Detecting and Tolerating Asymmetric Races

Paruj Ratanaworabhan
Cornell University
paruj@csl.cornell.edu

Martin Burtscher
The University of Texas at Austin
burtscher@ices.utexas.edu

Darko Kirovski and Benjamin Zorn
Microsoft Research
{darkok, zorn}@microsoft.com

Rahul Nagpal
Indian Institute of Science
rahul@csa.iisc.ernet.edu

Karthik Pattabiraman
University of Illinois at Urbana-Champaign
pattabir@uiuc.edu

Abstract

This paper introduces ToleRace, a runtime system that allows programs to detect and even tolerate asymmetric data races. Asymmetric races are race conditions where one thread correctly acquires and releases a lock for a shared variable while another thread improperly accesses the same variable. ToleRace provides approximate isolation in the critical sections of lock-based parallel programs by creating a local copy of each shared variable when entering a critical section, operating on the local copies, and propagating the appropriate copies upon leaving the critical section. We start by characterizing all possible interleavings that can cause races and precisely describe the effect of ToleRace in each case. Then, we study the theoretical aspects of an oracle that knows exactly what type of interleaving has occurred. Finally, we present two software implementations of ToleRace and evaluate them on multithreaded applications from the SPLASH2 and PARSEC suites. Our implementation on top of a dynamic instrumentation tool, which works directly on executables and requires no source code modifications, incurs an overhead of a factor of two on average. Manually adding ToleRace to the source code of these applications results in an average overhead of 6.4 percent.

Categories and Subject Descriptors D.2.5 [**Software Engineering**]: Testing and Debugging – Debugging aids, Diagnostics; D.1.3 [**Programming Techniques**]: Concurrent Programming – Parallel programming

General Terms Reliability, Performance, Experimentation

Keywords race detection and toleration; runtime support; dynamic instrumentation

1. Introduction

As general-purpose microprocessors move from a single core to multiple cores per chip, programming needs to migrate from sequential to parallel code if programs are to exploit more than one CPU. This software transition, however, has not been as easy and natural as the hardware counterpart has. Programmers find it difficult to write and reason about parallel programs. As a result, such programs are usually rife with errors, many of which are unheard-of in sequential programs, e.g., atomicity violations and data races. Moreover, these errors are harder to deal with than sequential programming errors because of their non-deterministic nature.

PPoPP'09 February 14–18, 2009, Raleigh, North Carolina, USA.

Finding a suitable model that addresses the programmability of parallel programs while keeping up with the performance expected of multi-core hardware is an active research area. Promising candidates include transactional memory [16] and the Galois system [21]. At present, lock-based parallel programs are still dominant, particularly those written in unsafe languages such as C or C++ with add-on libraries for threading and synchronization. There is also ongoing research [9] that aims to improve the rigor of this programming paradigm.

Our work tackles race conditions in lock-based programs. In general, a race is defined as a condition where multiple threads access a shared memory location without synchronization and there is at least one write among the accesses. With proper synchronization, lock-based programs adhere to the data-race-free model [4] where synchronization operations are made explicit by calls to specific library functions, e.g., pthread_mutex_lock in POSIX threads (pthreads). In this model, the hardware appears sequentially consistent with respect to the programs even though it may be weakly ordered in reality [2]. We are interested in asymmetric races, which occur 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.).

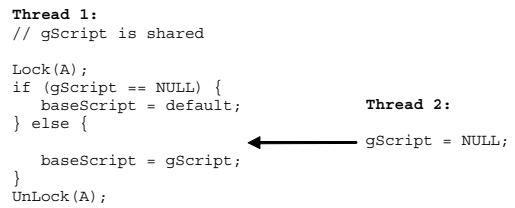


Figure 1. An asymmetric race.

An example of an asymmetric race is shown in Figure 1. Here, 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, thus creating a race. The race occurs infrequently, i.e., only when Thread 2's update happens between the test for NULL and the `else` part of the conditional in Thread 1. Our reasons for focusing on asymmetric races are:

1. They are common in software development projects.

This conclusion comes from direct experience with developers in software houses like Microsoft. There are two reasons for this. First, usually a programmer's local reasoning about concurrency, e.g., taking proper locks to protect shared variables, is correct. Errors due to taking wrong locks or no locks lie outside of the programmer's code, for example, in third party libraries. Given that lock-based programs rely on convention, this phenomenon is understandable. The second reason has to do with legacy code. As

software evolves, assumptions about a piece of code may be invalidated. For instance, a library may have been written assuming a single-threaded environment, but later the requirements change and multiple threads use it. An expedient response to this change is to demand that all clients wrap their calls to the library, acquiring locks before entry and releasing them on exit. Because this solution requires that all clients be changed, races can be introduced when clients fail to observe the proper locking discipline.

```
// K and flag are declared volatile

Thread 1:                Thread 2:

K = x;                   while (flag != true);
flag = true;             y = K;
```

Figure 2. User-defined synchronization.

2. Symmetric races are usually benign.
Because calls to synchronization operations are expensive, programmers often resort to lightweight user-defined synchronization as shown in Figure 2 where Threads 1 and 2 synchronize on the `flag` variable. In this situation, even though a race occurs by definition (the shared variable `flag` is accessed without explicit synchronization), it does not harm the program. Narayanasamy et al. [28] show other types of benign symmetric races, e.g., redundant writes and disjoint bit manipulation. Their experience with Windows Vista and Internet Explorer indicates that these benign races are rather common.

3. Programmers want to reason locally about the correctness of the critical sections in their code.
Normally, local reasoning cannot be applied when considering the correctness of parallel programs with shared variables. Components that are locally correct (e.g., use locks to protect a shared variable) are rendered incorrect by arbitrary code elsewhere 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 would be forced to program around the error in an ad hoc way. Our goal is to provide a tool that allows programmers to detect and respond to external concurrency errors in a structured and principled way with no changes to the external code.

ToleRace is a runtime system that allows programs to continue executing in the presence of asymmetric races and possibly complete with a well-defined semantic. Inspired by the DieHard system [6], which probabilistically tolerates memory safety errors, ToleRace uses replication to detect and/or tolerate races. It provides an approximation of isolation in critical sections by creating local copies of shared variables when a critical section is entered, operating on the local copy while in the critical section, detecting conflicting changes to shared data when the critical section is exited, and propagating the appropriate copy when possible to 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. In this paper, we focus on the fundamental properties of ToleRace and describe two possible software implementations.

ToleRace can be compared to transactional memory (TM) [16]. The ToleRace mechanism outlined above is analogous to constructing a read-write set while executing in a transaction with a lazy versioning policy and lazily detecting conflicts to the set, i.e., just before the transaction commits. However, ToleRace is not based on optimistic synchronization as TM is; there is no notion of abort-and-rollback, nor is there a need for contention management. Whereas handling side effect operations and nested transactions are still open issues with TM, ToleRace handles all I/O operations as well as overlapped critical sections transparently, preserving the semantics of the original lock-based program. While TM can provide isolation and tolerates races just as ToleRace does, it is not clear how TM can be applied to existing lock-based codes. Converting from lock-based to transaction-based code is not trivial [8].

