*$	$\mathbf{r}^+\mathbf{wx}^*$	T ✓	$\mathbf{r}^+$	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{r}^+\mathbf{wx}^*$	T
$\mathbf{wx}^*$	$\mathbf{r}^+$	$\mathbf{r}^+$	F	$\mathbf{wx}^*$	$\mathbf{wx}^*$	$\mathbf{r}^+$	T ✓	$\mathbf{wx}^*$	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{r}^+$	T ✓
$\mathbf{wx}^*$	$\mathbf{r}^+$	$\mathbf{wx}^*$	T ✓	$\mathbf{wx}^*$	$\mathbf{wx}^*$	$\mathbf{wx}^*$	F	$\mathbf{wx}^*$	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{wx}^*$	CT ✓
$\mathbf{wx}^*$	$\mathbf{r}^+$	$\mathbf{r}^+\mathbf{wx}^*$	T ✓	$\mathbf{wx}^*$	$\mathbf{wx}^*$	$\mathbf{r}^+\mathbf{wx}^*$	T ✓	$\mathbf{wx}^*$	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{r}^+\mathbf{wx}^*$	T ✓
$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{r}^+$	$\mathbf{r}^+$	F	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{wx}^*$	$\mathbf{r}^+$	T ✓	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{r}^+$	T
$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{r}^+$	$\mathbf{wx}^*$	T ✓	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{wx}^*$	$\mathbf{wx}^*$	CT ✓	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{wx}^*$	CT
$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{r}^+$	$\mathbf{r}^+\mathbf{wx}^*$	T ✓	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{wx}^*$	$\mathbf{r}^+\mathbf{wx}^*$	T ✓	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{r}^+\mathbf{wx}^*$	$\mathbf{r}^+\mathbf{wx}^*$	T

Table 1: Tabulating classes of race instances. Column marked "Race" denotes that a schedule $T_1'T_2T_1''$ results in a race. Symbols T and F specify that a race has and has not occurred respectively. Symbol CT points to the fact that a race occurs conditionally. The conditions are outlined in Theorem 1. Symbol ✓identifies whether the ToleRace oracle tolerates a specific atomic race type.

prior by T_2 causes a races when interspersed in-between any two operations of T_1. This statement is conditional in certain special cases. A special subset of **XrwX**, *denoted as $R_2 \equiv \{\mathbf{l}_1\mathbf{w}_{11}\mathbf{x}_1^*\mathbf{r}_2^+\mathbf{w}_2\mathbf{x}_2^*\mathbf{w}_{12}\mathbf{u}_1\}$, is an indirect race only if T_2 alters at least one live variable $u \neq v$ in the program's computation state as $u = f(v)$, where $f()$ is an arbitrary non-constant function and v is the variable protected by T_1. For sequences in $\{\mathbf{l}_1\mathbf{x}_1^+\mathbf{r}_2^+\mathbf{w}_2\mathbf{x}_2^*\mathbf{w}_1\mathbf{u}_1\} - R_2$, an additional constraint is that T_2 alters at least one more live variable in the program's computation state besides v.*

With no effect on the generality of the theorem, in all sequences we assume that the last operation in T_1 which completes the race condition, is the last operation in the critical section.

Proof. Straightforward by combining cases from Table 1. We analyze the end-sequence $\mathbf{r}_2^+\mathbf{w}_2\mathbf{x}_2^*\mathbf{w}_{12}$ in more detail. Here we have two sub-cases:

- $R_1 \equiv \{\mathbf{l}_1\mathbf{r}_1\mathbf{x}_1^*\mathbf{r}_2^+\mathbf{w}_2\mathbf{x}_2^*\mathbf{w}_{12}\mathbf{u}_1\}$, which is not a direct race as it corresponds to T_2 never executing with respect to the protected variable. In case T_2 alters any other live variable in the program's computation state, this sequence is an indirect race.

- $R_2 \equiv \{\mathbf{l}_1\mathbf{w}_{11}\mathbf{x}_1^*\mathbf{r}_2^+\mathbf{w}_2\mathbf{x}_2^*\mathbf{w}_{12}\mathbf{u}_1\}$ is an interesting case. It is not a direct race as it could correspond to T_2 never executing with respect to the protected variable v or it could correspond to the execution schedule T_2T_1. In the first case, if T_2 alters any other live variable in the program's computation state, this sequence is an indirect race. In the second case, if T_2 alters any other live variable u in the program's computation state as $u = f(v)$ where $f()$ is an arbitrary non-constant function, this sequence is also an indirect race. We accept only the second constraint, as the first is its superset.

All considered sequences cover $\mathbf{x}_{11}^+\mathbf{x}_2^+\mathbf{x}_{12} \cup \mathbf{x}_1^+\mathbf{x}_2^+\mathbf{r}_1^+\mathbf{w}_1$. □

The conditions that need to be satisfied for all races of type III and some races of type IV to occur, are relatively mild. Thus, in the remainder of this paper we assume that they are fulfilled for each malicious thread.

