EFFICIENT RUNTIME DETECTION AND TOLERATION OF ASYMMETRIC RACES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Paruj Ratanaworabhan

February 2010

EFFICIENT RUNTIME DETECTION AND TOLERATION OF ASYMMETRIC RACES

Paruj Ratanaworabhan, Ph. D.

Cornell University 2010

This work introduces ToleRace, a runtime system that allows parallel 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. This dissertation starts by characterizing all possible interleavings that can cause races and precisely describes the effect of ToleRace in each case. Then, it presents the theoretical aspects of an oracle that knows exactly what type of interleaving has occurred. After that, it presents software and hardware implementations of ToleRace and evaluates them on multithreaded applications from the SPLASH2 and PARSEC benchmark suites.

BIOGRAPHICAL SKETCH

Paruj Ratanaworabhan received an M.Eng. degree in Electrical and Computer Engineering from Cornell in 2002. After spending a year at Georgia Institute of Technology, he joined the Cornell's Computer Systems Laboratory (CSL) as a Ph.D. student in the fall of 2003. He has since been advised by Martin Burtscher and is currently a visiting student at Institute for Computational Engineering and Sciences, the University of Texas at Austin. His research focus has been on race detection and toleration systems, phase-aware architectures, floating-point data compression, and value-based compiler optimization.

Before coming to the US for graduate study, he was an undergraduate student at Kasetsart University in Thailand and an instrument engineer with Black & Veatch (BV). While at BV, he was part of a team that designed and commissioned control and instrumentation systems in co-generation and coal-fired power plants.

To my late grandmother and father, Tome Chantawanich and Sawart Ratanaworabhan.

ACKNOWLEDGMENTS

I am deeply grateful to my advisor Martin Burtscher. His keen guidance throughout the course of my Ph.D. study has made me a contributor to computer systems research. He is a hand-on teacher and researcher who has always been there for me. He stresses simplicity and conciseness above all other qualities. After all, great works in science and engineering are based on these essences. To instill these two qualities to a person is not as simple and concise as the word's meanings suggest. I would have never grasped them myself had it not been for a dedicated teacher like Martin.

My sincere thanks also go to all my committee members, David Albonesi, Edward Suh, and Rajit Manohar. Dave was a teacher who taught me to be aware of other important issues in computer architecture beside performance. His course titled Contemporary Issues in Computer Architecture explored, among other things, power/performance trade-off and thermal management. No other persons could have taught it better than a veteran computer architect like Dave. Ed was my "security" teacher. He led an invigorating seminar in the spring of 2007 on computer security research. I could have never realized then that I would later become involved in a security research project and that Ed's seminar would lay a good foundation for me to succeed in that project. Rajit is undisputedly a teacher beloved by every student in the Computer Systems Laboratory. He is a prolific researcher and is conversant in virtually any subjects.

I am especially thankful to Edwin Kan who agreed to serve as a proxy examiner for my defense. I took Edwin's revolutionary course on Micro-Electro Mechanical Systems, which he taught with great passion. I was pleasantly surprised to learn that he was once a computer architect himself but chose to walk the solid-state physics path later on.

While I was a visiting student at The University of Texas at Austin, I was

associated with a research group led by the great man Keshav Pingali. He was so generous to me that I was allowed to work on projects not within the group's interest. His wit and genius have led many people to knight him "the God of Application Programs". I thank him greatly for his generosity.

Darko Kirovski and Benjamin Zorn of Microsoft Research were the ones who had the original vision of ToleRace. I would later explore it in great length and it eventually becomes my dissertation topic. It was at Microsoft Research (MSR) in Redmond, WA where I met and worked with the two researchers. Darko was an outgoing researcher whose skills range from RF circuit design to parallel programming. I thank Darko for his kind and warm reception. If there is someone beside Martin to whom I am most indebted, that person is undoubtedly Benjamin Zorn. I learn a great deal from Ben. He was my mentor and manager while I was at MSR. Since Ben was Martin's advisor, he is my academic grand advisor. I consider myself one of the luckiest students to have benefited from two generations of advising rooted in integrity and intellect.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

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

Finding a suitable model that addresses the programmability of parallel programs while keeping up with the performance expected of multi-core hardware is an active research area. Promising candidates include transactional memory [24] and the Galois system [30]. However, lock-based parallel programs are still dominant, particularly those written in unsafe languages such as C or C++ with add-on libraries for threading and synchronization. There is also ongoing research [11] that aims to improve the rigor of this programming paradigm.

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

weakly ordered in reality [2]. This work focuses on asymmetric races, which occur when one thread correctly protects a shared variable using a lock while another thread accesses the same variable improperly due to a synchronization error (e.g., not taking a lock, taking the wrong lock, taking a lock late, etc.).

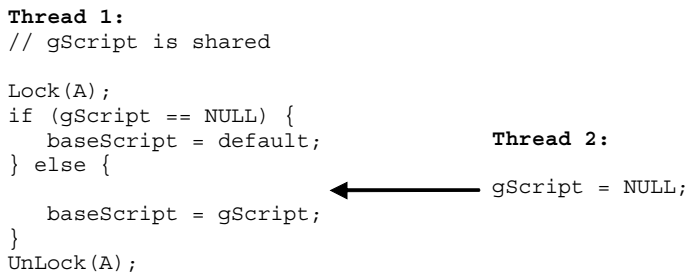


Figure 1: An asymmetric race.