This paper makes the following contributions:

- **Comprehensive runtime management of races.** ToleRace enables programs to tolerate races, which decreases the likelihood that the races will cause incorrect program behavior. Increasing a program's tolerance to races reduces the need for the races to be debugged/patched. In instances where ToleRace cannot tolerate races, it detects them either precisely or with high probability, depending on the implementation. Note that there can be instances where ToleRace silently and correctly tolerates races without detecting them (cf. Section 3.1).
- **Precise detection.** ToleRace only identifies races that actually happen at runtime. It detects a race when the critical section in which the race took place exits.
- **Programmer-centric local reasoning.** ToleRace enables programmer tools to allow local reasoning about correctness and to facilitate a structured means of detecting and tolerating errors that are caused by code outside the programmer's control. The programmer can control the overhead by selectively turning ToleRace on or off for individual critical sections. This is useful for debugging, testing, and patching released executables.
- **Low overhead software implementation.** We present two software implementations of ToleRace. Our general dynamic instrumentation-based approach incurs an overhead of a factor of two whereas our source-code based implementation incurs an overhead of 6.4% on average.

2. Characterizing Asymmetric Races

We denote as $l()$ and $u()$ the atomic functions that acquire and release a specific lock, respectively. We further denote as $r()$ and $w()$ two functions that read from and write to a specific variable. We first consider cases when a single variable is protected and accessed in a non-nested critical section. We then extend this theoretical framework to cover the cases involving multiple variables and overlapped critical sections. We use **l**, **u**, **r**, and **w** to denote the fundamental functions over that specific variable, and we use **x** to denote a "*don't care*" function that can be either a read or a write. **r**+ denotes a sequence of at least one read and **r*** indicates zero or more reads. The 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 a variable, then reads and writes exactly once, followed by at least one read before it unlocks the variable. The first digit in the operation index denotes the thread index and the second digit 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 between the first write and the second read in the first thread and thus causes a race condition.

Table 1. Tabulating classes of race instances. Column marked "race" denotes whether the schedule $T'_1T_2T''_1$ results in a race.

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
R+	r+	R+	false	R+	wx*	R+	true	R+	r+wx*	R+	true
R+	r+	WX*	false	R+	wx*	WX*	true	R+	r+wx*	WX*	true
R+	r+	R+WX*	false	R+	wx*	R+WX*	true	R+	r+wx*	R+WX*	true
WX*	r+	R+	false	WX*	wx*	R+	true	WX*	r+wx*	R+	true
WX*	r+	WX*	true	WX*	wx*	WX*	false	WX*	r+wx*	WX*	true
WX*	r+	R+WX*	true	WX*	wx*	R+WX*	true	WX*	r+wx*	R+WX*	true
R+WX*	r+	R+	false	R+WX*	wx*	R+	true	R+WX*	r+wx*	R+	true
R+WX*	r+	WX*	true	R+WX*	wx*	WX*	true	R+WX*	r+wx*	WX*	true
R+WX*	r+	R+WX*	true	R+WX*	wx*	R+WX*	true	R+WX*	r+wx*	R+WX*	true

To characterize asymmetric races, we 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 that consists of a single critical section, and
- a non-safe thread that might access a shared variable but does not have a critical section or uses the wrong lock to guard it.

Definition 1. A race condition represents any one of all possible execution interleavings of a set of threads $T = \{T_1 \ldots T_N\}$ *where at least one of the threads in* T *is non-safe and at least one is safe, such that the final computation state after all threads have executed does not correspond to any case when all safe threads in* T *have executed in isolation.*

Note that this definition does not say anything about what happens to the values of shared variables in non-safe threads. Because non-safe threads do not control their access to the values of shared variables, we assume they are written in such a way that they are able to tolerate arbitrary updates to these variables at any time. Because our solution focuses on tolerating and detecting asymmetric races, we consider an execution race free *only* with respect to the values of the shared variables in the safe threads.

Our definition is agnostic to the threads' execution order. 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 non-safe) in T could execute but not affect the program's computation state. In this case, we informally relax *Definition 1* to accept execution schedules as correct where a subset of the threads does not execute.

With the execution model defined above, ToleRace preserves the data-race-free-0 model semantics [22] when it tolerates all the occurring races. Such a model observes sequentially consistent execution for all synchronization operations while allowing the underlying hardware to be weakly ordered.

To understand how the safe and non-safe threads can interact, we exhaustively explore all interleavings where the non-safe thread T_2 executes between operations in the safe thread T_1. To simplify the analysis, we note that 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 (**r+**), by setting its value regardless of its prior (**wx***), and by setting its value based upon its prior (**r+wx***). 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 dependent upon the value retrieved by **r** and (ii) **w** is not dependent upon the value retrieved by **r**. Sequences where (ii) is true could be analyzed as independent manifestations of two sequences of type **r+** and **wx***. 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 a safe thread T_1 and a non-safe 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. 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. XwR = $\{\mathbf{l}_1\mathbf{x}+_1\mathbf{w}_2\mathbf{x}*_2\mathbf{r}_1\mathbf{u}_1\}$. *This case specifies that any sequence of operations by* T_2 *that starts with a write and occurs after any operation but before a read in* T_1 *causes a race.*

II. WrW = $\{\mathbf{l}_1\mathbf{r}*_{11}\mathbf{w}_{11}\mathbf{x}*_1\mathbf{r}+_2\mathbf{r}*_{12}\mathbf{w}_{12}\mathbf{u}_1\}$. *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. RXwW = $\{\mathbf{l}_1\mathbf{r}_1\mathbf{x}*_1\mathbf{w}_2\mathbf{x}*_2\mathbf{w}_1\mathbf{u}_1\}$. *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.*

IV. XrwX = $\{\mathbf{l}_1\mathbf{x}+_{11}\mathbf{r}+_2\mathbf{w}_2\mathbf{x}*_2\mathbf{x}_{12}\mathbf{u}_1\}$. *This case specifies that any sequence starting with a write based upon a prior by* T_2 *causes a race when interleaved between any two operations of* T_1.

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. □

There is previous work [24, 32] that also proposes enumeration of possible interleavings. However, it does not focus on race toleration as we do. Section 3.1 describes how we employ the classification from Table 1 for this purpose.

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. Consider a single safe thread among K interacting threads. The K-1 non-safe threads impart intervening sequences of operations **r+**, **wx***, or **r+wx*** to the safe thread. When these three sequences interleave, the resulting sequence still belongs to one of the three sequences. As far as the safe thread is concerned, no matter how many non-safe threads interact with it, it only observes the resulting intervening sequence. If such a sequence is one of the three sequences mentioned, it is as if it interacted with

just a single non-safe thread, and the resulting race instances can be classified by Table 1.

Now, consider multiple safe threads among the K interacting threads. Because safe threads, by definition, hold consistent locking for a given shared variable, only one can be in the critical section accessing this variable at a given time. This brings us back to the first case we just considered and completes the proof. □

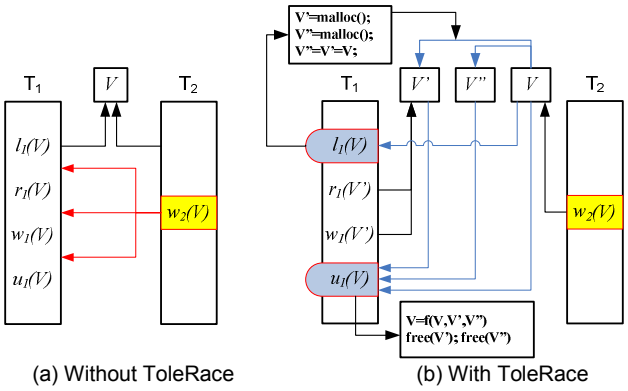


Figure 3. ToleRace uses two additional copies of a variable to tolerate races.