Theorem 2. ***Reduction of race conditions.*** *Any race condition among $K > 2$ threads can always be reduced to one of the* I-IV *cases of a race between two threads.*

Proof. (sketch) A race is a consequence of two or more threads interleaving reads and writes to alter the proper program output. Based upon Theorem 1 race types I-III occur due to a single interleaved instruction issued by a thread T_2 accessing the data protected by T_1. Once this instruction is fired, the presence of other threads $T_i, i > 2$ does not impact the occurrence of these race types. Only one race type could be launched by more than two threads, race type IV. Here the unprotected "write based upon a prior," **rw**, could be launched by more transforming the functionality that computes the value that **w** writes onto the protected variable, to be computed in a single thread, we conclude our proof. □

In the remainder of this paper we consider only race types I-IV occuring in an environment with two concurrent threads.

3. THE TOLERACE ORACLE

The core of our approach to managing race condition cases specified in Theorem 1 is to replicate the protected shared state so that the thread that acquires a lock on the shared state has an exclusive copy (see Figure 2). This thread continues reading from and/or writing to this copy until it releases the lock. When the lock is released, ToleRace provides a family of software/hardware mechanisms to determine which race has occurred, with possible outcomes ranging from tolerating the race completely, to reporting that a race has occurred, to executing a programmer-specific handler when an intolerable race is detected. In hardware assisted implementations, suspended thread execution is also possible with ToleRace. In this section, we evaluate the effect of ToleRace on the race cases described in Theorem 1 assuming an oracle determines which case has occurred.

Initialization and finalization: We assume that the binding of locks (x_V) to shared variables (V) is known before the critical section in T_1 is entered ("lock – shared variable" associations are input to ToleRace and is discussed in detail in Subsection 3.4) and that storage for two additional copies (V', V'') of variable V has been allocated. After the lock is released, storage for the two additional copies is deallocated.

Lock (Entry): When lock x_V is acquired in T_1, we copy V to V' and V'' ($V'' = V' = V$). Because multi-variable

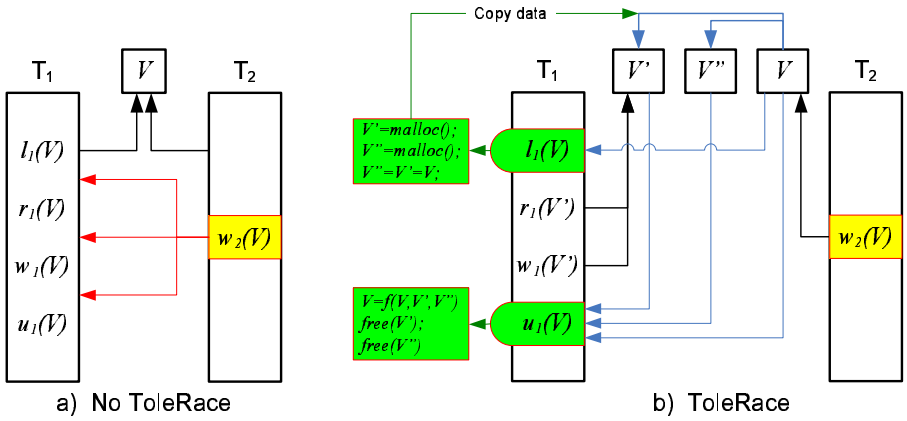

Figure 2: How ToleRace uses two additional copies of a variable to tolerate races.

copying is not atomic without hardware support, copying may introduce additional races, which we discuss below.

Reads and Writes inside the Critical Section: ToleRace alters all instructions in the critical section of T_1 to use V' instead of V. Thus, V' is a local copy of V for T_1 which cannot be accessed by other threads due to a race.[2] All other threads such as T_2 are unchanged and continue using V for all accesses. Copy V'' is not accessed until T_1 reaches the exit of the critical section.

Unlock (Exit): When T_1 exits the critical section by releasing the acquired lock, ToleRace analyzes the contents of V', the original value V'', and the value V that could have been altered by other threads as a consequence of a race. Depending on the relationship of values in $\{V, V', V''\}$ and knowledge about the specific case in Theorem 1 that has occurred, ToleRace deploys a resolution function $V = f(V, V', V'')$ that defines the value of V after T_1 is executed.

3.1 Effectiveness of the ToleRace Oracle

Combining the mechanism outlined above with the exhaustive interleavings enumerated in Table 1, we reason about which cases ToleRace will tolerate. Assuming perfect knowledge of the specific case of race that has occurred, Table 2 summarizes the definition of f and indicates the cases that ToleRace correctly tolerates. Table 1 also contains this information in expanded form, where the symbol ✓ denotes whether the ToleRace oracle tolerates a specific interleaving.

Race type		$V = V''$	$f(V, V', V'')$	Tol?	π
I	**XwR**	false	V	true	T_1T_2
II	**WrW**	true	V'	true	T_2T_1
III	**RXwW**	false	V	true	T_1T_2
IVA	**RrwR**	false	V	true	T_1T_2
IVB	**WrwX**	false	V'	true	T_2T_1
IVC	**RrwX**	false	custom $f'()$	false	N/A

Table 2: Tabulating the outcome for $f()$ for each race type.

Because ToleRace can tolerate only some races of type IV, in Table 2 we subdivide this case into three sub-cases:

[2]Other threads could potentially access V' via an erroneous pointer, however ToleRace does not address such bugs by default although it may point to them.

IVA $\mathbf{RrwR} \equiv \{\mathbf{l}_1\mathbf{r}_{11}^+\mathbf{r}_2^+\mathbf{w}_2\mathbf{x}_2^*\mathbf{r}_{12}\mathbf{u}_1\}$,

IVB $\mathbf{WrwX} \equiv \{\mathbf{l}_1\mathbf{w}_1\mathbf{x}_{11}^*\mathbf{r}_2^+\mathbf{w}_2\mathbf{x}_2^*\mathbf{x}_{12}\mathbf{u}_1\}$, and

IVC $\mathbf{RrwX} \equiv \mathbf{XrwX} - \{\mathbf{RrwR} \cup \mathbf{WrwX}\}$.

