

Detecting and Tolerating Asymmetric Races

Paruj Ratanaworabhan¹, Martin Burtscher², Darko Kirovski³, Rahul Nagpal⁴,
Karthik Pattabiraman⁵, and Benjamin Zorn³

¹Cornell University, ²The University of Texas at Austin, ³Microsoft Research, ⁴IISc Bangalore,
and ⁵University of Illinois at Urbana-Champaign

Abstract

Because data races represent a hard-to-manage class of errors in concurrent programs, numerous approaches to detect them have been proposed and evaluated. We specifically consider asymmetric races, a subclass of all race conditions, where a programmer’s thread correctly acquires and releases a lock for a given variable, while another thread causes a race by improperly accessing this variable. We introduce ToLeRace, a runtime system 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 case. We study the theoretical aspects of an oracle that knows exactly what type of interleaving has occurred. Then, we present a software implementation of ToLeRace on top of a dynamic instrumentation tool. We evaluate our implementation on multithreaded applications from the SPLASH2 and PARSEC suites and show that its overhead is acceptable, i.e., a factor of two on average.

Keywords: race detection and toleration, runtime support, dynamic instrumentation

1. Introduction

Race conditions are memory errors that occur when multiple threads read and write a memory location in an unspecified 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 [6, 12, 14, 15, 16, 17, 18, 20].

This paper presents ToLeRace, a runtime system that allows programs with concurrency errors to tolerate them and continue executing. Inspired by the DieHard system [2], which probabilistically tolerates memory safety errors, ToLeRace uses replication to deterministically or probabilistically manage asymmetric data races. 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. In this paper, we focus on the fundamental properties of ToLeRace and describe one possible software implementation. Our contributions 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 actually happen at runtime. It detects a race when the critical section in which the race takes 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 during debugging, testing, and in patching released executables.
- **Low overhead software implementation.** We demonstrate a software implementation of ToLeRace and show that it incurs a low overhead of about a factor of two on average.

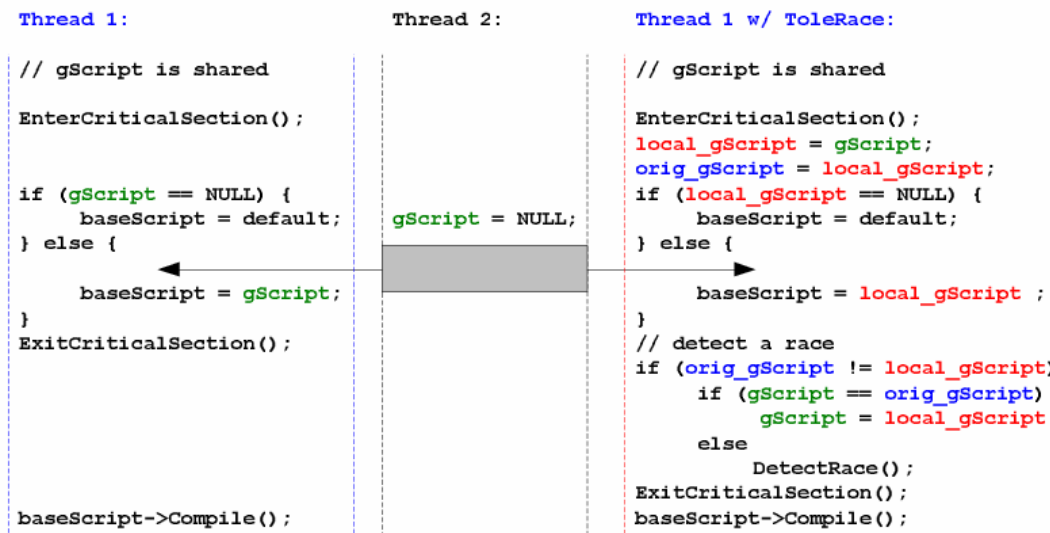


Figure 1: An example of an RwR race.

The example in Figure 1, inspired by a real race detected in the Mozilla application suite [12], 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_gScript`) 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 rendered 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 would 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 the 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 causes 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 the 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 later (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 use it. 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.

