Efficient Runtime Detection and Toleration of Asymmetric Races

Paruj Ratanaworabhan[1], Martin Burtscher[2], Darko Kirovski[3], Benjamin Zorn[3], Rahul Nagpal[4], and Karthik Pattabiraman[5]
[1]Kasetsart University, [2]Texas State University, [3]Microsoft Research, [4]Indian Institute of Science, [5]University of British Columbia

Abstract - We introduce 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 software implementations of ToleRace and evaluate them on multithreaded applications from the SPLASH2 and PARSEC suites.

Index Terms - Debugging aids, dynamic instrumentation, parallel programming, race detection and toleration.

1 Introduction

This paper tackles data race problems in lock-based parallel programs. It focuses on programs written in unsafe languages such as C or C++ that use add-on libraries for threading and synchronization. At present, a large installed code base of such programs exists and programmers continue to write parallel code in this paradigm.

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 [3] 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.

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.).

```
Thread 1:
// gScript is shared

Lock(A);
if (gScript == NULL) {
   baseScript = default;            Thread 2:
} else {
                              <---- gScript = NULL;
   baseScript = gScript;
}
UnLock(A);
```

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 follow 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 often 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. [22] 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.

This work presents ToleRace, a runtime system that not only detects asymmetric races but also tolerates them. ToleRace allows programs to continue executing in the presence of asymmetric races and possibly complete with a well-defined semantic. Inspired by the DieHard system [5], 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 can be compared to transactional memory (TM) [14]. 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. 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 [7].

This paper makes the following contributions:

- **Foundations for run time management of races.** We present a theoretical framework that investigates all possible interactions among safe threads that observe a proper locking discipline and unsafe threads that fail to do so. Then, we focus on cases where a race occurs, categorize them, and describe our race detection and toleration scheme for each category.
- **Precise race detection.** ToleRace identifies races that actually happen at run time. It detects a race when the critical section in which the race took place exits and, by design, never generates a false positive.
- **Low overhead software implementation.** We present three software implementations of ToleRace. Our first version uses a dynamic instrumentation-based approach and performs all analysis at run time. For the second version, we add a static program analysis phase to remedy the shortfalls in the first version. The third version is radically different from the first two. It is based on source-code modifications to implement ToleRace.

2 Characterizing Asymmetric Races

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 that there are two types of threads:

- a safe thread consisting of a single critical section, and
- a non-safe thread that might access a shared variable outside of a critical section or using the wrong lock to guard it.

Let r, w, and x denote read, write, and don't-care operations, respectively. An x don't-care can be either a read or write operation. Let lower case letters represent accesses of non-safe threads and upper case letters accesses of safe threads. r+ denotes a sequence of at least one read and r* indicates zero or more reads. The operators + and * are equally defined for writes and don't-cares. There are only three ways in which a sequence of operations from 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*). For the r+wx* sequence, we assume that w is dependent upon the value retrieved by r.

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.*

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. 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 outcomes 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 = X+ wx* R+X*. *This case specifies that any sequence of operations by* T_2 *that starts with a write and occurs after one or more arbitrary operations but before a read in* T_1 *causes a race.*
II. WrW = R*WX* r+ R*WX*. *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. RwW = R+X* wx* WX*. *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 = X+ r+wx* X+. *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* T_1 *operations.*

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. Direct result of combining cases from Table 1. □

There is previous work [18, 26] 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* on the safe thread. When these three sequences interleave, the resulting sequence still belongs

Table 1. Tabulating classes of race instances. Column marked "race" denotes if the schedule $T_1'T_2T_1''$ results in a race.

operation interleaving					operation interleaving					operation interleaving				
T_1'	T_2	T_1''	race	type	T_1'	T_2	T_1''	race	type	T_1'	T_2	T_1''	race	type
R+	r+	R+	false		R+	wx*	R+	true	I	R+	r+wx*	R+	true	IV_A
R+	r+	WX*	false		R+	wx*	WX*	true	III	R+	r+wx*	WX*	true	IV_C
R+	r+	R+WX*	false		R+	wx*	R+WX*	true	I	R+	r+wx*	R+WX*	true	IV_C
WX*	r+	R+	false		WX*	wx*	R+	true	I	WX*	r+wx*	R+	true	IV_B
WX*	r+	WX*	true	II	WX*	wx*	WX*	false		WX*	r+wx*	WX*	true	IV_B
WX*	r+	R+WX*	true	II	WX*	wx*	R+WX*	true	I	WX*	r+wx*	R+WX*	true	IV_B
R+WX*	r+	R+	false		R+WX*	wx*	R+	true	I	R+WX*	r+wx*	R+	true	IV_C
R+WX*	r+	WX*	true	II	R+WX*	wx*	WX*	true	III	R+WX*	r+wx*	WX*	true	IV_C
R+WX*	r+	R+WX*	true	II	R+WX*	wx*	R+WX*	true	I	R+WX*	r+wx*	R+WX*	true	IV_C

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.

