Position Paper—ToleRace: Tolerating and Detecting Races

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

Abstract

This paper introduces ToleRace, a software tool that increases the reliability of multi-threaded programs by tolerating or detecting race conditions. ToleRace modifies the implementation of critical sections at runtime to provide the following benefits. ToleRace allows programs with certain classes of races to operate as though the race did not exist. ToleRace probabilistically allows programmers to detect many of the remaining races when they happen, with low performance overhead. ToleRace achieves its ability to tolerate and detect races by judiciously duplicating shared data inside a critical section, thereby providing an illusion of atomicity when the shared data is updated. Our early experiments reveal that the performance overhead of ToleRace is considerably lower than existing dynamic race detection tools.

1 Introduction

With increasingly concurrent hardware, the challenging task of writing correct concurrent software becomes more important. Race conditions are memory errors that occur when multiple threads read and write a memory location in an underspecified order. Because race conditions depend on the interleaving of the memory operations of individual threads, they are notoriously difficult to reproduce and represent a major obstacle in the task of writing correct concurrent programs. This problem has been investigated heavily in recent literature (e.g., [3, 6]).

Current approaches to solving the problem of race conditions focus on the problem of race detection and face three significant obstacles. First, both static and dynamic methods for race detection can produce false positives that significantly reduce their effectiveness in practice. Because race conditions are difficult to reason about, determining if a race reported by a detection tool is a real race is time consuming and counter-productive. Second, race detection tools based on dynamic detection can have lower false positive rates, but also increase execution time significantly (ranging from 2x to 30x slowdown without hardware support) [2, 7, 8]. Third, when a true race is detected, it must still be fixed, requiring code changes that may introduce new errors. For example, fixing a race condition may require additional synchronization constructs, and the incorrect use of such constructs can result in deadlock. Programmers are faced with the choice between allowing a race that occurs infrequently and creating potential new synchronization problems that have broader impact. In practice, for commercial software, such races can require many months to correct.

In this paper, we present ToleRace, a runtime system that dynamically attempts to reduce the likelihood of a race condition. The contributions of ToleRace include:

- **Tolerates race conditions when possible, otherwise reports with high probability** Unlike prior work, ToleRace allows programs with races to tolerate their existence by increasing the likelihood that races will not cause incorrect program interleavings. Increasing a program's tolerance to races reduces the need for the race to be debugged and patched. In many cases where ToleRace cannot tolerate a race condition, it detects and reports that races have occurred.

- **Precise detection.** ToleRace only identifies races that happen in practice. ToleRace detects a race has happened when the critical section where the race takes place exits.[1]

- **Proactive resolution of dynamically detected races** ToleRace enables the programmer to add compensation code at the exit of a critical section with an objective to design a custom resolution of a detected race. This allows the programmer to tolerate a suspected race in the software without ever identifying the exact source of the bug.

- **Low overhead** ToleRace adds overhead at runtime only to code that executes within a critical section and only to code that potentially modifies a shared variable.

2 ToleRace in Practice

ToleRace uses replication and redundancy to provide error tolerance and detection, and is inspired by DieHard [1] and Samurai [4, 5], two runtime systems that use replication to achieve the same goal. ToleRace creates the illusion 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.

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 tolerating and preventing other races (e.g., where neither thread uses a lock to protect the shared variable), asymmetric races are important for several reasons. First, most code is written correctly—in many cases local reasoning about concurrency, including taking proper locks has been done correctly, leaving a majority of remaining concurrency errors as asymmetric races.

[1]ToleRace reports no false positives using a definition commonly used in the literature, however, ToleRace may report a race for thread interleavings that without ToleRace would not exhibit a race but that would exist in other possible interleavings.

Second, asymmetric races are caused when software evolves and assumptions are invalidated. For example, code might be developed with the assumption that application initialization never occurs in a multi-threaded context. However, new code might be introduced (e.g., a second start-up thread) that violates the original invariant.

```
Thread 1:                         Thread 2:

// gScript is shared

EnterCriticalSection();           ...
if (gScript == NULL) {            gScript = NULL;
   baseScript = default;          ...
} else {
   baseScript = gScript;
}
ExitCriticalSection();
...
baseScript->Compile();
```

Figure 1: Example **RwR** Race

Figure 1 is an example inspired by a real race detected in the Mozilla application suite [2]. In this example, we see that Thread 1 is correctly using a critical section to protect its read accesses to the shared variable `gScript`. Thread 2 is incorrectly updating `gScript` without a lock, creating an asymmetric **RwR** race (where Thread 1 is reading and Thread 2 is writing). The race occurs infrequently, when Thread 2's update is interleaved between the test for NULL and the `then` part of the conditional in Thread 1.

While we envision a number of different implementations of ToleRace, ranging from pure software to hardware-assisted, in this paper we outline a low-overhead software implementation based on only modifying the code in critical sections. ToleRace modifies the critical section to determine if a race has occurred, identifies possible races that can be tolerated, and reports or allows the programmer to address races that cannot automatically be tolerated. The ToleRace modifications have three components:

- **Prolog** When the critical section is entered, ToleRace makes two additional copies of the shared data that the lock protects. A boolean flag tracks whether the shared data is updated in the critical section.

- **Body** The body of the critical section is modified so that updates to the shared variable are instead made to a local copy. The replacement assures that the updates are atomic with respect to actions taken by other threads.
- **Epilog** When the critical section is exited, ToleRace uses the values of the copies and the shared variable to determine if a race may have occurred. Depending on the values, ToleRace may tolerate the race, allow the programmer to compensate for the race, or report the race as a runtime error.