Table 1: Classes of race instances. The column marked “race” denotes whenever the schedule $T_1T_2T_1$ results in a race.

operation interleaving			race	operation interleaving			race	operation interleaving			race
T_1	T_2	T_1		T_1	T_2	T_1		T_1	T_2	T_1	
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

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 consider cases when a single variable is protected and accessed in a non-nested critical section. However, the theoretical framework developed here can be extended to cases involving multiple variables and overlapped critical sections. We refer readers to our technical report [10] for greater detail. We use \mathbf{l} , \mathbf{u} , \mathbf{r} , and \mathbf{w} to denote the fundamental functions over that specific variable. Finally, we use \mathbf{x} to denote a “don’t care” function that can be either a read or a write. A sequence of at least one read is denoted as \mathbf{r}^+ and a sequence of a non-negative number of reads as \mathbf{r}^* . 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, $T_1 = [\mathbf{l}_1\mathbf{r}_1\mathbf{w}_1\mathbf{r}^+\mathbf{u}_1]$ 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 distinguishes between sequences of operations of the same type. We denote one possible interleaved execution of critical operations of two threads $T_1 = [\mathbf{l}_1\mathbf{r}_1\mathbf{w}_1\mathbf{r}^+\mathbf{u}_1]$ and $T_2 = [\mathbf{w}_2]$ as the following sequence $S = \{\mathbf{l}_1\mathbf{r}_1\mathbf{w}_1\mathbf{w}_2\mathbf{r}^+\mathbf{u}_1\}$. Sequence S specifies that the write from the second thread occurred after the write in the first thread and thus causes a race condition.

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 malicious thread that does not have a critical section although it could access a shared variable.

Definition 1. A race condition represents any one of all possible execution interleavings of a set of threads $T = \{T_1 \dots T_N\}$ where at least one of the threads in 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 T have executed atomically with sequential memory consistency.

Note that this definition does not say anything about what happens to the values of shared variables in malicious threads. Because malicious 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 malicious) in T could execute but not affect the program computation state. In this case, we informally relax *Definition 1* to accept execution schedules where a subset of threads does not execute as correct.

For the purpose of this discussion, we assume the hardware supports sequential memory consistency semantics [11] and that Tolerace preserves these semantics.

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 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 ($\mathbf{r}+$), 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 $\mathbf{r}\mathbf{w}$ could occur in two versions: (i) \mathbf{w} is dependent upon the value retrieved by \mathbf{r} and (ii) \mathbf{w} is not dependent upon the value retrieved by \mathbf{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 $\mathbf{r}\mathbf{w}$ issued by the same thread we assume (i).

Table 1 tabulates all possible interactions between 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. 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. $X\mathbf{w}\mathbf{R} = \{\mathbf{I}_1\mathbf{x}+\mathbf{I}_2\mathbf{w}\mathbf{x}^*\mathbf{r}_1\mathbf{u}_1\}$. This case specifies that for any sequence of operations by T_2 that starts with a write and is followed by a read in T_1 , a race will occur.
- II. $\mathbf{W}\mathbf{r}\mathbf{W} = \{\mathbf{I}_1\mathbf{r}^*\mathbf{I}_2\mathbf{w}\mathbf{I}_1\mathbf{x}^*\mathbf{r}+\mathbf{I}_2\mathbf{r}^*\mathbf{I}_2\mathbf{w}\mathbf{I}_2\mathbf{u}_1\}$. 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. $\mathbf{R}\mathbf{X}\mathbf{w}\mathbf{W} = \{\mathbf{I}_1\mathbf{r}_1\mathbf{x}^*\mathbf{I}_2\mathbf{w}\mathbf{I}_2\mathbf{u}_1\}$. 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. $\mathbf{X}\mathbf{r}\mathbf{w}\mathbf{X} = \{\mathbf{I}_1\mathbf{x}+\mathbf{I}_2\mathbf{r}+\mathbf{I}_2\mathbf{w}\mathbf{x}^*\mathbf{I}_2\mathbf{u}_1\}$. This case specifies that any sequence that starts 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.

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) Consider a case where there is a single safe thread among K interacting threads. The $K-1$ malicious threads impart intervening sequences of operations $\mathbf{r}+$, $\mathbf{w}\mathbf{x}^*$, or $\mathbf{r}+\mathbf{w}\mathbf{x}^*$ 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 malicious 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 interacts with just a single malicious thread, and the resulting race instances can be classified by Table 1.

Now, consider another case where there are 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 have just considered and completes the proof.

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 2). 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 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 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 a 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 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.