3. The ToleRace Oracle

The core of our approach to managing the 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 3). This thread continues reading from and/or writing to this copy until it releases the lock. When the lock is released, the ToleRace runtime can employ a variety of software and/or hardware mechanisms to determine which race, if any, has occurred. Possible outcomes range from tolerating the race completely to reporting that a race has occurred to executing a programmer-specific handler when an intolerable race is detected.

Next, we study the effect of ToleRace on the cases described in Theorem 1, assuming an oracle determines which race 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 and that storage for two additional copies (V', V'') of variable V has been allocated. After the lock is released, the storage for the two copies is deallocated.

Lock (Entry): When lock x_V is acquired by T_1, we copy V to V' and V'' ($V''=V'=V$) atomically.

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 the local copy of V for T_1 that cannot be accessed by other threads due to a race. All other threads such as T_2 are unchanged and continue using V for all accesses. Copy V'' is not accessed by any thread until T_1 exits the critical section.

Unlock (Exit): When T_1 exits the critical section by releasing the acquired lock, ToleRace analyzes the content 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 the 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 finishes its critical section. The resolution function is executed atomically in the oracle ToleRace.

3.1 Tolerating and Detecting Races with the oracle ToleRace

Combining the mechanism outlined above with the exhaustive interleavings enumerated in Table 1, we can reason about which cases ToleRace will tolerate. Assuming perfect knowledge of the specific race case that has occurred, Table 2 summarizes the definition of f and indicates the cases that ToleRace correctly tolerates.

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

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

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

IV_C: RrwW=XrwX – {RrwR ∪ 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 column presents π, the schedule of threads that ToleRace's result represents. Table 2 shows that the ToleRace oracle tolerates all races except sequences of the form RrwW with the resolution function f defined by Table 2.

Table 2. Tabulating the outcome of f for each race type.

race type		V = V''	f(V, V', V'')	tolerable	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
IV_A	RrwR	false	V	true	T_1T_2
IV_B	WrwX	false	V'	true	T_2T_1
IV_C	RrwW	false	custom f'	maybe	N/A

For races of type RrwW, 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 once T_1 has entered the critical section, execute in parallel, and then join at the exit of the critical section of T_1. T_1 and T_2 see the same value returned by the read, which would not be possible if T_1 had executed its critical section in isolation.

When functioning as a pure race *detector*, the oracle ToleRace inherently generates no false positives. When $V \neq V''$, an asymmetric race has occurred by definition. It produces a false negative when:

a) the last write in the intervening sequence writes the same value as the value in V''. This is the so called ABA problem. Surprisingly, ToleRace tolerates this case as ABA is indistinguishable from AAB.

b) there is a WrW race. Note that, while ToleRace cannot detect this race case, it can tolerate it.

3.2 Multiple Variables and Nested Critical Sections

So far, we have considered the oracle ToleRace in a multithreaded, single-variable, and non-nested critical section context. We now extend this framework to handle general cases, which involve multiple variables and nested critical sections. The extension is straightforward. Local copies and the resolution function need to be made and executed atomically for multiple variables. The oracle ToleRace treats multiple variables as multiple instances of a single variable. It resolves races on a per variable

basis, and, thus, reconciling multiple variables is the same as multiple reconciliations of a single variable. Nested critical sections share their local copies with the outermost critical section. However, they have their own resolution function to resolve races for their protected variables. This extension to ToleRace has the following characteristic.

Theorem 3. Sequentially inconsistent execution. In the general case of tolerating asymmetric races involving multiple variables and nested critical sections, ToleRace may reorder operations of a non-safe thread such that the operations do not follow their original program order. If there are dependencies among the operations that must be observed, this reordering is not serializable and ToleRace reverts to detection mode.

The detailed proof is given in a technical report [19]. Here we give a sketch of the proof presenting the high-level idea. Note that the dependencies in Theorem 3 refer to data dependences, which occur when a write to a given variable depends on a read of another variable.

We consider cases I through IV_B from Table 2 where ToleRace tolerates races without a custom resolution function. ToleRace can schedule operations from the non-safe thread to have come before or after the critical section. Any intervening sequence **r+** always appears to have come before the critical section (race type II) whereas the sequence **wx*** always appears after (race type I and III). For the **r+wx*** sequence, the schedule depends on the race type (after for IV_A and before for IV_B).

Table 3. Enumeration of intervening sequences to P and Q. Trailing x* and r+ of P sequence may overlap with Q sequence.

P	Q	reordered by ToleRace	dependency from P to Q	ToleRace action
r+	r+	No	No	Tolerate
wx*	r+	Yes	No	Tolerate
r+wx*	r+	If race IV_A to P	No	Tolerate
r+	wx*	No	maybe	Tolerate
wx*	wx*	No	maybe	Tolerate
r+wx*	wx*	No	maybe	Tolerate
r+	r+wx*	No	maybe	Tolerate
wx*	r+wx*	If race IV_B to Q	maybe	Detect if reordered, tolerate otherwise
r+wx*	r+wx*	If race IV_A to P and IV_B to Q	maybe	Detect if reordered, tolerate otherwise

Consider an asymmetric race involving two variables P and Q. Let a non-nested critical section protect both variables in a safe thread. In a non-safe thread, let an intervening sequence to P come before an intervening sequence to Q in program order, but the two can overlap each other. Table 3 enumerates all possible P and Q intervening combinations from the non-safe thread. The third column indicates whether ToleRace reorders the intervening operations to P and Q. This follows directly from the resolution function in Table 2 as just described. The fourth column lists if there is a dependency from P to Q. In general, when there is a write to Q and the accesses to P may contain a read, then Q may be dependent on P, and, hence, the operations must observe program order. The fifth column shows the ToleRace action for each combination, which can be deduced directly from the result in columns 3 and 4. ToleRace reverts to detection mode when it determines that there may be a dependency among variables and the resolution function allowed out-of-order execution.

In general, we do not expect this extended framework to be invoked often. A recent study by Lu et al. [23] points out that the common cases for concurrency bugs, including races, involve only a single variable and no more than two threads.

4. ToleRace Software Implementation

This section presents a software implementation of ToleRace on top of an existing software instrumentation tool. The next section evaluates the performance of our implementation and demonstrates that its overhead is reasonable. Although the oracle ToleRace framework allows for both software and hardware implementations, a software approach may be more appealing as it can be deployed immediately. Our software version makes all decisions at runtime and does not perform any static program analysis. In this sense, it should give us an upper bound on the overhead. We expect other implementations to be more efficient.

4.1 Pin Tools

We chose to implement ToleRace on top of Pin [26], a dynamic instrumentation toolkit from Intel. Pin is available for a variety of operating systems and architectures, including x86 Linux and Windows platforms. The dynamic compilation and instrumentation nature of Pin allows us to identify critical sections and the shared variables they protect at runtime without static annotations. Moreover, Pin is stable and its performance compares favorably with other similar tools.

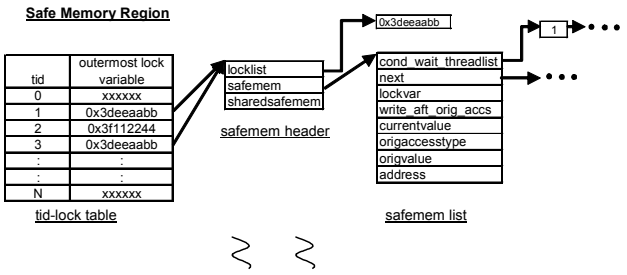
We now describe our x86 Linux Pin-ToleRace implementation for parallel pthreads-based programs in detail. Although our implementation is somewhat platform specific and the multithreaded applications we use employ a specific thread library, we believe the framework described here generalizes to other platforms and threading libraries. Pin-ToleRace supports statically and dynamically linked executables. As we are only interested in the critical sections of specific code regions, we assume information is provided to ToleRace so that it can identify the code regions in question. Note that, in the rest of this paper, the term *user code* refers to the code regions that are to be protected by ToleRace whereas *library code* refers to all other code (which does not have to reside in a library though it might).