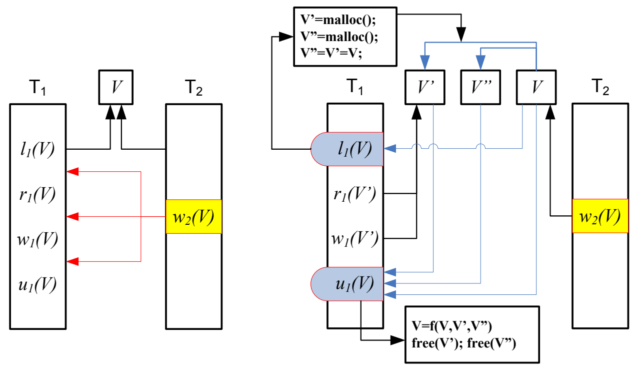
Now, consider multiple safe threads among the K interacting threads. Because safe threads, by definition, hold consistent locking for a given shared variable, only one can be in the critical section accessing this variable at a given time. This brings us back to the first case we just considered and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

3 The ToleRace Oracle

Having characterized asymmetric races, we now present our theoretical framework, the ToleRace Oracle, and describe how it handles all the race cases specified in Theorem 1.

The core of our approach to managing races 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.



(a) Without ToleRace (b) With ToleRace

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

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.

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	RwW	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	N/A	false	N/A

3.1 Tolerating and Detecting Races with the Oracle

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= R+ r+wx* R+,

IV_B: WrwX= WX* r+wx* X+, 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 with the resolution function *f* defined by Table 2 except sequences of the form RrwW.

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 the oracle ToleRace is used as a pure race *detector*, i.e., when the resolution function is turned off, we can reason about the situations in which it may produce false positives or false negatives. The oracle ToleRace inherently generates no false positives. When $V \neq V''$, an asymmetric race has occurred by definition. However, 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, i.e., the intervening sequence writes B and then A after the safe thread reads A. From the viewpoint of ToleRace, the clean copy appears to be untouched and ToleRace does not report a race. Surprisingly, although ToleRace does not detect this case, it tolerates it by scheduling the operations from the intervening sequence to have come before those of the safe thread. ABA will now appear as BAA.

b) there is a WrW race. ToleRace cannot detect this race case, but it again tolerates it.

3.2 Multiple Variables and Nested Critical Sections

So far, we have considered the oracle ToleRace in a multithreaded, single-variable, non-nested critical-section context. We now extend this framework to handle general cases, which may involve multiple variables and nested critical sections. Making local copies and executing the resolution function need to be done atomically for multiple variables. Nested critical sections share their local copies with the outer critical sections. However, they have their own resolution function to resolve races for their protected variables. When dealing with these general cases, the race toleration mechanism employed in ToleRace may lead to inconsistent execution. If this happens, ToleRace prevents the shared variable reordering by acting as a race detector only.

Theorem 3. 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 the original program order. If there are data dependencies among the operations that must be observed, ToleRace disallows such reorderings and reverts to detection mode.

Here we outline the proof of Theorem 3. 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).

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 first two columns show the nine possible combinations. The third column indicates whether ToleRace reorders the intervening operations to P and Q. This follows directly from the resolution function in Table 2. For example, in the second row of Table 3, there is a reordering to make it appear that the Q intervening sequence comes before the P intervening sequence since r+ sequences will always be scheduled to appear before any critical section operations (case II in Table 2) whereas the reverse is true for wx* sequences (case I and III in Table 2). The fourth column specifies whether 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 the variables and the resolution function allows out-of-order execution.

Table 3. Possible 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

The oracle ToleRace we have described represents a theoretical framework that cannot be fully realized in practice. The next three sections describe software implementations that approximate it. Although the framework permits both software and hardware implementations, a software approach may be more appealing as it can be deployed immediately. Section 4 describes an initial implementation that is restricted and suboptimal. It serves as a baseline for other implementations to benchmark against. Section 5 presents an improved version that addresses the shortfalls in the initial version. It approximates what would likely be deployed in practice. Section 6 investigates the idealized version of software ToleRace. It assumes an oracle compiler and the availability of the program's source code.

4 Software ToleRace: A First Version

This section discusses the initial version of software ToleRace that is non-optimal and possesses some inherent restrictions. This first version makes all decisions at run time and does not perform any static program analysis. It allows us to gauge an upper bound on the software ToleRace overhead. In the next section, we will present an improved implementation that incorporates an additional static analysis phase to generate hints for the runtime, allowing it to make better decisions. This improved version has a lower overhead and eliminates all the restrictions of the first version.