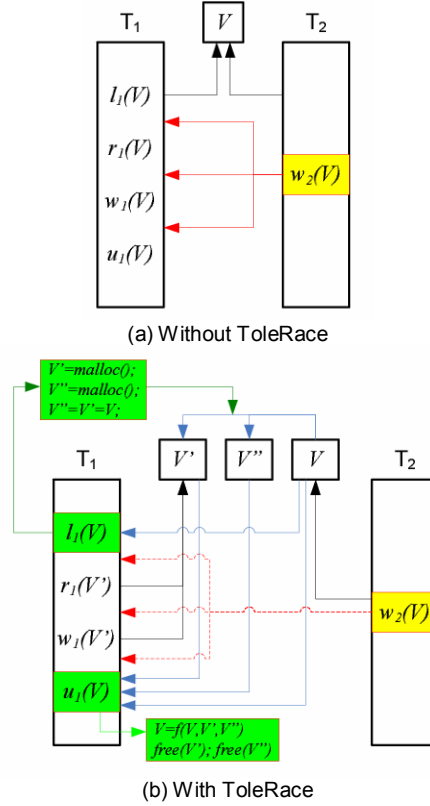


Figure 2: How ToleRace uses two additional copies of a variable to tolerate races.

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 case of race that has occurred, Table 2 summarizes the definition of f and indicates the cases that ToleRace correctly tolerates.

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	RrwX	false	custom f'	false	N/A

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

- IV_A: $RrwR = \{I_1r_{+1}r_{+2}w_2x_2^*r_{12}u_1\}$,
- IV_B: $WrwX = \{I_1w_1x_{-1}^*r_{+2}w_2x_2^*x_{12}u_1\}$, and
- IV_C: $RrwX = XrwX - \{RrwR \cup WrwX\}$

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 σ , the schedule of threads that ToleRace's result represents. Table 2 shows that the ToleRace oracle tolerates all races except sequences that belong to RrwX with the resolution function f defined by Table 2.

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 the three possible outcomes.

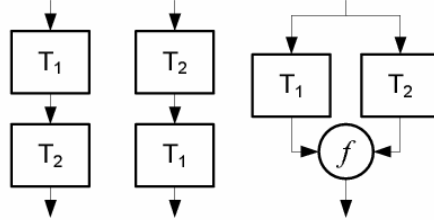


Figure 3: Atomic execution order enabled by ToleRace. Schedules T_1T_2 or T_2T_1 are correct, whereas the parallel execution T_1 and T_2 is a race.

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 interrupted.

As case RrwX cannot be tolerated by either of the outcomes that resolve cases I-IV_B, ToleRace will either raise an exception indicating that a race has been detected, or execute a programmer-defined resolution function, f' , when a semantically sound resolution function is provided. An example of an efficient f' that tolerates an 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.

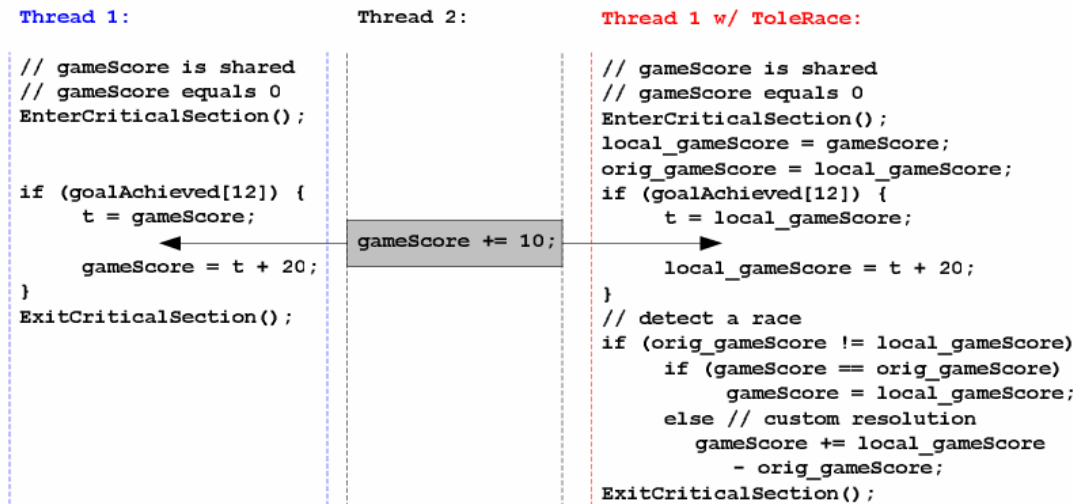


Figure 4: An example of how program semantics can define a custom resolution function f' that could tolerate RrwX races. The programmer knows the prior that all writes in the program to the shared variable are only additions and subtractions of constants.