The first column in Table 2 lists the race type based upon the classification from Theorem 1, the second column specifies whether V is equal to V'' at the point when $f()$ is called, the third column shows a resolution function $f()$ that allows ToleRace to tolerate the race, the fourth column indicates whether $f()$ provably succeeds in tolerating the race, and the fifth presents π, the schedule of threads that ToleRace's result represents. Table 2 shows that the ToleRace oracle tolerates all race cases except sequences that belong to **RrwX** with a resolution function $f()$, defined by Table 2.

Based on Table 1, we introduce a simplified close-form resolution function:

$$V = f(V, V', V'') \equiv \begin{array}{ll} V', & V = V'' \\ V, & V \neq V'' \end{array}, \qquad (1)$$

which enables tolerance of all race types except IVB and IVC.

For races of type **RrwX**, the interleaving of reads and writes from T_2 breaks the program's sequential memory consistency. Here, T_1 and the interleaved part of T_2 both read the value of the shared variable before entering the critical section, execute in parallel, and then join at the exit of the critical section of T_1 (see Figure 3). In this case, either schedule T_1T_2 or T_2T_1 results in the read of the second thread executing seeing the value of the variable written by the first thread. Figure 3 illustrates all three outcomes.

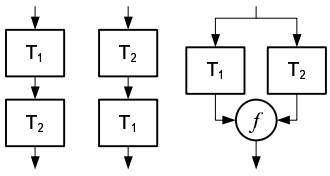

Figure 3: Atomic execution order enabled by ToleRace. Schedules T_1T_2 and T_2T_1 are correct, whereas the parallel execution of T_1 and T_2 is a race.

It is interesting to notice that if **w** does not depend upon **r** in **RrwX**, then ToleRace tolerates this case such that **r** effectively executes before T_1 and **w** executes afterwards, while the execution of T_1 is effectively uninterrupted.

In summary, surprisingly Table 2 shows that with correct knowledge of what case of race has occurred, ToleRace correctly tolerates all but one class of races (IVc). Section 4 outlines simple techniques how software and hardware implementations could enable ToleRace to distinguish between race subclasses. Depending on the final disposition of values V, V', and V'', we denote the two possible sets of outcomes as A and B (see Theorems 3 and 4) and prove the results below.

THEOREM 3. *A – **Equivalence.** $V = V''$ signals race-free execution of T_1.*

PROOF. Here, two cases could happen with respect to an interleaved thread T_2:

- T_2 never wrote to V during the execution of T_1 and
- T_2 had a sequence of arbitrary operations $\mathbf{x}_2^+$ before finally writing V'' to V.

In the former case, only a **WrW** race type could have possibly happened. From Table 2, one can verify that ToleRace fully tolerates such races. In the latter case, a race could occur only due to a race of type **RrwX** (see Table 2). An interfering malicious sequence $\mathbf{r}_2^+\mathbf{w}_2\mathbf{x}_2^*$ that initiates a **RrwX**-type race with the last write in $\mathbf{w}_2\mathbf{x}_2^*$ that resets V to its original value V'', is surprisingly tolerated by ToleRace with a correct execution schedule T_2T_1. Because T_2 sets in this case the value of the shared variable V to its original value, we conclude that the schedule T_2T_1 executes race-free. In essence, the effect of executing T_1 on the computation state is the same whether T_2 executed or not. □

When ToleRace detects $V = V''$ it exits the critical section in T_1 by setting $V = f(V, V', V'') = V'$, de-allocating all replicated data, and unlocking x_V. If the construct:

```
if (vii == v) v = vi;
```

is not atomic, a race of type III or IV could occur by an interleaved $w_2(V)$. We denote this post-execution ToleRace-induced race as A_P. Thus, a race could occur under ToleRace if $V = V''$ only in the unlikely subcase A_P.

If $V \neq V''$, ToleRace concludes that at least one interleaved $w_2(V)$ has occurred while T_1 executed its critical section. From Theorem 1, we see that several cases are possible. In cases I and IVa, thread T_2's write to V has occurred after all writes (if any) in thread T_1, hence this case is correctly tolerated by leaving V as it is, resulting in the execution order T_1T_2. In case III, thread T_1's final memory operation is to write to V and thread T_2 does not read V before writing it. As a result the execution order T_1T_2 does not require that thread T_2 start executing after thread T_1 writes the value of V, allowing the resolution of leaving V the same on exit to provide the semantics of T_1T_2.

This condition does not necessarily mean that an actual race occurred, e.g., sequence $\mathbf{Xw} \equiv \{\mathbf{l}_1\mathbf{x}_1^*\mathbf{w}_2\mathbf{u}_1\}$ is not a race, yet results in $V \neq V''$. Since another interleaving of the same threads could result in a race (e.g., **RXwW**), it is desirable to report these incidents to the programmer.