We implement ToleRace on top of Pin [20] running on x86 Linux systems. Our parallel applications are written in C/C++ and use the pthreads library for synchronization operations. However, we believe the framework described here generalizes to other platforms and threading libraries. In the rest of this paper, we apply software ToleRace to critical sections in the *user code* whereas critical sections in the *library code* receive no ToleRace protection. We assume that we can readily distinguish the two code regions. For example, in an x86/Linux executable compiled to use shared libraries, all routines in the .text section are considered user code (see some exceptions in Section 5.3.2). Library code is not present at load time and is discovered only at run time via the procedure linkage table in the .plt section.

4.1 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 data.

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 outside of critical sections are instrumented. The instrumented code is generally referred to as the analysis routines. It 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.

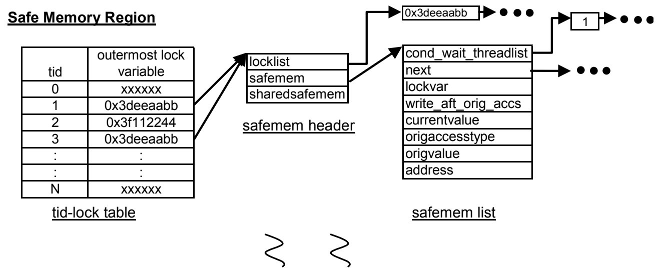
For this first version of Pin-ToleRace, we assume that code segments touched while executing in a critical section can be reached from outside of critical sections only after they have already been instrumented inside of the critical section. We will revisit this restriction in Section 5 when we introduce the improved version of Pin-ToleRace. For now, it suffices to say that the presence of Pin's code cache in its dynamic translator engine necessitates this restriction.

4.2 Implementation Details

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

4.2.1 The Safe Memory Region

The safe memory contains three main data structures: a thread ID (tid) lock mapping table, a `safemem header`, and a list of `safemem nodes`. The first two structures are required to handle condition variables and nested/overlapped critical sections. If the program has neither, i.e., it contains only non-nested critical sections, only the `safemem list` is necessary. In a `safemem node`, the fields `origvalue, origaccesstype, currentvalue,` and `write_aft_orig_accs` are used by the resolution function to tolerate races. The `lockvar` field indicates the lock variable protecting a given memory location. It is used in conjunction with `locklist` in the `safemem header` to correctly resolve races in nested/overlapped critical sections. `cond_wait_threadlist` and `sharedsafemem` track the number of outstanding threads waiting to be signaled. The `tid-lock table` associates the ID of a thread executing inside of a critical section with the outer lock variable. When multiple threads can be inside of a critical section at the same time, there will be a sharing of the `safemem` structures as shown in Figure 4. The role of each of these structures and their associated fields are explained next.



```
// Instrumentation Routine
VOID Instruction(INS ins) {
  if (call to pthread_mutex_lock && in user code) {
    Insert analysis routine CSEnter
  }
  else 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 the accesses to the safe memory.
    }
  }
}
```

Figure 4. Pin-ToleRace framework.

4.2.2 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.

4.2.3 Instrumenting Accesses to Shared Memory

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. 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.

4.2.4 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 4.3 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 4.2.5 and 4.2.7).

4.2.5 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. Note that the inner mutex lock variable itself cannot be part of the protected shared variable under the outer mutex. If it could, the safe thread might be left spinning forever on a local copy of the inner lock variable that no other thread can reset (i.e., unlock), thus leading to deadlock.

A critical section that executes inside another critical section never creates a new safemem list; 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 may be 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.

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.

4.2.6 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. As we want to protect only user routines, Pin-ToleRace must distinguish them from library routines. Note that Pin itself instruments every instruction dynamically and has no knowledge if the instruction comes from a user or library routine. Shared memory accesses in user code need redirection to the safe memory whereas those in library code need not. 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 disallow the addition of new nodes to the list.

4.2.7 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. Our current implementation does not support waits on more than a single mutex variable.

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.

4.3 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 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 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. When dealing with intolerable races, Pin-ToleRace reports the race and halts program execution.

4.4 Evaluation

4.4.1 Benchmarks

We use 13 applications from the SPLASH2 [27] and PARSEC [6] benchmark suites to evaluate Pin-ToleRace. We also developed three microbenchmarks to stress-test a safe thread's 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. For each of the eight programs, the default inputs were used. However, we increased some of the input sizes to lengthen the program run times. We selected the five programs from the PARSEC suite that use the pthreads library. They are run with the simlarge inputs.