4.2 The General Pin-ToleRace Framework

As the oracle ToleRace has complete knowledge of all the shared variables protected by a critical section, it can create the local copies as soon as the critical section is entered. Of course, such oracle knowledge may not be available in practice due to dynamically allocated shared variables. Hence, our Pin-ToleRace implementation assumes no such knowledge and the shared variables associated with a particular critical section are always determined on the fly. Pin-ToleRace works directly on the executable; no source code is required. The notion of shared variables, thus, is redefined to that of shared memory locations. We conservatively assume that all memory accesses in a critical section touch shared memory locations except for those touching the thread local stack. We use the term *safe memory* to refer to the region of memory that holds the local copies of the shared memory.

The safe memory is initially empty. Once a running thread is detected to have entered a critical section, each executed instruction with a memory operand touching a shared location is instrumented; no instructions get instrumented outside of critical sections. The instrumentation code searches the safe memory region for a local copy of the shared memory that is being accessed. If found, the memory access is redirected to this copy. If not found,

the analysis routine creates a new node in the safe memory. The node records the address, the original value and the current value of the shared memory location together with other metadata that we describe later. It serves as a local copy of this shared location that all subsequent accesses in this critical section will consult. When exiting from the critical section, Pin-ToleRace traverses the nodes in the safe memory region and compares the saved original value with the value in the corresponding true memory location. After taking the appropriate action to tolerate or detect a race, if any, it frees the nodes.




```
// Instrumentation Routine
VOID Instruction(INS ins) {
  if (call to pthread_mutex_lock && in user code) {
    Insert analysis routine CSEnter
  }
  if (call to pthread_mutex_unlock && in user code) {
    Insert analysis routine CSExit
    Insert analysis routine for the resolution function
  }
  if (CSLevel[PIN_ThreadId()]>=1) {
    if (non-stack accesses) {
      Rewrite memory operands
      Insert analysis routine to redirect accesses to the safe memory
    }
  }
}
```

Figure 4. Pin-ToleRace framework.

4.3 Implementation Details

This subsection describes the implementation of Pin-ToleRace, whose framework is shown pictorially in Figure 4.

4.3.1 The Safe Memory Region

As mentioned, the safe memory region is where the local copies of the shared memory locations reside. It contains three main data structures: the thread id (tid) lock mapping table, the safemem header, and the list of `safemem` nodes. The lock mapping table size is determined by the maximum number of threads allowed in the system. The other two are dynamic structures, and their content is alive as long as the execution proceeds through a critical section. The content is created by the first instruction that accesses a shared memory location. The role of each of these structures is explained next.

4.3.2 Identifying Critical Sections

A critical section is defined by a mutex variable and a pair of pthread_mutex_lock and pthread_mutex_unlock calls with the mutex variable as their argument. Pin-ToleRace instruments lock/unlock calls dynamically. When a lock routine is executed, it adds a call to the CSEnter analysis routine. The analysis routine increments the CSLevel counter and sets the respective entry in the tid-lock table by updating it with the thread id and lock variable argument passed to it. The CSLevel counter is a per thread counter that keeps track of the critical section nesting level. When an unlock call is encountered, a call to the CSExit routine is added, which decrements the CSLevel counter. A thread is executing inside a critical section if its CSLevel counter (CSLevel[tid]) is greater than or equal to one. Because Pin-ToleRace is only concerned with user code (see earlier definition), we only instrument lock/unlock calls in the selected code regions.

4.3.3 Instrumenting Accesses to Shared Memory Locations

When an instruction is executed, Pin-ToleRace determines which thread it belongs to with the PIN_ThreadId() function. Then, it checks the value of CSLevel[tid] and whether the instruction is accessing a shared memory location. Instrumentation is enabled only when CSLevel[tid] is greater than zero. We ignore operands that access the local stack; all other locations are presumed to be shared, which includes all truly shared locations as well as some false locations such as private heap variables. Pin-ToleRace cannot determine whether a particular heap location is shared, and, therefore, conservatively assumes all heap locations to be shared. Once we decide that an instruction accesses a shared location, we rewrite its memory operand. Note that some CISC instructions require multiple operands to be rewritten per instruction. The operand is converted from its current addressing mode to the base register addressing mode using one of Pin's scratch registers. We instrument this instruction and pass the effective address of the memory operand to the analysis routine. The analysis routine determines which thread is executing it and searches the corresponding `safemem` linked list using the effective address as the search key. If a match is found, the routine returns the address of the `currentvalue` field of the matching node. This address is written into the scratch register that is used as the base address register for the rewritten operand. If no match is found, the analysis routine creates a new node and updates the `origvalue` and `currentvalue` fields with the true memory value obtained by dereferencing the effective address. (This performs the $V''=V'=V$ operation.) It then returns the address of the `currentvalue` field like in the found case. Although the instrumentation routine is a callback routine that is called by multiple threads, it does not create a race as it is serialized under Pin. Any thread can instrument code as long as it is executing in a critical section, and the same instrumented code will apply to all other threads.

4.3.4 Critical Section Exit

Before the call to the unlock routine at the critical section exit, we insert a call to an analysis routine that executes the resolution function. The associated lock variable is passed to this routine to handle nested critical sections. At this point, we resolve all race conditions to the shared memory locations accessed within the critical section according to Table 2. Section 4.4 provides more detail. After the race condition resolution, the `safemem` nodes are freed, provided that the current critical section is not nested and that there are no outstanding waits on condition variables (cf. Sections 4.3.6 and 4.3.8).

4.3.5 Handling Partial Reads and Writes

The `address` field in a `safemem` node is aligned to the native machine width. In case of IA-32, the last two bits are always zero. When an instruction accesses a `safemem` node with a size of less than 4 bytes, i.e., a byte or a short access, its memory operand address needs to be checked against a range of addresses.

4.3.6 Nested and Overlapped Critical Sections

The main component of the safe memory data structure that handles nested and overlapped critical sections is the `locklist` in

the safemem header. The `locklist` is maintained such that the head of the list always points to the most recent lock variable associated with the innermost critical section. This approach correctly associates shared memory accesses with the most recent lock variable acquired.

A critical section that executes inside another critical section never creates a new safemem list. Instead, it shares this structure with the outer critical section(s). If this were not so, the inner critical section could access stale memory values as the most up to date values might reside in another safe memory region.

Upon critical section exit, the resolution function selectively resolves races for the shared memory locations that are associated with the current lock variable. Recall from the previous section that the lock mutex variable is passed to the analysis routine. We traverse all `safemem` nodes, check for a matching `lockvar` value, resolve races for that particular node, and delete that node from the safemem list. The corresponding node in the lock list is also deleted. At this point, the shared memory associated with the matching `lockvar` becomes globally visible. If the `locklist` becomes empty, the safemem header is freed and the respective entry in the tid-lock table is reclaimed.