Case **RrwX** cannot be tolerated by either of the outcomes that resolve cases I–IVb, and as a result, ToleRace will either raise an exception indicating that a race has been detected, or execute a programmer-defined resolution function, $f = f'$, when a semantically sound resolution function is provided. An example of an efficient $f'()$ that tolerates a **RrwX** race is shown in Figure 4. By knowing that the variable `gameScore` participates as an operation destination only in additions and subtractions of constants, the programmer uses a custom resolution upon a race detection to tolerate **RrwX** races over `gameScore`. In the example, a system without ToleRace exposes the erroneous nature of a specific race instance.

Again, if the construct:

```
if (vii == v) v = f(v,vi,vii);
```

is not atomic, a race of type III or IV could occur by an interleaved $w_2(V)$. We denote this post-execution ToleRace-induced race as B_P.

3.2 Properties of the ToleRace Oracle

To refine the claims related to ToleRace, we introduce two definitions of dynamic race detection. In both cases the detector identifies dynamically a specific instance of interleaved execution of a thread T_2 with a lock-protected thread T_1. The instance is described as an execution of a sequence $\mathbf{x}_2^+$ of consecutive instructions of T_2 in-between two consecutive instructions in T_1.

DEFINITION 3. *A **hard race detector** reports $\mathbf{x}_2^+$ as a race if at least one of all possible instances of execution interleaving of $\mathbf{x}_2^+$ with T_1, could cause a race.*

DEFINITION 4. *A **soft race detector** reports $\mathbf{x}_2^+$ as a race if and only if the detected interleaving instance of $\mathbf{x}_2^+$ with T_1 causes a race.*

A soft race detector is more precise, detecting only those interleavings that result in an actual race. In deployed applications, this is a more desirable form of race detection (as it ignores benign races), but is also considerably more difficult to implement. A hard race detector only detects interleavings that have the potential to cause a race, but will report races in specific interleavings that did not actually exhibit a race. A hard race detector is of greater value in a testing or debugging environment, where testers need to know about the existence of a race even when many interleavings of the race are benign, so that they can fix the code to eliminate the possibility of the race altogether.

THEOREM 4. *B – **Discrepancy.** ToleRace is a deterministic hard detector of all races that are not tolerated already by the platform.*

PROOF. The only race type not tolerated by ToleRace, is **RrwX** (see Table 2). The last write in $\mathbf{w}_2\mathbf{x}_2^*$ can result in $V \neq V''$ in which case the race is detected, or $V = V''$ in which case the race is tolerated (see Theorem 3). □

THEOREM 5. ***False Positives.** ToleRace reports false positives with respect to the hard race detection oracle in the case when: $\boldsymbol{wWwWw} \equiv \{\mathbf{l}_1\mathbf{w}_{11}^+\mathbf{w}_2^+\mathbf{w}_{12}^+\mathbf{u}_1\} \cup \{\mathbf{l}_1\mathbf{w}_1^+\mathbf{w}_2^+\mathbf{u}_1\} \cup \{\mathbf{l}_1\mathbf{w}_2^+\mathbf{w}_1^+\mathbf{u}_1\}$.*

PROOF. ToleRace reports a race only when $V \neq V''$. This can only happen if T_2 writes a new value to V. The only case when T_2 writes to V in-between two operations over V by T_1 and a race could not occur, is $\mathbf{WXwW} \equiv$

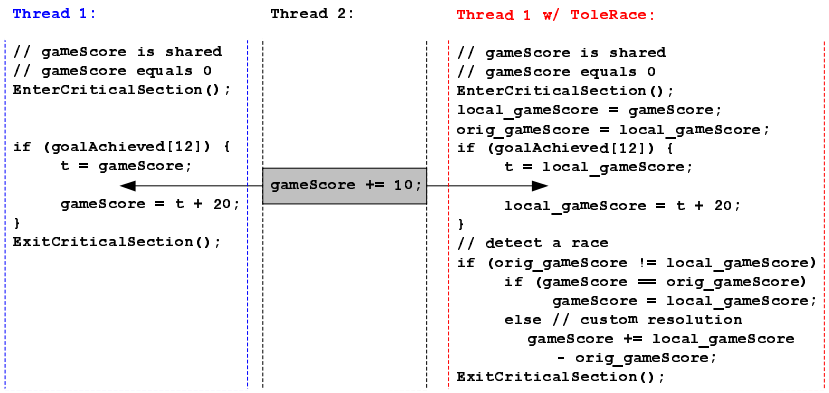

Figure 4: An example of how program semantics can define a custom resolution function f' that could tolerate RrwX races. The programmer knows the prior that all writes in the program to the shared variable are only additions and subtractions of constants.

$\{\mathbf{l}_1\mathbf{w}_1\mathbf{x}_1^*\mathbf{w}_2\mathbf{x}_2^*\mathbf{w}_1\mathbf{u}_1\}$ (see proof of Theorem 1). From the perspective of a hard race detector, if there were any reads in $\mathbf{x}_1^*$ or $\mathbf{x}_2^*$, there could be a possible permutation of operations in T_1 and T_2 that could lead to a race. Thus, we are left with $\{\mathbf{l}_1\mathbf{w}_{11}^+\mathbf{w}_2^+\mathbf{w}_{12}^+\mathbf{u}_1\}$ as one pool of cases where ToleRace reports false positives. The remaining cases in **wWwWw** represent the situations in which T_2 writes to V before T_1 writes to V' and/or T_2 writes to V after the last write of T_1 to V'. The correct execution permutations in these cases are T_2T_1 and T_1T_2 respectively. □

From Theorem 5 one can conclude that the lock and unlock operations over V should be placed as close as possible to the first and last access to V respectively within the corresponding critical section. By doing so, the programmer reduces the likelihood that another thread places a write that could trigger a false positive by ToleRace.