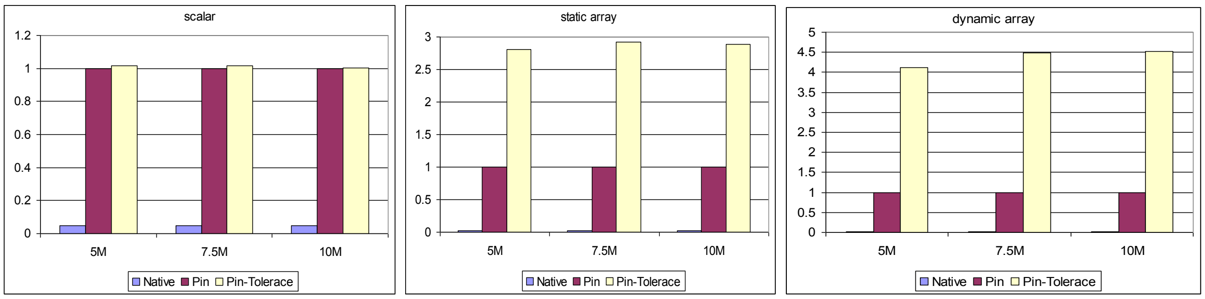
4.4.2 System and Compiler

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.

4.4.3 Stress Test

The stress tests demonstrate Pin-ToleRace's ability to tolerate races of the form RwW. In this test, 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,



(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.

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 RwW 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 locations change to the thread ID of the non-safe thread that ran last.

Figure 5 reports the overhead of Pin-ToleRace for tolerating these RwW races. It is normalized to the run time 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 for this high overhead 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 always busy tolerating races inside a critical section. The next section shows that real applications have critical section characteristics that lead to a much lower Pin-ToleRace overhead.

4.4.4 Benchmark Applications

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

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.

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 locations accessed to possibly shared locations per critical section by each thread.

	unique locations accessed	
	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

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

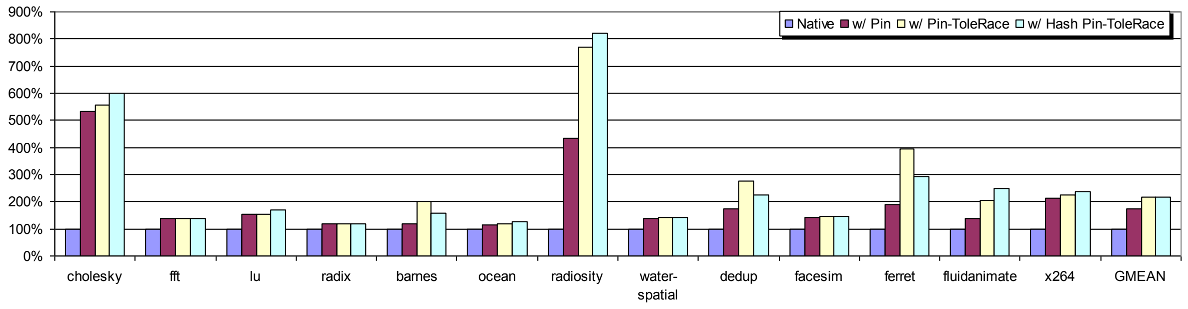


Figure 6. Normalized execution time of Pin-ToleRace.

`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 locations accessed. 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 locations accessed 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 locations accessed seems to be in proportion with the number of instructions executed per critical section.

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 chose this size to minimize the collisions in `dedup` and `ferret`.

Figure 6 presents the results. The timing measurements are normalized to the native run time. 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. Much of this overhead is an artifact of using Pin; the slowdown due to Pin alone is 1.8X. If we consider the overhead of Pin-ToleRace relative to the Pin runs, it is only about 24%. By adding static analysis (see Section 5) or hardware support, it should be possible to reduce the overhead. Note that when a program runs under Pin-ToleRace, it effectively runs with a race detector. Therefore, the results in Figure 6 include the detection overhead. When an intolerable race is detected, Pin-ToleRace simply stops the program and reports the race.

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, 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.

Note that it is sufficient to measure Pin-ToleRace performance with no-race execution since the cost of executing race-free is always equal to or greater than the cost of tolerating races. With no-race execution, when there is a write access to a shared variable, Pin-ToleRace needs to writes back the local copy V' to the actual shared location V. When it tolerates a race, however, sometimes no such write back is necessary since the intervening write update by an unsafe thread to V might already be legitimate to pass on.

.

5 Improving the Initial Pin-ToleRace Version

This section describes how to implement a more efficient Pin-ToleRace. The improved version also eliminates the restriction mentioned in Section 4.1.

5.1 Inefficiency in Pin-ToleRace

The sources of inefficiency in the initial Pin-ToleRace can be attributed to the following.