If the multithreaded program under consideration contains nested critical sections but none that overlap, we can simplify our scheme because there is no need for a list of lock variables. The current call to the unlock routine will correctly be matched with the most recent call to the lock routine. Shared memory accesses in the inner critical section can always be associated with the nesting level given by CSLevel[tid] without the extra lock variable context.

One subtlety with Pin-ToleRace involves a (non-nested) critical section that calls a function that is also called from outside any critical section. This creates a situation where the non-critical code in the called function is executed under a non-nested critical section whereas the code inside the critical sections receives an extra nesting level. A problem arises once the function's code is no longer executed under any critical section as it may contain accesses to false locations whose addresses were redirected by the code instrumentation. Since there is no resolution routine, the content of the safe memory is never transferred to the true memory locations, which will likely crash the program. Our solution to this problem is to put a guard on the analysis code that only allows it to perform the safe memory access when the CSLevel is greater than zero. Thus, when the function is executed outside a critical section, it will access the original memory locations.

4.3.7 Routine Calls inside a Critical Section

Function calls inside a critical section are handled correctly with the already described data structures of the safe memory. If a call passes a shared memory value on the stack, this shared value is correctly obtained from the safe memory region. Or, if the called function accesses shared memory locations, its accesses are redirected to the safe memory. However, we must distinguish between a call to a user-defined and a call to a library routine. We only want to protect user code, and, therefore, do not want to redirect shared memory accesses in library code. Nevertheless, we cannot simply exclude accesses to the safe memory from libraries because a call to a library routine can pass pointers to shared variables as arguments. To handle this case, we allow the library code to access the existing nodes in the safemem list but not to create new nodes.

4.3.8 Handling Condition Variables

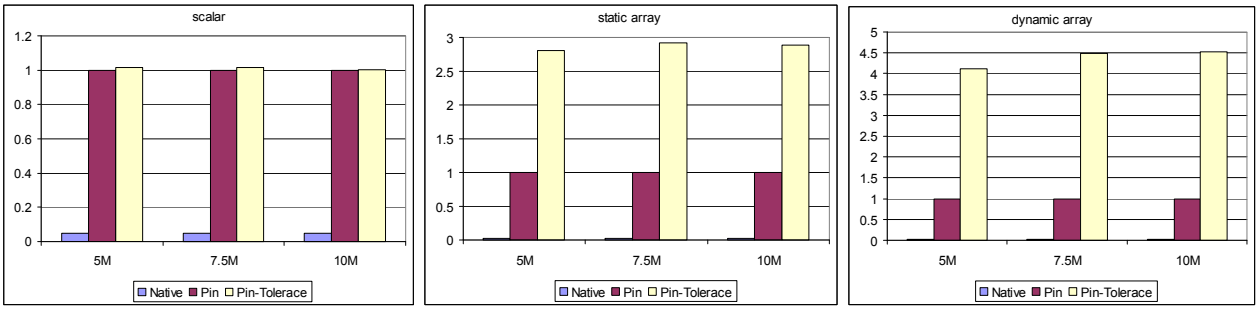
In addition to lock and mutex variables that synchronize threads by controlling access to data, the pthreads library also supports the use of condition variables to synchronize threads based on a data value. A call to pthread_cond_wait with a condition variable and a mutex variable as arguments atomically unlocks the mutex variable and makes the thread wait for the value of the condition variable. A call to pthread_cond_signal with the corresponding conditional variable wakes up one of the waiting threads. These two calls are instrumented with an analysis routine that increments and decrements, respectively, the global wait counter.

Condition variables complicate ToleRace because they allow multiple threads to be in a critical section at the same time. When a new thread enters a critical section while some other threads are waiting, this new thread cannot simply create its own copy of the safe memory. Instead, it must share this copy with the waiting threads. Hence, whenever a thread enters the critical section and there is an outstanding conditional wait as indicated by the wait counter, Pin-ToleRace searches the tid-lock table for the lock variable, uses the safemem header associated with this lock variable, and increments the `sharedsafemem` field in the safemem header. When the thread updates or creates a node in the safemem list, it puts its tid on the node's `cond_wait_threadlist`. When it exits the critical section, it checks whether it is the last thread to exit, and, if so, follows the normal exit procedure and frees the safemem list. Otherwise, it resolves races only on the locations it touched. If it was the only thread accessing this node, it deletes the node from the list. If the node has been accessed by multiple threads, the thread resolves any races for the node but leaves the node in the list and only deletes its tid from the node's `cond_wait_threadlist`. If the thread needs to copy the value to the true memory, it must also update the `origvalue` field with the `currentvalue`. This ensures that when the remaining threads sharing this node resolve race conditions, they will not signal a false race.

4.4 Tolerating and Detecting Races with Pin-ToleRace

When Pin-ToleRace performs the resolution function, it knows the type of the first access to a shared location as this information is recorded in the `origaccesstype` field when the node is created. It also knows whether subsequent accesses to this location included a write (`write_aft_orig_accs` field). Therefore, Pin-ToleRace can determine the types of accesses that are involved in a race to this shared location. When it compares V with V'' and finds that $V \neq V''$, the non-safe interleaving thread must contain a write. However, it cannot distinguish between the two write sequences, **wx*** and **r+wx*.** In some environments, the write sequence may be known, which enables Pin-ToleRace to tolerate all races that the oracle ToleRace can tolerate (see Table 2). In general, however, Pin-ToleRace must conservatively assume the worst case interleaving, i.e., **r+wx***, which prevents it from tolerating type III races. Aside from this restriction, it tolerates the same race types as the oracle.

As a race *detector*, Pin-ToleRace has the same properties as the oracle (cf. Section 3.1) except it introduces an additional false negative situation due to its non-atomic execution of the resolution function. This happens when immediately after the comparison of V and V'' returns equal, the intervening sequence writes to V. Given that the intervention must happen precisely at that moment, the probability of this occurring should be low. Pin-ToleRace does tolerate races in this situation. To see this, let us



(a) (b) (c)

Figure 5. Normalized execution time of Pin-ToleRace for scalar (a), static array (b) and dynamic array (c) for different iteration counts.

revisit Table 2. It is sufficient to consider only race case IV as Pin-ToleRace assumes **r+wx*** for all intervening write sequences. In the absence of a race, when the safe thread operations contain only reads, Pin-ToleRace never writes the local copy back; when the operations start with a write, it always writes back the local copy. This effectively enforces schedule T_1T_2 and T_2T_1 and thus tolerates race types IV_A and IV_B, respectively, if they occurred. Only race type IV_C remains problematic.

5. Evaluation

5.1 Benchmarks

We use 13 applications from the SPLASH2 [33] and PARSEC [7] benchmark suites to evaluate Pin-ToleRace. We also developed three microbenchmarks to stress-test a program's safe thread race toleration in the presence of non-safe threads. The microbenchmarks are called scalar, static array, and dynamic array.

The eight programs from the SPLASH2 suite were chosen per the minimum set recommended by the suite's guidelines. Four of the programs, cholesky, fft, lu, and radix, are kernels whereas the other four, barnes, ocean, radiosity, and water, are full applications. We replaced the SPLASH2 suite's PARMAC macros with a pthreads library implementation. We use the default input for each program but increase the size to lengthen the runtime where necessary.

We selected the five programs from the newly released PARSEC suite that use the pthreads library and define their critical sections by pthread_mutex_lock and pthread_mutex_unlock call pairs. One of these programs, dedup, is a kernel and the other four, facesim, ferret, fluidanimate, and x264, are real applications. The PARSEC suite aims to provide up-to-date multithreaded programs that focus on emerging workloads in recognition, mining, and synthesis. We use the simlarge inputs.