As a custom resolution function cannot readily be implemented in general, we do not consider it in our software implementation described in the next section. We only implement the resolution function per Table 2 for race types I- IV_B and flag the IV_C case as detected races that cannot be tolerated.

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 and we expect other implementations to be more efficient.

4.1 Pin Tools

We chose to implement ToleRace on top of Pin [13], a dynamic instrumentation toolkit from Intel. Pin is available for a variety of operating systems and architectures, including the ubiquitous 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 while *library code* refers to all other code (which does not have to reside in a library though it often will).

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 is able to 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. 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. 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.

4.3 Implementation Details

This subsection describes the implementation of Pin-ToleRace whose framework is shown pictorially in Figure 5.

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 sec-

tion. The content is created by the first instruction that accesses a shared memory location. The role of each of these structures is explained next.

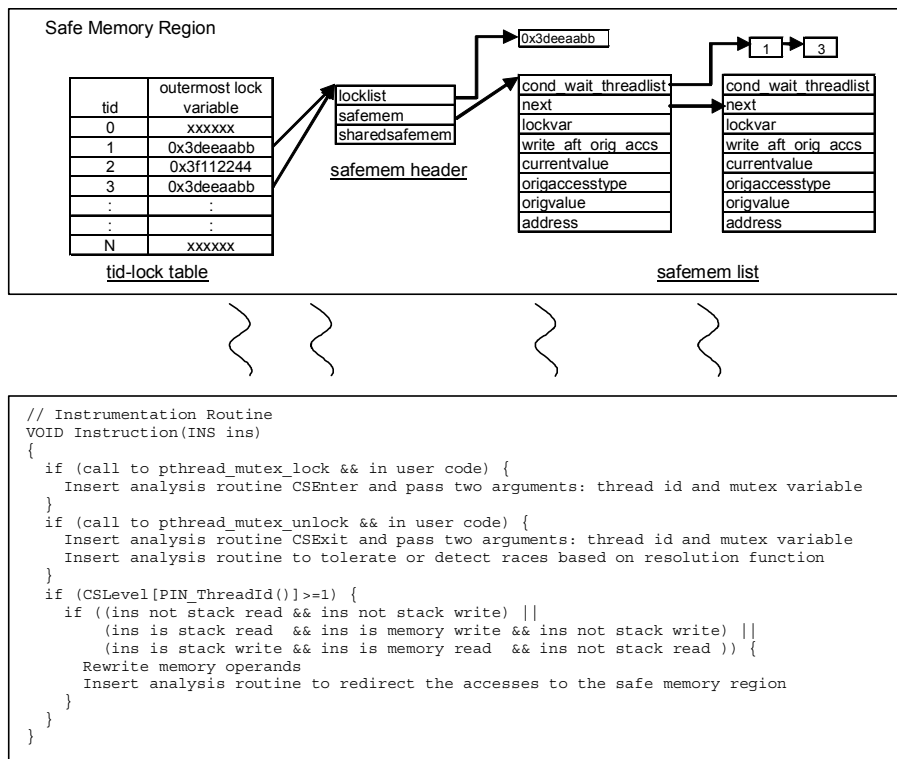


Figure 5: Pin-ToleRace Framework.

4.3.2 Identifying Critical Sections

A critical section is defined by a mutex variable and a pair of library calls to `pthread_mutex_lock` and `pthread_mutex_unlock` 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 that reside in this domain.

4.3.3 Instrumenting Instructions Accessing 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. The if conditions inside the check for `CSLevel` in Figure 5 accomplishes the latter. 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 current value field like in the found case. Although the instrumentation routine is a callback routine that is serialized under Pin and is called by multiple threads, it does not create a race condition. 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 for the purpose of handling 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 (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. For example, if the address field is 0x3a445500, the range for byte access matching is 0x3a445500 through 0x3a445503.

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. 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-Tolerant 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 to perform the safe memory access when the CSLevel is greater than one. 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 waiter counter.

Condition variables pose a bit of a complication to 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 waiter counter, Pin-ToleRace searches the tid-lock table for the lock variable, uses the safemem header associated with this lock variable, and increments the `sharesafemem` 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 malicious interleaving thread must contain a write. However, it cannot distinguish between the two write sequences, \mathbf{wx}^* and $\mathbf{r+wx}^*$. 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., $\mathbf{r+wx}^*$, which prevents it from tolerating type III races. Aside from this restriction, it tolerates the same race types as the oracle.