For a software-only implementation, we conclude:

THEOREM 6. ***False Negatives.***[3] *ToleRace identifies* ***all*** *race instances in protected threads except* $A_P \cup B_P$.

Thus, software-only ToleRace would aim to provide a probabilistic race prevention that significantly reduces the likelihood of races in programs. As a rough estimate, in an execution with no stalls if the number of instructions in T_1 is K times greater than the number of instructions in $f()$, we expect a $\mathcal{O}(K)$-fold lower likelihood of a race occurrence.

3.3 Relaxing the Assumptions

In this section we address two strong assumptions introduced earlier: we allow multiple critical sections within a thread, we allow their nesting and overlap, then finally we allow that one lock protects a group of variables as opposed to a single variable.

We differentiate between two modes of thread execution:

- **sequentially consistent execution** – where instructions in the thread are executed in the order specified by the program [20], and
- **out-of-order execution** – where instructions in the thread could be executed in an arbitrary order that does not alter program functionality [18, 30].

[3]No proof provided. Straightforward from the discussions in Subsection 3.

First, we allow nested and interleaved critical sections within a single thread, where each critical section protects a single variable.

LEMMA 1. ***Sequential Inconsistency - I.*** *In the general case of multi-threaded programs with threads that contain multiple critical sections per thread such that each critical section protects a distinct shared variable, the ToleRace oracle does not enable sequentially consistent execution.*

PROOF. (by an example) Consider two nested critical sections within a single thread T_1: $\mathbf{l}_1(x_P)\mathbf{l}_1(x_Q)\mathbf{w}_{11}(P)\mathbf{r}_{11}(Q)\diamond \mathbf{r}_{12}(Q)\mathbf{w}_{12}(P)\mathbf{u}_1(x_Q)\mathbf{u}_1(x_P)$, where $\diamond$ denotes the location where a malicious thread T_2 interleaves the following sequence of instructions $\mathbf{w}_2(Q)\mathbf{r}_2(P)$. ToleRace will tolerate both races from the perspective of out-of-order execution, however based upon Table 2 $\mathbf{w}_2(Q)$ will execute after $\mathbf{u}_1(Q)$ and $\mathbf{r}_2(P)$ will execute before $\mathbf{l}_1(P)$, breaking their program-specified order of execution. A similar example could be constructed for two overlapping critical sections. □

Lemma 1 introduces an interesting paradox. Consider the proof of this lemma. Although $\mathbf{w}_2(Q)\mathbf{r}_2(P)$ in T_2 actually executed in this order interleaved with instructions in T_1, from the perspective of the overall execution time-line due to ToleRace the values in P and Q after $\mathbf{u}_1(x_P)$ executed, are such as if $\mathbf{w}_2(Q)$ executed <u>after</u> $\mathbf{r}_2(P)$.

Next, assume that a single lock, i.e., critical section, protects a set of M variables $\mathbb{V} = \{V_1, \ldots, V_M\}$ under the original assumption that one thread represents a single critical section.

LEMMA 2. ***Sequential Inconsistency - II.*** *In the general case of multi-threaded programs with threads that contain a single critical section which protects a set of shared variables, the ToleRace oracle does not enable sequentially consistent execution.*

PROOF. (by an example) Consider a critical section over a set of two variables P and Q with the following order of instructions $\mathbf{l}_1(x_{PQ})\mathbf{w}_{11}(P)\mathbf{r}_{11}(Q)\diamond\mathbf{r}_{12}(Q)\mathbf{w}_{12}(P)\mathbf{u}_1(x_{PQ})$, where $\diamond$ denotes the location where a malicious thread interleaves the following sequence of instructions $\mathbf{w}_2(Q)\mathbf{r}_2(P)$. ToleRace will tolerate both races from the perspective of out-of-order execution, however based upon Table 2 $\mathbf{w}_2(Q)$ will execute after $\mathbf{u}_1(Q)$ and $\mathbf{r}_2(P)$ will execute before $\mathbf{l}_1(P)$, breaking their program-specified order of execution. □

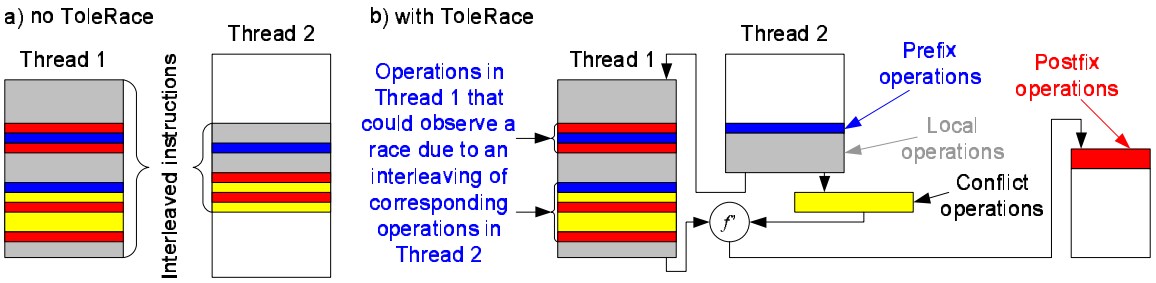

Figure 5: Out-of-order execution in ToleRace for a multivariable critical section. The data flow of the malicious thread is partitioned into mutually independent subgraphs. Subgraphs are denoted as rectangles with distinct colors. Each subgraph, based upon internal variable dependencies, is scheduled either as a prefix, postfix, or a parallel, conflicting execution construct. Subfigures depict the execution order in the case when ToleRace is (b) and is not (a) used.