5.2 System, Compiler, and Timing Measurement

All benchmarks, including the microbenchmarks, are compiled and run on an Intel 32-bit system (IA-32) with a four-core 2.8 GHz Pentium4-Xeon CPU with a 4-way associative 16 kB L1 data cache per core, a 2 MB unified L2 cache, and 2 GB of main memory. The operating system is Red Hat Enterprise Linux Release 4 and the compiler is gcc version 3.4.6. We compiled the SPLASH2 and PARSEC programs per each suite's guideline with the -O2 and -O3 optimization level, respectively. The microbenchmarks use the -O3 optimization level. The system enforces memory alignment, which is necessary for Pin-ToleRace to function correctly (cf. Section 4.3.5). All timing measurements refer to the elapsed time as measured by the UNIX shell command *time*.

5.3 Stress Test

The stress tests demonstrate Pin-ToleRace's ability to tolerate races of the form RXwW. In this type of race, the safe thread performs read-increment-write operations on some shared locations while the non-safe threads write random values to these locations.

In the program scalar, the safe thread increments a single shared location from zero to a given number of iterations. The entire incrementing loop resides in a single critical section. At the same time, several non-safe threads set this memory location to their thread id and then read the value back to compute its square. The programs static array and dynamic array perform the same function. However, instead of a single shared location, the safe thread increments all elements in a static array of size 10 and all elements in a 5x5 2-D dynamic array allocated on the heap, respectively. The non-safe threads write their IDs to all of these shared locations.

For these tests, we know that the non-safe threads will cause races that always begin with a write to a shared location. By monitoring all shared accesses to the safe memory region, Pin-ToleRace determines that the safe thread reads and then writes to the shared locations. Once it identifies this RwxW type race, it can tolerate it by scheduling the non-safe thread's action to have happened after the safe thread's read-increment-write operations. Our test setup uses five non-safe threads and runs the three programs with 5M, 7.5M, and 10M iterations. In each experiment, we observe the correct values in all shared locations just before the critical section exit. We also see that after exiting from the critical section, the values of these shared locations change to the thread id of the non-safe thread that last ran.

Figure 5 reports the overhead of Pin-ToleRace for tolerating these RXwW races. It is normalized to the runtime of the three programs under Pin with no instrumentation. We find that the overhead is largely constant with respect to the number of iterations. Note that the native and Pin runs of all three programs suffer from race conditions while the Pin-ToleRace runs have all their races correctly tolerated.

For all three microbenchmarks, the overhead of Pin-ToleRace over native is very high—up to 80 times in the dynamic array case. The primary reason is that we are riding on the Pin overhead. If we measure the overhead of Pin-ToleRace over Pin, the dynamic array benchmark incurs an overhead of about 4.5 times. While this is substantial, it should be noted that the microbenchmarks almost always execute in a critical section, which is where all the Pin-ToleRace code resides. Moreover, because the `safemem` nodes are organized as a linked list, the linear search operation in the presence of many shared locations contributes greatly to the overhead. For example, going from scalar to static

array more than doubles the overhead. In other words, these microbenchmarks reflect worst case scenarios as they are always busy tolerating races inside a critical section. The next section shows that real applications have critical section characteristics that are more benign and thus result in a much lower overhead.

One additional point to note is that, with Pin-ToleRace, the overhead of tolerating a race is about the same as detecting a race. In both cases, all operations to the safe memory region are the same up to the critical section exit. At this time, if Pin-ToleRace decides to perform race detection, it reports the race on a particular shared location and terminates the application. If it decides to tolerate the race, it either leaves the state of the shared locations as it is or writes to them, depending on the type of race. Thus, the overhead of tolerating different types of races may differ slightly, but the difference should be small.

Table 4. Critical section characteristics.

	unique	nested CS	total executed	dynamic number of instrs per CS (user)	% dynamic instrs in CS
cholesky	14	no	11,849	29	< 0.1%
fft	10	no	55	17	< 0.01%
lu	7	no	1,043	17	< 0.01%
radix	9	no	51	17	< 0.01%
barnes	10	no	1,098,771	94	0.18%
ocean	26	no	3,335	17	< 0.01%
radiosity	36	yes	1,739,512	18	0.11%
water-spatial	16	no	853	13	< 0.01%
dedup	7	yes	256,380	600	0.42%
facesim	5	yes	10,161	46	< 0.01%
ferret	4	yes	552,173	690	1.59%
fluidanimate	11	no	4,359,405	13	0.40%
x264	2	no	16,767	11	< 0.01%

Table 5. Unique accesses to possibly shared locations per critical section by each thread.

	unique accesses	
	AVG	STD
cholesky	4.78	0.38
fft	1.37	0.04
lu	2.99	0.01
radix	2.82	0.19
barnes	19.13	0.03
ocean	3.00	0.00
radiosity	4.92	0.23
water-spatial	2.62	0.01
dedup	80.87	3.52
facesim	7.70	1.14
ferret	72.89	33.83
fluidanimate	5.00	0.00
x264	2.16	0.02

5.4 Real Applications

This section characterizes the critical sections of the 13 benchmarks and discusses the overhead of Pin-ToleRace on these programs.

5.4.1 Critical Section Characterization

For this study, we compiled the 13 benchmarks to use four processors, which corresponds to the number of cores on our evaluation platform. We then used Pin to collect the critical section statistics shown in Table 4. Note that we only study critical sections that reside in the user code, i.e., we exclude all library code.

The second column of Table 4 shows that the number of unique critical sections per benchmark is quite small. radiosity tops the list with 36. All but two of the programs have 16 or fewer critical sections. Only four benchmarks, radiosity, dedup, facesim, and ferret, contain nested critical sections. Note that some of these nestings are statically non-nested. For example, a call inside a non-nested critical section to a function that contains a non-nested critical section dynamically results in nesting. The last column shows the total number of executed instructions within the critical sections. The numbers in this column exclude the instructions of any library routines called from the critical sections. All programs except ferret execute less than one percent of their dynamic user instructions in critical sections.

The fourth column of Table 4 shows the total number of executed critical sections. The counts range from under one hundred in fft and radix to over one million in barnes, radiosity, and fluidanimate. The average number of instructions executed in user code per critical section is given in column five. Two benchmarks, dedup and ferret, stand out. Both execute over 600 instructions per critical section. barnes follows as a distant third at 94. These three benchmarks execute loops inside their critical sections. The rest of the programs execute fewer than 30 instructions per critical section. Nevertheless, some of them have a high total dynamic instruction count inside critical sections, notably fluidanimate and radiosity whose small critical sections are being looped over.

Next, we look at the critical sections from the point of view of Pin-ToleRace. Table 5 shows the average number of shared memory locations accessed per critical section execution by each benchmark. With the exception of ferret, this number is very uniform across the running threads as the standard deviations indicate. Nine out of the 13 benchmarks perform fewer than five unique accesses. With so few accesses, Pin-ToleRace's linked list structure in the safe memory should not be a performance bottleneck. However, in barnes and especially in dedup and facesim, the number of unique accesses to shared locations is quite high. With these programs, the linear search through the linked list structure can add considerably to the Pin-ToleRace overhead. Overall, the number of unique shared memory accesses seems to be in proportion with the number of instructions executed per critical section.