4.4.1 False Positives

There are inherently no false positives with Pin-ToleRace. When an `origvalue` field is compared with its corresponding true memory value and the two differs ($V \neq V'$), it is an asymmetric race by definition.

4.4.2 False Negatives

Pin-ToleRace produces a false negative for a race to a shared location when: a) the last write in the intervening sequence writes the same value as the value in the `origvalue` field, and b) immediately after the comparison of V and V' returns equal, the intervening sequence writes to V . Note that both a) and b) have a much lower probability of occurring than the unprotected race does.

5. Evaluation

5.1 Benchmarks

We use 13 applications from the SPLASH2 [19] and PARSEC [3] 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 malicious 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 pairs of `pthread_mutex_lock` and `pthread_mutex_unlock` calls. One of these programs, `dedup`, is a kernel while 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. The SPLASH2 programs are compiled with the `-O2` optimization level whereas the microbenchmarks and the PARSEC programs 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 report 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 malicious 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 malicious 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 malicious threads write wrong values to all of these shared locations.

For these tests, we know that the malicious 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 malicious thread’s action to have happened after the safe thread’s read-increment-write operations.

Our test setup uses five malicious 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 malicious thread that last ran.

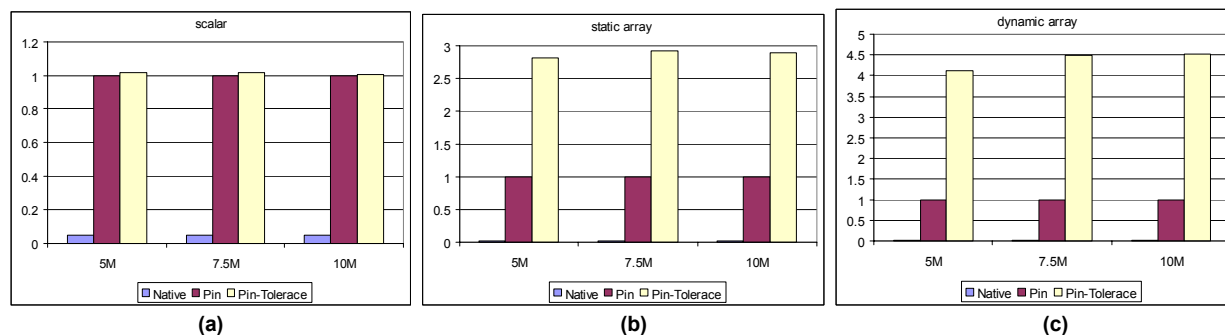


Figure 6: Overhead of Pin-Tolerace for scalar (a), static array (b) and dynamic array (c) for different numbers of iterations.

Figure 6 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 still 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 sce-

narios 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 overheads.

One additional point to note is that, with Pin-TolerRace, 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-TolerRace 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 3: Critical section characteristics.

	unique	nested critical section	total executed	num instrs (user)	dynamic instrs
cholesky	14	no	11,849	29	343,621
fft	10	no	55	17	935
lu	7	no	1,043	17	17,731
radix	9	no	51	17	867
barnes	10	no	1,098,771	94	103,284,474
ocean	26	no	3,335	17	56,695
radiosity	36	yes	1,739,512	18	31,311,216
water-spatial	16	no	853	13	11,089
dedup	7	yes	256,380	600	153,828,000
facesim	5	yes	10,161	46	467,406
ferret	4	yes	552,173	690	380,999,370
fluidanimate	11	no	4,359,405	13	56,672,265
x264	2	no	16,767	11	184,437

Table 4: 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 first characterizes the critical sections of the 13 benchmarks and then discusses the overhead of Pin-TolerRace 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 3. Note that we only study critical sections that reside in the user code, i.e., we exclude all library code.

The second column of the table 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. With our inputs, most benchmarks execute under a million “critical” instructions. Only barnes, dedup, and ferret execute over 100M. The numbers in this column exclude the instructions of any library routines called from the critical sections. All programs except ferret execute less than 1% of their dynamic user instructions in critical sections.

The fourth column of Table 3 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 probably 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. It seems that their small critical sections are being looped over.

Next, we look at the critical sections from the point of view of Pin-Tolerace. Table 4 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.