Provision for generality: As the initial Pin-ToleRace assumes no a priori knowledge when encountering a critical section, it needs to be conservative and has to provision for the general case. Thus, the system creates the full structure of the safe memory every time a critical section is executed. However, if a critical section is non-nested and does not have any condition variables, the `tid-lock table` and the `safemem header` become unnecessary and introduce two extra levels of indirection when accessing the `safemem nodes`.

malloc and free operations: As we postpone all the analysis of possibly shared memory locations until run time, our safe memory needs to be able to grow dynamically to account for those locations that are generated on the fly. It is natural to use malloc and, hence, its corresponding free operations for this purpose. However, malloc and free are rather heavyweight calls and are not easily amortized in small critical sections. Worse yet, as these small critical sections are being looped over, the call overhead can add up significantly. Ideally, if we can bound the number of possibly shared locations, we can resort to a stack-based allocation style where the corresponding malloc and free operations are reduced to adding and subtracting a value from the stack pointer.

Fixed data structure for the safe memory: With the previous implementation of Pin-ToleRace, the safe memory data structure is fixed throughout the entire run of a program. This may not be optimal for an application that contains both short and long critical sections. We, therefore, want to selectively assign the right safe memory structure to each critical section.

5.2 Inherent Restriction in Pin-ToleRace

Figure 7 shows a situation where the assumption we made for the initial Pin-ToleRace in Section 4.1 may not hold. Statements 1 through 4 may get executed inside of a critical section, i.e., when cond2 is true, or outside of a critical section, i.e., when cond2 is false. In addition, the function f() may be called from within a critical section (line 7) or from without (line 2).

```
1:  while (cond1) {
2:    f();
3:    if (cond2)
4:      pthread_mutex_lock(&mutex);
5:    statement 1;
6:    statement 2;
7:    f();
8:    statement 3;
9:    statement 4;
10:   if (cond2)
11:     pthread_mutex_unlock(&mutex);
12: }
```

Figure 7: An example illustrating how the assumption in the first version of Pin-ToleRace may be violated.

The Pin's code cache poses some complication to the situation depicted in Figure 7. First, let cond1 be true and cond2 be false. Statement 1 through 4 and function f() get executed outside of a critical section and their translated execution code is stored into the code cache. Then, let cond1 stay the same and cond2 become true. The four statements and f() now execute inside of the critical section. This time, however, the executed code, in particular, the instructions that may access shared memory may not get the proper operand rewriting and instrumentation. When the runtime system consults the code cache, it may find and use instances of the translation of the first execution, causing incorrect ToleRace operation as the previously translated code does not rewrite memory operands and redirect accesses of shared memory locations. In general, in the presence of a code cache, code segments that can potentially be executing both inside and outside of critical sections may cause incorrect run time behavior in Pin-ToleRace.

Aliasing caused by indirect calls: Indirect calls inside critical section may have their targets alias with functions that can both be executed inside and outside of critical sections. Furthermore, indirect calls outside of critical sections can also be problematic as their targets may alias with code that executes inside of critical sections. These scenarios bring back the correctness issue we have just discussed above.

5.3 Static Program Analysis

In this section, we discuss static program analysis whose role is to generate and pass additional information and hints to the runtime systems. Such information will be used to remedy both the inefficiencies as well as the restrictions in the first version of the Pin-ToleRace. Figure 8 shows a block diagram for the static analysis phase. The input program is first passed into a call graph construction module. This module produces a graph representation of all calls in the program; every function is a node in the graph and there is an edge from function X to function Y if X calls Y. This call graph information together with the original program are in turn fed into the second module that traverse every critical section in the program. The output from this second module is a candidate list of instructions that potentially access shared memory locations inside critical sections. These modules and their interactions are described in detail below.

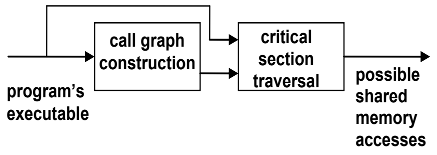


Figure 8: Static program analysis phase.

5.3.1 Assumptions about the Input Program

We assume that the program's executable contains all the user code and is available to us. The corresponding source code, however, may or may not be available. We assume that the program is compiled to use shared libraries. While the library source code is not available to us, the library's function prototypes are. That is we are fully aware of the interface given to the user, i.e., the number and type of parameters for all library calls are known. Threading and synchronization libraries (pthreads in our case) are also parts of the shared libraries.

5.3.2 Static Call Graph Construction

Below are the details on how the call graph construction module functions.

Input: The call graph module takes the program's executable as its input.