5.4.2 Pin-ToleRace Performance

This section studies the overhead of Pin-ToleRace on our benchmark applications. Given the results of the previous subsection, we decided to investigate two implementations of the safe memory. One uses the linked list approach described earlier and the other uses a chained hash table with 128 entries. We choose this size to minimize the collisions in dedup and ferret.

Figure 6 presents the results. The timing measurements are normalized to the native runtime. Note that this is different from the normalization we used for the stress tests. The second bar shows the pure Pin overhead without instrumentation for each program. The third and fourth bars indicate the overhead of Pin-ToleRace with linked list and hash table implementations of the safe memory, respectively. On average, Pin-ToleRace incurs about a factor of two slowdown relative to the native runs and about 24% overhead relative to the Pin runs. We believe these performance degradations to be low enough to make Pin-ToleRace deployable in production environments. Moreover, by adding static analysis or hardware support, it should be possible to reduce the overhead.

As expected, the hash table implementation of the safe memory reduces the Pin-ToleRace overhead of barnes, dedup, and ferret. Unfortunately, it increases the overhead of all the other programs. The reason is that the chained hash table is more expensive to initialize and free than the linked list. With the hash table scheme, there is a fixed minimum number of entries to process (proportional to the table size) whereas with the linked list

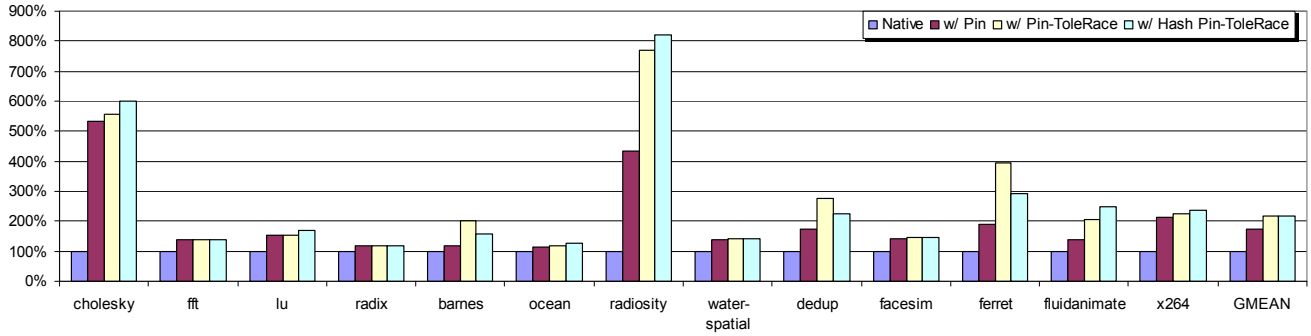


Figure 6. Normalized execution time of Pin-ToleRace.

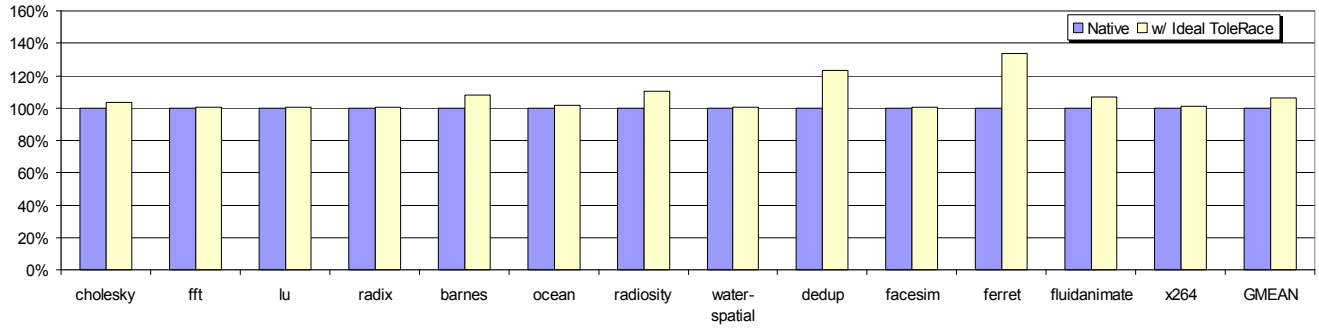


Figure 7. Normalized execution time of ideal software ToleRace.

there are only as many nodes as there are unique shared memory locations. Therefore, the hash table is only attractive when the execution in a critical section can amortize this overhead. Recall from the previous section that each of the three benchmarks for which the hash table implementation works better executes a relatively large number of instructions and touches many unique shared memory locations inside the critical sections. The remaining benchmarks have small critical sections, and each critical section execution does not touch many unique shared locations, making the linked list implementation better suited.

5.4.3 Idealized Software ToleRace

Suppose we have an oracle compiler that knows all the shared locations within a critical section. The performance overhead of a ToleRace implementation based on such a compiler presents a lower bound on what we can achieve in software. (Recall that Pin-ToleRace infers all the shared memory locations on-the-fly, thus yielding an upper bound.)

To mimic the effect of such an oracle compiler, we manually modified the source code of our benchmarks after carefully studying the critical sections and the shared variables in each of them. In a few critical sections, we could not precisely mimic the effect of the oracle compiler because of shared variables that are allocated at runtime. In these instances, we instead mimic the mechanism used in Pin-ToleRace. Moreover, in barnes and radiosity, we only modified frequently executed critical sections that cumulatively account for 99% and 90% of all dynamic critical section executions, respectively. We believe that doing so should not significantly affect the overhead result.

After we incorporated ToleRace into the critical sections, we recompiled and ran these applications. Figure 7 shows the overhead results, which are normalized to the native execution time without ToleRace. The ideal software ToleRace incurs a 6.4% overhead on average across our benchmarks. ferret executes inside critical sections more often than other applications and has many runtime allocated shared variables. Consequently, it incurs the highest overhead. dedup, which has the second highest overhead, has similar characteristics as ferret. Most of the applications, however, incur less than 1% overhead with the ideal software ToleRace.

6. Related Work

Related race-detection research includes both static and dynamic approaches. Static race detection relies on program analysis and either assumes existing programming languages (e.g., Java [27]) or defines new programming language semantics that help improve the static detection of races (e.g., Cyclone [14]). Static analysis techniques face several challenges. First, because many of the techniques are based on some form of model checking [15], 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 [12] and Houdini/rcc [13] 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 lock-sets [31]. Experience with this approach has shown that the overhead of maintaining the locksets is high and that false positives can be problematic. Subsequent approaches extend locksets with hap-

pens-before analysis [3]. Combining locksets with a happens-before scheme results in higher precision dynamic race detectors [10, 11, 29, 34]. Even with refinements, the execution overhead of these approaches is typically larger than a factor of two. As we have seen, previous work focuses primarily on detecting data races rather than tolerating them. The ToleRace detection technique is distinct from the lockset and happens-before algorithms. Focusing only on asymmetric races allows ToleRace to take a transaction-like approach to race detection and toleration, which significantly reduces the overhead of dynamic race detection.

Dynamic race detection approaches have also been adopted by Intel's Thread Checker [18] and Sun's Thread Analyzer [17], which are commercial tools capable of locating data races in concurrent programs. Both tools suffer from a high memory footprint and runtime overhead and are, thus, primarily used for software testing.

Atomicity violation is another important class of concurrency errors. It can be addressed statically [5] or dynamically. The AVIO system [24] belongs to the latter category and enumerates erroneous access interleavings similar to our asymmetric race interleavings. However, it only looks at single load/store pairs and not sequences of accesses. Without hardware support, the overhead of AVIO is very high, which makes it suitable only for test environments. The work by Lucia et al. [25] offers to tolerate some degree of atomicity violation with implicit atomicity by grouping consecutive memory operations into atomic blocks.