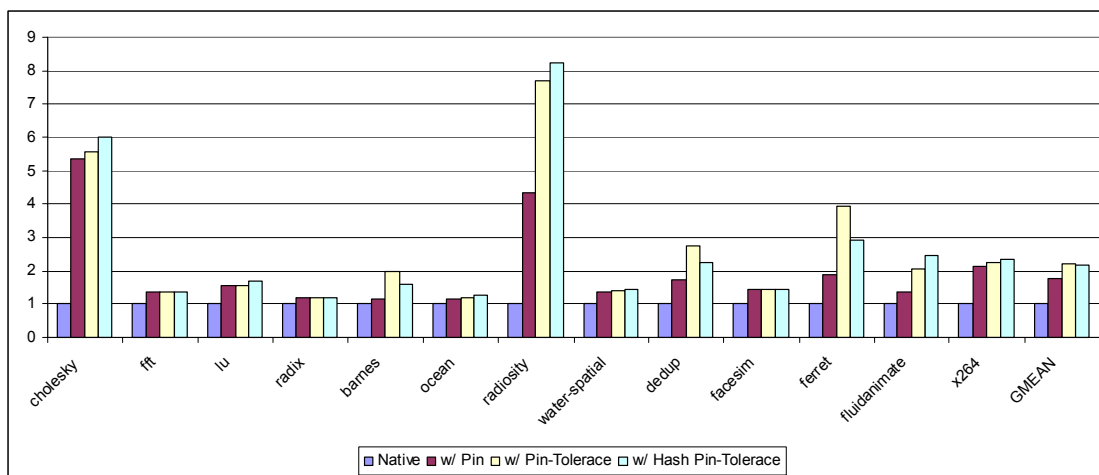


Figure 7: Pin-Tolerace Overhead.

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 collision in dedup and ferret.

Figure 7 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 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 for 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, there is a fixed minimum number of entries to process (proportional to the table size) whereas with the linked list 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. For the rest of the benchmarks, the critical section characteristics are quite the opposite. Those programs have small critical sections, and each critical section execution does not touch many unique shared locations. The linked list implementation is better suited for this type of critical section.

6. Related Work

Related race detection research includes both static and dynamic approaches. Static race detection relies on program analysis and either assumes existing [14] or defines new programming language semantics that help improve the static detection of races (e.g., Cyclone [8]). Static analysis techniques face several challenges. First, because many of the techniques are based on some form of model checking [9], 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 [6] and Houdini/rcc [7] 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 locksets [18]. Lockset analysis discovers where programs access shared variables without properly holding locks. 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 happens-before analysis [1], which identifies data accesses with no implied ordering. Combining locksets with a happens-before scheme results in higher precision dynamic race detectors [5, 4, 16, 20]. Even with refinements, the execution overhead of these approaches is typically larger than a factor of two. Furthermore, previous work focuses primarily on detecting races rather than tolerating them.

The AVIO system takes a training-based approach to identifying erroneous access interleavings [12]. After learning which interleavings are benign, the system deploys runtime checking to detect malicious interleavings dynamically. Without hardware support, the overhead of this checking is high. 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.

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 a software implementation. Our evaluation indicates that real applications can run on top of software ToleRace with acceptable overhead, i.e., about a factor of two on average.

8. References

- [1] 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.
- [2] E. D. Berger and B. G. Zorn, *DieHard: probabilistic memory safety for unsafe languages*, ACM SIGPLAN Notices, 41 (2006), pp. 158--168.
- [3] C. Bienia, S. Kumar, J. Singh and K. Li, *The PARSEC Benchmark Suite: Characterization and Architectural Implications*, Princeton University Technical Report TR-811-08, Princeton University, 2008.
- [4] 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.
- [5] T. Elmas, S. Qadeer and S. Tasiran, *Goldilocks: Efficiently Computing the Happens-Before Relation Using Locksets*, in K. Havelund, n. e. M. N'vu, G. Rosu and B. Wolff, eds., *FATES/RV*, Springer, 2006, pp. 193--208.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.

- [10] 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.
- [11] L. Lamport, *How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor*, *IEEE Transactions on Computers*, 46 (1997), pp. 779--782.
- [12] 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.
- [13] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klausner, 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.
- [14] 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.
- [15] R. O'Callahan and J.-D. Choi, *Hybrid dynamic data race detection*, *PPoPP '03: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM Press, New York, NY, USA, 2003, pp. 167--178.
- [16] 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.
- [17] P. Pratikakis, J. S. Foster and M. Hicks, *LOCKSMITH: context-sensitive correlation analysis for race detection*, *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM Press, New York, NY, USA, 2006, pp. 320--331.
- [18] 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.
- [19] 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.
- [20] 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.