Processing: We use a two-pass algorithm. During the first pass, we traverse the program's executable and collect target addresses and possibly names of the user routines. We obtain such information from examining the .text section of the program. We eliminate certain routines that are not actually parts of the program, but get put in per operating system requirement, for example, `call_gmon_start`. These target addresses become nodes of the call graph to be constructed in the next step. In addition, we also gather target addresses and possibly names of the shared libraries including pthreads libraries. Such information is manifested in the procedure linkage table, which is contained in the .plt section of the executable. Note that we deal with x86/Linux platforms here; others may have different executable formats and conventions.

After we have collected all the necessary information in the first pass, in the second pass, we traverse the .text section to build a call graph. We walk each routine in the section one by one. For a given routine, we traverse every instruction in the routine from start to end in static program order. We search for calls to other routines. If a call is found, we check its target and create an edge from the current (calling) routine to the called routine. When examining each routine, we also gather other information required by the analysis in the next module (see output below). Note that we only deal with a call whose target is known at compile time. We discuss handling of indirect calls in Section 5.3.4.

After the call graph has been constructed, we generate a call chain for each routine. A call chain for a particular routine gives all the user routines that can be reached by initiating a call to the said routine. The chain is generated by traversing the call graph given the said routine as the starting node.

Output: After processing, we have information about each routine in the .text section, which represents a user routine. For each routine, we are able to tell:

- its call chain
- its list of calls to shared libraries
- its instructions that may access shared memory
- if it contains indirect calls

5.3.3 Static Critical Section Traversal

The purpose of this module is to identify all instructions that may access shared memory locations and are reachable from critical sections.

Input: The module takes the original program and the output from the call graph construction module as its inputs.

Processing: At the heart of the processing stage is the critical section traversal routine. This function gets invoked when a call to pthread_mutex_lock routine is found while we traverse the .text section of the program. The first action is to advance to the next instruction and mark the instruction as visited. It then recursively traverses instructions in the critical section. The recursion terminates when the routine finds all unlocks to match the number of locks found along a possible execution path.

When we encounter a conditional branch, we traverse the fall through path first, check if the branch target instruction has been visited, and if not traverse the target path accordingly. For the unconditional branch case, we need to only traverse the target path if the target instruction has not already been visited. In both cases, whenever we encounter a branch target address that is less than the current branch address, i.e., a back edge, we check if this forms a loop and whether there are instructions potentially accessing shared memory locations in the loop. The loop analysis information will be used to decide if malloc/free calls can be eliminated as well as to select a suitable data structure for the safe memory (Section 5.3.4.).

For a critical section that contains calls to user routines, we also need to include the candidate instructions from the called routines. We first consult the call chain of each called routine. Then, we obtain the list of candidate instructions from all routines in the call chain. The call chain and the list of candidate instructions are taken from the output of the previous module.

Output: After we traverse every critical section in the program, we produce a list of addresses of instructions that may execute inside of critical sections and access shared memory locations. We also obtain the following information about each critical section in the program:

- its list of calls to shared libraries
- if it contains indirect calls
- if it may access shared memory inside loops
- if it contains condition variables
- if it contains overlapped critical sections
- if it contains statically nested critical sections
- if it contains dynamically nested critical sections

5.3.4 Putting It All Together

This section describes how we use the result of the static program analysis to remedy the inefficiency and restrictions in PinToleRace. First, we address the inefficiency.

Addressing provisions for generality: This inefficiency is caused by uniformly implementing the full safe memory structure in every critical section. With the analysis, we can tailor the safe memory to suit a particular critical section, i.e., each critical section implements only the parts of the safe memory that are necessary for its correct operation. We need to know whether a given critical section contains condition variables, overlapped critical sections, or nested critical sections. For example, if the critical section contains none of the above, we can eliminate the `tid-lock table`, the `safemem header`, and the `lockvar` field. This allows us to access the `safemem node` directly without any extra indirection, which should improve the efficiency of the safe memory accesses.

Eliminating malloc/free calls: Generally, if we can bound the number of shared memory locations touched when a given critical section is executed, we can use stack-style memory allocation in place of malloc and free calls. This allows us to replace the costly call overhead with simple stack pointer operations. If the analysis result for a critical section indicates that there are no accesses to shared memory locations inside loops, the number of locations touched is bounded. With stack-based allocation, we preallocate a chunk of memory for every thread when it starts. In setting the chunk size, we need to consider all the critical sections whose shared memory accesses can be bounded, find the maximum number of bound accesses, and set the chunk size accordingly.