Vaziri et al. [32] classify harmful interleavings into 11 categories, which is more than the six race cases (with case IV subdivided) we considered. The extra categories address high-level data races at the object granularity. Their approach to race detection requires source-code annotation and targets safe language environments.

Kiena et al. [20] propose two schemes to dynamically heal data races for Java programs. In one scheme, they reduce the probability of races happening by forcing threads that are about to cause racy accesses to yield. This is done at the byte-code level through yield() calls. In the other scheme, they add extra locks to some common code patterns that are likely to result in races.

Concurrent to our work, Rajamani et al. [30] propose a runtime system called Isolator that enforces isolation through page protection. The idea is to protect the pages containing shared variables (that are protected by a lock) so that accesses to them can be intercepted. Then, accesses to those variables that observe the proper locking discipline are redirected to a local copy of the corresponding page. Any improper access will be to the original page and hence raise a page protection fault. Similarly, Abadi et al. [1] use page-level protection to guarantee strong atomicity in software transactional memory.

7. Summary

We introduce ToleRace, a novel runtime system that uses data replication for detecting and tolerating concurrency errors in lock-based multithreaded programs. ToleRace addresses asymmetric races, where one use of a shared variable is correctly protected with locks while other uses are not. We present a theoretical framework as well as two software implementations. Our evaluation indicates that real applications can run on top of software ToleRace with acceptable overhead.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments on this paper. The Fusion and M3 groups in the Computer Systems Laboratory at Cornell University provided some of the computing resources used to obtain the results for this work. James Cownie and Gregory Lueck were very helpful with the Pin and Thread Checker tools. Emery Berger, Chen Ding, Manuel Fahndrich, Tim Harris, Sriram Rajamani, Ganesan Ramalingam, and Jason Yang gave useful comments and suggestions during the development of ToleRace. Martin Burtscher is supported by NSF grants 0833162, 0719966, 0702353, 0615240, 0541193 as well as grants from IBM and Intel. Paruj Ratanaworabhan is supported by DOE grant DE-FG02-06ER25722.

References

[1] M. Abadi, T. Harris and M. Mehrara, *Transactional memory with strong atomicity using off-the-shelf memory protection hardware*, *Proceeding of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* Raleigh, NC, 2009.

[2] S. V. Adve and M. D. Hill, *Weak Ordering - A New Definition*, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 2-14.

[3] S. V. Adve, M. D. Hill, B. P. Miller and R. H. B. Netzer, *Detecting data races on weak memory systems*, *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, ACM Press, New York, NY, USA, 1991, pp. 234-243.

[4] S. V. Adve, V. S. Pai, P. Ranganathan and A.-S. H., *Recent Advances in Memory Consistency Models for Hardware Shared-Memory Multiprocessors*, Proceedings of the IEEE, special issue on distributed shared-memory, 87 (1999), pp. 445-455.

[5] R. Agarwal, A. Sasturkar, L. Wang and S. Stoller, *Optimized Run-Time Race Detection and Atomicity Checking Using Partial Discovered Types*, *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005, pp. 233-242.

[6] E. D. Berger and B. G. Zorn, *DieHard: probabilistic memory safety for unsafe languages*, ACM SIGPLAN Notices, 41 (2006), pp. 158-168.

[7] C. Bienia, S. Kumar, J. Singh and K. Li, *The PARSEC Benchmark Suite: Characterization and Architectural Implications*, *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[8] C. Blundell, C. Lewis and M. Martin, *Deconstructing Transactional Semantics: The Subtleties of Atomicity*, *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, Madison, Wisconsin, 2005.

[9] H. Boehm, *Foundations of the C++ Concurrency Memory Model*, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, 2008.

[10] R. Callahan and J.-D. Choi, *Hybrid Dynamic Data Race Detection*, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM Press, New York, NY, 2003.

[11] T. Elmas, S. Qadeer and S. Tasiran, *Goldilocks: Efficiently Computing the Happens-Before Relation Using Locksets*, in K. Havelund, N. Manuel, G. Rosu and B. Wolff, eds., *FATES/RV*, Springer, 2006, pp. 193-208.

[12] D. R. Engler and K. Ashcraft, *RacerX: effective, static detection of race conditions and deadlocks*, *SOSP '03: Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2003, pp. 237-252.

[13] C. Flanagan and S. N. Freund, *Detecting race conditions in large programs*, *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ACM Press, New York, NY, USA, 2001, pp. 90-96.

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

[15] T. A. Henzinger, R. Jhala and R. Majumdar, *Race checking by context inference*, *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ACM Press, New York, NY, USA, 2004, pp. 1-13.

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

[17] http://developers.sun.com/sunstudio/downloads/ssx/tha/.

[18] http://www.intel.com/cd/software/products/asmona/eng/286406.htm.

[19] D. Kirovski, B. Zorn, R. Nagpal and K. Pattabiraman, *An Oracle for Tolerating and Detecting Asymmetric Races*, *Microsoft Research Technical Report MSR-TR-2007-122*, Microsoft Research, 2007.

[20] B. Krena, Z. Letko, R. Tzoref, S. Ur and T. Vojnar, *Healing Data Races On-The-Fly*, *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, London, UK, 2007.

[21] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala and P. Chew, *Optimistic Parallelism Requires Abstractions*, *Programming Language Design and Implementation*, San Diego, CA, 2007.

[22] L. Lamport, *How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor*, IEEE Transactions on Computers, 46 (1997), pp. 779-782.

[23] S. Lu, S. Park, E. Seo and Y. Zhou, *Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics*, *The 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, 2008.

[24] S. Lu, J. Tucek, F. Qin and Y. Zhou, *AVIO: detecting atomicity violations via access interleaving invariants*, *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, NY, USA, 2006, pp. 37-48.

[25] B. Lucia, J. Devietti, K. Strauss and L. Ceze, *Atom-Aid: Detecting and Surviving Atomicity Violations*, *The 35th International Symposium on Computer Architecture*, Beijing, China, 2008.

[26] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, *Pin: building customized program analysis tools with dynamic instrumentation*, *In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, USA, 2005.

[27] M. Naik, A. Aiken and J. Whaley, *Effective static race detection for Java*, *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM Press, New York, NY, USA, 2006, pp. 308-319.

[28] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards and B. Calder, *Automatically Classifying Benign and Harmful Data Races Using Replay Analysis*, *International Conference on Programming Language Design and Implementation*, 2007.

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

[30] S. Rajamani, G. Ramalingam, V. Ranganath and K. Vaswani, *ISOLATOR: Dynamically Ensuring Isolation in Concurrent Programs*, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, 2009.

[31] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. E. Anderson, *Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs*, *SOSP*, 1997, pp. 27-37.

[32] M. Vaziri, F. Tip and J. Dolby, *Associating Synchronization Constraints with Data in an Object-Oriented Language*, *The 33rd Annual Symposium on Principles of Programming Languages*, Charleston, SC, 2006.

[33] S. Woo, M. Ohara, E. Torrie, J. Singh and A. Gupta, *The SPLASH-2 Programs: Characterization and Methodological Considerations*, *In Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, 1995.

[34] Y. Yu, T. Rodeheffer and W. Chen, *RaceTrack: efficient detection of data race conditions via adaptive tracking*, *SOSP '03: Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, 2005, pp. 221-234.