In general, if a malicious thread T_2 which operates over $\mathbb{V}$ is interleaved with a safe thread T_1 that properly protects the access to $\mathbb{V}$ and uses ToleRace to handle races, then individual operations in T_2 will get effectively rescheduled and executed around and in parallel with T_1. Before we describe the rescheduling, we classify the variables in $\mathbb{V}$ via a data flow graph [9]. We partition the data flow graph of a particular execution instance of T_2 into P disconnected subgraphs $\mathbb{G} = \{G_1, \ldots, G_P\}$, where P is a variable that depends upon the data flow graph. By doing so we isolate variables in $\mathbb{V}$ that are independent from groups of other variables that are mutually dependent. Next, we merge all variables from $\mathbb{V}$ that appear in a single subgraph G_i into a single variable W_i. This step ensures that dependencies among different variables are identified as a source of potential **XrwX**-type conflicts. In an **XrwX**-type race condition, **r** and **w** could be performed on different variables, however **w** could still be dependent upon the result of **r** both directly or indirectly via a variable that is local to T_2. Then, for each subgraph G_i we identify the sequence of accesses to W_i. Due to ToleRace all operations in G_i then are rescheduled and executed based upon the sequence of writes and reads to W_i as follows:

- **prefix operations** – **WrW** and **RrwX** such that $(\forall V_j \in W_i) V_j = V_j''$ (see Theorem 3) – all instructions in G_i are effectively executed before T_1.
- **postfix operations** – **XwR**, **RXwW**, and **RrwR** – all instructions in G_i are effectively executed after T_1.
- **conflict operations** – **RrwX** – all instructions in G_i are effectively executed in parallel with operations in T_1 and require a custom resolution function $f = f'$.

Subgraphs with operations that affect only local variables of the malicious thread can be effectively scheduled in an arbitrary way with a preserved consistency of the computational state. These operations cannot cause a race with or without ToleRace. The out-of-order execution of subgraphs is illustrated using Figure 5.

Now we augment the definition of the ToleRace approach to address the case when a single variable V is protected using two different locks, x_A and x_B such that $V \subset A \cap B$, and when these locks are used by two critical sections that overlap, i.e., $\{\mathbf{l}(x_A)\mathbf{l}(x_B)\mathbf{u}(x_A)\mathbf{u}(x_B)\}$. Then ToleRace copies V at the first lock $\mathbf{l}(x_A)$ and resolves this variable at the last unlock $\mathbf{u}(x_B)$.

THEOREM 7. ***Out-of-order Execution.*** *In the general case of multi-threaded programs with threads that contain multi-variable critical sections that can intersect via nesting or overlapping, the ToleRace oracle does not maintain sequential memory consistency. ToleRace does not introduce new conflicts among the interleaved operations of the malicious thread that are already tolerated (prefix and postfix) at the level of an individual critical section of the safe thread.*

PROOF. (sketch) Lemmas 1 and 2 prove that ToleRace in the most general case of thread organization does not preserve sequential memory consistency. ToleRace does reorder the sequence of executed instructions even in the case when a processor executes them in-order.

In ToleRace, the protection lifetime of a single variable V spans over all overlapping critical sections that could use distinct locks to protect it. Therefore, operations in the malicious thread that access V and that are tolerated by ToleRace, are effectively rescheduled either before the first lock or after the last unlock in the set of overlapping critical sections. As they are shifted outside of the range of protection they cannot create any new conflicts that are not allowed by the program specification. □

Theorem 7 proves that ToleRace introduces only a mild restriction to the execution environment. As in-order execution is important only from the perspective of the programmer while debugging, due to the out-of-order execution that ToleRace provides, in-order symbolic debugging on such systems could be difficult to achieve [15]. Thus, in ToleRace a debugger should provide the user with the observability and controllability of all copies of the original shared variable. This means that ToleRace is not transparent to the existing program development process. However, in-order execution is of little importance for testing or executing released software. Based upon that premise most modern processors enable out-of-order execution to take advantage of the program's instruction level parallelism.

3.4 Associating locks and shared data

We assume that there is a known static association between locks and shared data fed as input to ToleRace. In the programmer's mind such a binding must exist at some level in order to write correct code. This relationship can be determined automatically using existing static analysis techniques (e.g., see Naik et al. [24]) or in cases where static

analysis fails, programmer annotation may be required (e.g., SAL [12]). The application of these tools is orthogonal to ToleRace; it can be done using arbitrary static analysis tools or manual effort. Over-approximating the data protected by a lock will not result in incorrect behavior with ToleRace; the program will simply execute less efficiently. When static analysis methods improve, the inefficiency introduced by over-approximation decreases.

4. IMPLEMENTING TOLERACE

Conservative approximations of the ToleRace oracle can be implemented using a variety of software-only and combined software/hardware techniques. In this paper, we have focused on describing and proving properties of the ToleRace oracle. In the remaining limited space, we outline possible implementation approaches.