Figure 2 shows how the code in the critical section from Figure 1 is modified by ToleRace. In this example, because the test for NULL takes place on the local copy, `l_gScript`, Thread 2 setting the value to NULL has no effect, and the race is tolerated. If Thread 2 executes simultaneously with the critical section in Thread 1, its update to `gScript` is detected, and because there are no local changes, the value is left as it is. The effect is that the threads ran with the order T1 then T2. Note that because the critical section does not update `gScript`, `gScript_dirty` is not used in this example.

A second scenario, where Thread 1 only writes the value of the shared variable, and Thread 2 only reads it (**WrW** race), is also completely tolerated by ToleRace. In that case, the epilog notes that the local copy is dirty (because `gScript_dirty` is set to true at the first update in the critical section), determines that the value of the shared variable is the same as it was originally, and copies the local copy back to the shared variable. Note that the unlikely case where Thread 2 modifies the shared variable and then sets it back to its original value remains and undetected race by ToleRace, but is surprisingly tolerated. This occurs because ToleRace will assume Thread 2 did not modify the variable and allow Thread 1's changes to remain, resulting in an execution sequence T2 then T1. This is a correct ordering because the only impact T2 can have on T1 is through the shared variable, and T1 reads the right value for that variable.

```
EnterCriticalSection();

// Prolog - create copies
gScript_dirty = false;  // track writes
o_gScript = gScript;    // original copy
l_gScript = gScript;    // local copy

// Body - replace references to shared variables
if (l_gScript == NULL) { // operate on local copy
   baseScript = default;
} else {
   baseScript = l_gScript;
}

// Epilog - resolve outcome
if (gScript_dirty) {
   // local copy has been updated
   // check if global copy updated
   if (gScript == o_gScript) {
      // possible there was no global update
      // local copy is most up-to-date
      // Thread sequence is T2 then T1
      gScript = l_gScript;
   } else {
      // both copies modified (Ww conflict)
      // Automatic toleration may not be possible
      DetectRace();
   }
} else {
   // no local changes
   // global copy is correct
   // thread sequence is T1 then T2
}

ExitCriticaSection();
```

Figure 2: **RwR** Conflict with ToleRace

The more complex scenarios arise when both threads modify the shared variable. ToleRace will detect these occurrences but cannot automatically determine an appropriate resolution because the operations may have interleaved in such a way that serializing T1 and T2 is not possible. In these cases (indicated by the call to `DetectRace` in the figure), ToleRace can be used in several different modes.

Testing For testing, ToleRace will detect and report only races that will actually occur in practice, resulting in few false positives. ToleRace has much lower overhead than existing race detection tools (e.g., Eraser [7]) so it is more likely to observe races in a test environment.

Deployment For deployed applications, we believe the overhead of ToleRace is sufficiently low that it can be left on all the time in many applications. If ToleRace is on all the time, many subtle races that occur but do not cause an application to immediately fail will be detected and reported. We note that ToleRace does not provide as much detail about the cause of a race as lock-set based race detection tools [7, 8]. Specifically, since we only modify the critical section, ToleRace cannot directly pinpoint the errant code causing the race.

Patching ToleRace is also valuable in fixing observed problems after an application has shipped. ToleRace can be selectively applied to only those critical sections that a developer suspects are likely to be involved in races, after they have been observed. In addition, while ToleRace cannot automatically tolerate **Ww** conflicts without help, a developer might understand the application sharing characteristics well enough to write semantically sound application-specific race resolution code that executes when a race is detected.

3 Discussion

We have implemented ToleRace as described above and measured its effect on small benchmarks in which we introduce known races. The overhead of ToleRace depends on how much data is protected by a critical section, and how often the critical section is entered. Even in programs in which 100% of the time is spent in critical sections, the overhead is small, typically less than 10%. Our initial measurements suggest that even in a larger synchronization-heavy application (e.g., a parallel hash-table library), the execution time overhead of ToleRace is still below 10%.

ToleRace currently introduces a memory overhead from making additional copies of shared data, so the space overhead is proportional to the fraction of program data that is shared between threads and the fraction of time spent in critical sections, depending on how we allocate the copies. In our benchmarks, ToleRace is able to completely hide some races, and reduce the likelihood of races in other cases by two orders of magnitude. We are in the process of applying ToleRace to larger programs to observe its overhead and effect on fault tolerance in those programs.

To summarize, ToleRace provides a new and effective approach at tolerating and detecting races in multi-threaded programs. We have described a low-overhead software implementation of ToleRace and discussed how it can help developers address race-related errors throughout the development cycle. While space prevents us from discussing many subtle issues in the design an implementation of ToleRace, in future work we will discuss these issues in greater detail.

References

[1] Emery Berger and Benjamin Zorn, DieHard: Probabilistic memory safety for unsafe languages, PLDI'06, pp. 158–168, June 2006.

[2] Shan Lu, Joe Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. ASPLOS'06, October 2006.

[3] M. Naik, A. Aiken, and J. Whalley, Effective Static Race Detection for Java, PLDI'06, pp. 308–319, June 2006.

[4] Karthik Pattabiraman, Vinod Grover, and Benjamin Zorn. Samurai - Protecting Critical Heap Data in Unsafe Languages. Microsoft Research Tech Report #2006-127, Microsoft Corporation, Redmond, WA. August 2006.

[5] Karthik Pattabiraman, Vinod Grover, and Benjamin Zorn. Software Critical Memory - All Memory is Not Create Equal. Microsoft Research Tech Report #2006-128, Microsoft Corporation, Redmond, WA. August 2006.

[6] P. Pratikakis, J. S. Foster, M. Hicks, LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection, PLDI'06, pp. 320–331, June 2006.

[7] Stefan Savage and Michael Burrows and Greg Nelson and Patrick Sobalvarro and Thomas Anderson, Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst., 15(4), 1997, pp. 391–411.

[8] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. ACM SIGOPS Oper. Syst. Rev., 39(5), 2005.