Suitable data structure for the safe memory: As previously noted, for long critical sections, we prefer a hash table structure, whereas for short critical sections, a linked-list structure is more efficient. We approximate these characteristics from the analysis result by saying that long critical sections may access shared memory inside of loops, whereas short critical sections never access shared memory inside of loops. Note that we use the same type of analysis here as we did when trying to eliminate malloc and free calls. These two optimizations, eliminating malloc/free and using an optimized safe memory structure, go hand in hand. Whenever we encounter critical sections

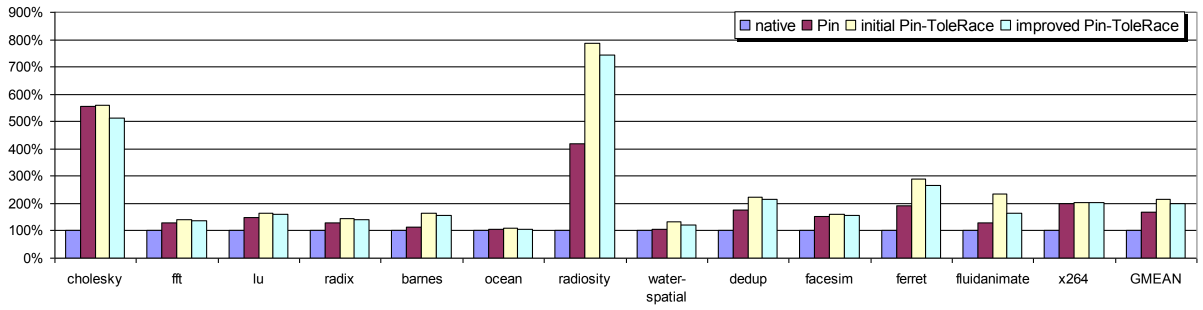


Figure 9: Normalized execution time for the improved version of Pin-ToleRace.

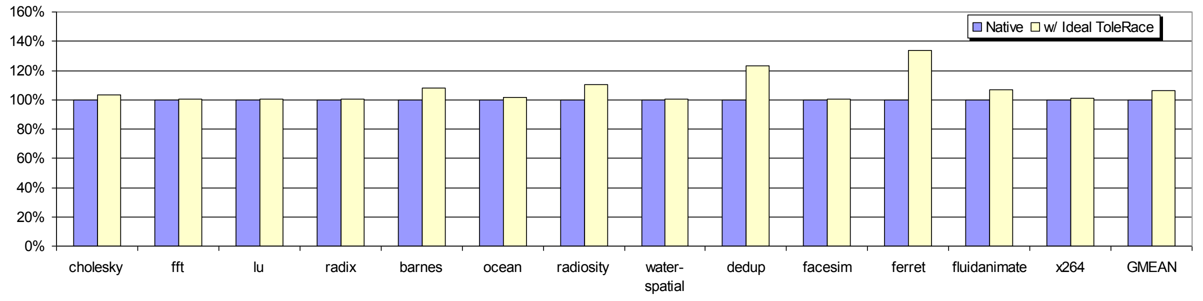


Figure 10. Normalized execution time of ideal software ToleRace.

that may never loop over shared memory accesses, we eliminate malloc/free calls, i.e., using stack-based allocation and choose a linked-list structure. Otherwise, we cannot avoid malloc/free completely and select a hash table structure.

We now turn to the restrictions in the first version of Pin-ToleRace. All the analysis that we have done enables us to solve the situation depicted in Figure 7. We are able to statically identify code segments that may execute inside critical sections and access shared memory locations. The critical section traversal module performs the analysis intra-procedurally while the call graph extends the analysis inter-procedurally, enabling whole program analysis. With the static analysis hints, the ToleRace runtime guarantees correctness even in the presence of the Pin's code cache. It instruments said code segments while they execute *both* inside and outside of critical sections. Note that this is in contrast to the initial Pin-ToleRace, which performs instrumentation only when the program executes inside of critical sections.

Handling indirect call aliasing: Because we have identified the code segments that may execute inside of critical sections upfront, aliasing from indirect calls executed exclusively outside of critical sections is not a problem. If such aliasing occurs, the runtime will correctly perform instrumentation at the instance the aliasing takes place.

What if we encounter indirect calls inside of a critical section, i.e., the critical section and the routines in its associated call chain contain indirect calls? Unfortunately, this situation cannot be solved completely with static program analysis. We simply do not know the targets of such indirect calls until run time. Therefore, any successful solutions to this problem inherently require the help of the ToleRace runtime. One possible solution is to keep track of all (user) routines executed outside of critical sections that have been translated by the just-in-time compiler. Once an indirect call is reached while executing inside a critical section, we instrument an analysis routine to search all the routines that have been translated, and, hence, reside in the code cache. If there is any aliasing, we flush the code cache so that the aliased routine is correctly instrumented.