4.1 Software-Only

Any software approach to race detection that requires every program read and write be modified incurs significant overhead in practice. Here, our purpose is to make the case that a much lower overhead implementation of ToleRace is possible, based on modifying only code inside the critical section of the safe thread. With such an implementation, we are able to detect and correctly tolerate race types $\mathbb{A} = \{\mathbf{RwR} \equiv \{\mathbf{l}_1\mathbf{r}_{11}^{+}\mathbf{w}_2\mathbf{x}_2^{*}\mathbf{r}_{12}\mathbf{u}_1\} \subset \mathrm{I}, \mathrm{II}, \mathrm{IV_A}\}$. In each of these cases, no knowledge of the malicious threads read behavior is needed to correctly identify and tolerate the race. Unfortunately, cases III, IVB, and IVC, cannot be distinguished without knowing more about the behavior of the malicious thread, specifically whether it performs a read of the shared variable on which a later write is dependent. If an efficient implementation for detecting occurrences of race types IVB and IVC existed, ToleRace would tolerate all race instances but IVB and IVC.

Thus, low-overhead software-only implementation of ToleRace would tolerate race cases $\mathbb{A}$ and serve as a reliable detector of the remaining race classes. Since modern architectures typically do not support atomic multi-variable test-and-set instructions, we conclude that software-only ToleRace would only probabilistically reduce race occurrences due to the existence of false negatives (see Theorem 6).

Despite its limits, a software-only implementation of ToleRace would be useful in debugging, testing, and in deployment. For debugging and testing, software-only ToleRace provides a lower-overhead complement to existing dynamic race detection tools (discussed below). In this mode, all conflicts detected by ToleRace could be logged and in cases where the race can be tolerated, the application can be allowed to continue. This approach facilitates long haul stress testing in which applications are run for many days to expose infrequent error conditions, including races. High overhead techniques or techniques that raise an exception the first time an error is detected are less appropriate for such testing. ToleRace's dynamic detection can also be used to filter and prioritize the results of static analysis techniques that may report numerous false positives.

ToleRace can also be used in deployed software. To further limit the performance overhead of software-only ToleRace, specific portions of the executable could be instrumented. For example, instrumentation sites can be pruned based on a particular lock, data, and region of code that are suspected to be involved in hard to reproduce races. Because certain races are difficult to correct, the final software release could be augmented at specific race-occurring locations with ToleRace to reduce the likelihood of a race occurrence. This could be done at release time or later as part of a software patch. Furthermore, if the behaviors of more common races are observed by ToleRace, a specific action to tolerate these races can be deployed via custom resolution functions.

The overhead of software-only ToleRace could be quite low when used to protect simple shared data objects, when compared to existing approaches such as lockset algorithms [29]. In our preliminary implementation, we recorded 3-20% performance overhead across a set of benchmarks with threads that aggressively access shared data. However, for complex dynamic data structures the overhead of software-only ToleRace could be excessive—one resolution to this problem is using a shared pointer to access such structures and resolve races prior to updating the structure.

4.2 Software/Hardware

The major benefit that hardware could provide in supporting ToleRace would be to lower the cost of measuring every read and write in the program, providing more data in resolving the race cases in Table 1. For example, case III could be distinguished from cases IVB and IVC if it were known that no read had occurred on the shared variable outside of the safe thread since the critical section was entered.

A more ambitious approach would detect the read-write dependency in the malicious thread that characterizes cases IVB and IVC. In both cases, the dependent **rw** over a shared variable could be established via a set of variables local to the malicious thread and we briefly sketch how this detection might be accomplished. With hardware support, a compiler could associate a "read – clean – dirty" 3-tuple of bits with each shared and all dependent local variables (based on the analyzed dataflow). The first read of a shared variable would set its "read" bit. The first write into a shared variable would set its "clean" bit. Each write into a variable that is dependent upon a variable that has its "read" bit set would set the "dirty" bit unless the "clean" bit is already set. With this support, a race over shared data with a set "dirty" bit and $V \neq V'$ belongs to IVB or IVC.

A separate detector which monitors the type of the first operation over any shared variable would distinguish among the subcases of race type IV. The hardware would signal this information to $f()$. Finally, one proposed resolution to races of type IVC (i.e., the only ones that ToleRace cannot tolerate) is to suspend execution before the first dependent write to a shared variable (for IVC race-types). For programs with infrequent races (vast majority of cases), the suspensions triggered by this hardware should cause negligible effect on resulting instruction level parallelism.

In summary, we anticipate that the proposed hardware techniques paired with atomic multi-variable copy instructions could enable race-free (if all memory accesses are monitored) execution of existing legacy code with relatively low overhead. We speculate that hardware mechanisms to support different variants of ToleRace would be less complex than transactional memory hardware (see Section 5). Implementing ToleRace in software alone, we anticipate a non-trivial performance overhead proportional to the amount of protected data and the frequency of using ToleRace for protection. To avoid large overhead, selective protection and using a common principle of accessing large shared data

structures with shared pointers should be sufficient to reduce races significantly.

5. RELATED WORK

Related race detection research includes static and dynamic approaches. Static race detection relies on program analysis and either assumes existing programming languages (e.g. Java [24]) or defines new languages with semantics that help improve the static detection of races (e.g., Cyclone [11]). Static analysis techniques face several challenges. First, because many of the techniques are based on some form of model checking [16], they are computationally expensive and issues of scalability arise. Second, the conservative and approximate nature of the analysis creates the potential for many false positives. RacerX [8] and Houdini/rcc [10] address these issues by combining traditional static analysis with heuristics and statistical ranking to identify the most probable races. One inherent drawback of static analysis for race detection is that asymmetric races can occur in contexts where the source code for the component containing the error is not available for examination.