Table 6: Critical sections with properties given in each column for each application.

applications	total	statically nested	statically overlapped	condition variables	indirect calls	shared mem. accesses in loops	user routine calls
cholesky	14	0	0	4	0	1	3
fft	10	0	0	7	0	0	1
lu	7	0	0	5	0	0	1
radix	9	0	0	7	0	0	1
barnes	13	0	0	6	0	2	5
ocean	25	0	0	20	0	0	1
radiosity	43	0	0	5	0	1	10
water-spatial	20	0	0	9	0	0	4
dedup	10	0	0	9	0	4	0
facesim	10	1	0	3	0	1	5
ferret	12	0	0	12	0	11	11
fluidanimate	11	0	0	0	0	0	0
x264	2	0	0	2	0	1	0

So far, we have been concerned only with indirect call aliasing within user code. However, whenever we discover a library call that may execute inside of critical sections, we also need to worry about indirect call aliasing coming from the library code. To tackle this problem, we check if the library call passes function pointers as callback arguments. If so, we hint to the ToleRace runtime to instrument these callback functions to use the safe memory. We assume that we have complete knowledge about these callback functions (cf. Section 5.3.1) so that we can statically identify them.

5.4 Results and Discussion

Table 6 shows characteristics of the critical sections in each benchmark application, i.e., the results from the static program analysis described in the previous section. The first column of the table gives the total number of critical sections discovered statically. This result is compatible with that given in Table 4. Apparently, certain critical sections in some applications never get executed, for example, we statically found 43 critical sections in `radiosity`, but only 36 of which are executed (see Table 4) with the given input.

The programs in the two benchmark suites we consider do not have indirect calls in critical sections or overlapped critical sections. This frees us of worry over indirect call aliasing and allows us to get rid of the `safemem header` structure. Hence, the improved Pin-ToleRace version should run more efficiently with these benchmarks. Most critical sections in some kernels of the SPLASH2 suite, `fft`, `lu`, and `radix`, contain condition variables. They are mainly there to support barrier-style synchronization. Similarly, in the PARSEC suite, almost all critical sections in `dedup` and `ferret` have condition variables. They are there to support pipelined-style parallelism. `facesim` is the only benchmark with a statically nested critical section. All critical sections in `fluidanimte` are simple in the sense that they are non-nested, do not contain any condition variables, and do not have any direct or indirect calls.

Figure 9 compares the overhead of the improved version of Pin-ToleRace against that of the initial version, bare Pin, and native execution. Note that the improved and the initial versions cannot be compared directly as the latter suffers from some restriction whereas the former does not (cf. Section 4.1 and 5.2). `fluidanimate` benefits most from the static analysis. Since it contains only simple critical sections, we can eliminate all the safe memory structures except the `safemem nodes` themselves. In addition, we can bound the shared memory locations for all the critical sections, allowing us to use stack-based allocations in place of malloc. Benchmarks such as `fft`, `lu`, `radix`, `ocean`, `water-spatial`, and `facesim` do not get significant benefit from the static hints as these programs spend very little time in critical sections.

6 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 run time. 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 10 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 the other applications and has many run time allocated shared variables. Consequently, it incurs the highest overhead. `dedup`, which has the second highest overhead, has similar characteristics. Most of the applications, however, incur less than 1% overhead with the ideal software ToleRace.

7 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 [21]) or defines new programming language semantics that help improve the static detection of races (e.g., Cyclone [12]). Static analysis techniques face several challenges. First, because many of the techniques are based on some form of model checking [13], 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 [10] and Houdini/rcc [11] 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 [25]. 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 [2]. Combining locksets with a happens-before scheme results in higher precision dynamic race detectors [8, 9, 23, 28]. Even with refinements, the execution overhead of these approaches is typically larger than a factor of two. 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 [16] and Sun's Thread Analyzer [15], which are commercial tools capable of locating data races in concurrent programs. Both tools suffer from a high memory footprint and run time overhead and are, thus, primarily used for software testing.

Atomicity violation is another important class of concurrency errors. It can be addressed statically [4] or dynamically. The AVIO system [18] 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. [19] offers to tolerate some degree of atomicity violation with implicit atomicity by grouping consecutive memory operations into atomic blocks.

Vaziri et al. [26] 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, which we do not consider. Their approach to race detection requires source-code annotation and targets safe language environments.

Kiena et al. [17] 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. [24] 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.

8 Conclusions

This paper introduces ToleRace, a runtime system that uses data replication for detecting and tolerating asymmetric races. We have presented a theoretical framework as well as three software implementations, which we evaluated on 13 real parallel applications from the SPLASH2 and the PARSEC suites.

References

[1] M. Abadi, T. Harris, and M. Mehrara, *Transactional memory with strong atomicity using off-the-shelf memory protection hardware, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, 2009, pp. 185-196.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] http://www.intel.com/cd/software/products/asmo-na/eng/286406.htm.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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 (PLDI)*, 2007.

[23] 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.

[24] 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 (ASPLOS)*, 2009.

[25] 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.

[26] 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.

[27] 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.

[28] 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.