Eraser is a dynamic race detection system based on maintaining locksets [29]. Lockset analysis discovers where programs access shared variables without properly holding locks. Experience with this approach has shown that the overhead of maintaining locksets is high and that false positives can be problematic. Subsequent approaches extend lockset analysis with happens-before analysis [2] which identifies data accesses with no implied ordering. Combining locksets with happen-before results in higher precision dynamic race detectors [7, 25, 26, 31]. Even with refinements, the execution overhead of these approaches is typically larger than a factor of two. Furthermore previous work does focuses primarily on detecting races rather than tolerating them.

The AVIO system takes a training-based approach to identify erroneous access interleavings [21]. After training to learn which interleavings are benign, they deploy runtime checking to detect malicious interleavings dynamically. The overhead of this checking is high without hardware support. Similarly to ToleRace, AVIO considers interleavings of local and remote references, and automatically filters those that cannot produce a race. Unlike AVIO, ToleRace can be implemented with a range of approaches, including a relatively low overhead software implementation.

Transactional memory systems replace lock-based critical sections with fine-grain atomic transactions [17]. Transactional memory mechanisms implemented entirely in software (e.g., [1, 13, 14]), or in some combination of hardware and software (e.g., [4, 5, 6, 19, 22]), has been proposed and evaluated. Similar hardware mechanisms have also been proposed for optimistically eliminating locks in non-transactional systems [28]. While transactional memory provides many advantages, it represents only one point in a broad spectrum of approaches to building correct concurrent systems. There are distinct advantages to using ToleRace over transactional memory. First, while transactional memory is still being researched, many existing lock-based concurrent programs can benefit from ToleRace right now. Second, ToleRace represents a spectrum of mechanisms that range from a low-overhead software implementation to more complex hardware mechanisms that still represent a simplification over existing hardware transactional memory designs. Finally, ToleRace can be used in many practical ways to improve existing programs at different stages of the development cycle, including testing, deployment, and patching.

6. SUMMARY

We introduce ToleRace, a conceptually novel runtime system that uses data replication for detecting and tolerating concurrency errors in lock-based multi-threaded programs. ToleRace addresses asymmetric races, where one use of a shared variable is correctly protected with locks while other uses are not. We enumerate the set of all possible interleavings that cause races and propose methods for their handling. We focus on the theoretical aspects of the proposed oracle and review several ideas for low-cost implementations.

Acknowledgements

We thank Emery Berger, Martin Burtscher, Chen Ding, Manuel Fahndrich, Sriram Rajamani, Ganesan Ramalingam, Paruj Ratanaworabhan, and Jason Yang for their valuable feedback.

7. REFERENCES

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37, New York, NY, USA, 2006. ACM Press.

[2] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 234–243, New York, NY, USA, 1991. ACM Press.

[3] Emery D. Berger and Benjamin G. Zorn. DieHard: probabilistic memory safety for unsafe languages. *ACM SIGPLAN Notices*, 41(6):158–168, June 2006.

[4] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–358, New York, NY, USA, 2006. ACM Press.

[5] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in transactional memory virtualization. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 371–381, New York, NY, USA, 2006. ACM Press.

[6] Peter Damron, Alexandra Fedorova, and Yossi Lev. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, New York, NY, USA, 2006. ACM Press.

[7] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. In Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff, editors, *FATES/RV*, volume 4262 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2006.

[8] Dawson R. Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 237–252, 2003.

[9] Gerald Estrin and Rein Turn. Automatic assignment of computations in a variable structure computer system. *IEEE Transactions on Electronic Computers*, EC-12(6):755–773, 1963.

[10] Cormac Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *PASTE '01: Proceedings*

of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pages 90–96, New York, NY, USA, 2001. ACM Press.

[11] Dan Grossman. Type-safe multithreading in cyclone. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 13–25, New York, NY, USA, 2003. ACM Press.

[12] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 232–241, New York, NY, USA, 2006. ACM Press.

[13] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[14] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–25, New York, NY, USA, 2006. ACM Press.

[15] John Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, 1982.

[16] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 1–13, New York, NY, USA, 2004. ACM Press.

[17] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[18] W. Hwu and Y. N. Patt. Hpsm, a high performance restricted data flow architecture having minimal functionality. In *ISCA '86: Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 297–306, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[19] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 209–220, New York, NY, USA, 2006. ACM Press.

[20] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, 1997.

[21] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, New York, NY, USA, 2006. ACM Press.

[22] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, New York, NY, USA, 2006. ACM Press.

[23] Arndt Muehlenfeld and Franz Wotawa. Fault detection in multi-threaded C++ server applications. In *Informal Proceedings of the International Workshop on Multithreading in Hardware and Software*, 2006.

[24] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, New York, NY, USA, 2006. ACM Press.

[25] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178, New York, NY, USA, 2003. ACM Press.

[26] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPoPP '03: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–190, New York, NY, USA, 2003. ACM Press.

[27] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–331, New York, NY, USA, 2006. ACM Press.

[28] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO 34: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[29] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *SOSP*, pages 27–37, 1997.

[30] James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *ISCA '85: Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[31] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '03: Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 221–234, Brighton, UK, October 2005. ACM.