An example of an asymmetric race is shown in Figure 1. Here, Thread 1 correctly uses a critical section to protect its read accesses to the shared variable `gScript`. Thread 2 incorrectly updates `gScript` without a lock, thus creating a race. The race occurs infrequently, i.e., only when Thread 2's update happens between the test for NULL and the `else` part of the conditional in Thread 1.

1.1 Why Asymmetric Races?

This work focuses on asymmetric races for three reasons:

1. They are common in software development projects.

This conclusion comes from direct experience with developers in software houses like Microsoft. There are two reasons for this. First, usually a programmer's local reasoning about concurrency, e.g., taking proper locks to protect shared variables, is correct. Errors due to taking wrong locks or no locks lie outside of the programmer's

code, for example, in third party libraries. Given that lock-based programs rely on convention, this phenomenon is understandable. The second reason has to do with legacy code. As software evolves, assumptions about a piece of code may be invalidated. For instance, a library may have been written assuming a single-threaded environment, but later the requirements change and multiple threads use it. An expedient response to this change is to demand that all clients wrap their calls to the library, acquiring locks before entry and releasing them on exit. Because this solution requires that all clients be changed, races can be introduced when clients fail to observe the proper locking discipline.

```
// K and flag are declared volatile

Thread 1:                Thread 2:

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

Figure 2: User-defined synchronization.

2. Symmetric races are usually benign.

Because calls to synchronization operations are expensive, programmers often resort to lightweight user-defined synchronization as shown in Figure 2, where Threads 1 and 2 synchronize on the `flag` variable. In this situation, even though a race occurs by definition (the shared variable `flag` is written and read without explicit synchronization), it does not harm the program. Narayanasamy et al. [40] show other types of benign symmetric races, e.g., redundant writes and disjoint bit manipulation. Their experience with Windows Vista and Internet Explorer indicates that these benign races are rather common.

3. Programmers want to reason locally about the correctness of the critical sections in their code.

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

1.2 Introducing ToleRace

ToleRace is a runtime system that allows programs to continue executing in the presence of asymmetric races and possibly complete with a well-defined semantics. Inspired by the DieHard system [8], which probabilistically tolerates memory safety errors, ToleRace uses replication to detect and/or tolerate races. It provides an approximation of isolation in critical sections by creating local copies of shared variables when a critical section is entered, operating on the local copy while in the critical section, detecting conflicting changes to shared data when the critical section is exited, and propagating the appropriate copy when possible to hide the race. ToleRace allows a variety of implementations that range from software only, where races are only probabilistically detected and tolerated, to a combination of hardware and software, where stronger guarantees are possible. This work focuses on the fundamental properties of ToleRace, describes and evaluates possible software and

hardware implementations.

ToleRace aims to be a dynamic race detection and toleration system that is always enabled whenever a program runs. Its uses are not limited to test runs only, but are meant to include production runs. Management of races during production runs, I believe, is the right approach to adopt. It complements a comprehensive testing phase and is based on the realization that testing for concurrency errors such as races can never be completely thorough [37]. Having the capability to check and even protect against races in the already deployed software is, thus, desirable as the inherently error-prone software is always monitored by a runtime system. This gives added confidence to both the vendors and the users of the software.

To be successful as a runtime system that operates in a production environment, it needs to be correct, efficient, and has near zero false positives. In addition, it has to be transparent. It must work directly with out-of-the-box executables and not require the availability of the program's source code. Enabling transparency is of prime importance to software vendors as they want to protect their intellectual property contained in the source code. As we shall see later in this dissertation, the ToleRace implementations proposed here have all these four qualities just mentioned.

ToleRace can be compared to transactional memory (TM) [24]. 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 [10].

1.3 Contributions

This work makes the following contributions:

- **Foundations for runtime management of races.** The next chapter presents a theoretical framework that investigates all possible interactions among safe threads that observe the proper locking discipline and unsafe threads that fail to do so. Then, it focuses on cases where a race occurs, categorizes those cases, and, in each category, describes race detection and toleration strategies.
- **Enabling race tolerance.** ToleRace enables programs to tolerate races, which decreases the likelihood that the races will cause incorrect program behavior. Increasing a program's tolerance to races reduces the need for the races to be debugged/patched. In instances where ToleRace cannot tolerate races, it detects them either precisely or with high probability, depending on the implementation. Note that there are instances where ToleRace silently and correctly tolerates races without detecting them (cf. Section 3.1).
- **Precise detection.** ToleRace identifies races that actually happen at runtime. It detects a race when the critical section in which the race took place exits and, by design, never generates a false positive.
- **Programmer-centric local reasoning.** ToleRace enables programmer tools to allow local reasoning about correctness and to facilitate a structured means of detecting and tolerating errors that are caused by code outside the programmer's control. The programmer can control the overhead by selectively turning ToleRace on or off for individual critical sections. This is useful for debugging, testing, and patching released executables.

• **Low overhead software implementation.** This dissertation presents three software implementations of ToleRace. The first version uses a dynamic instrumentation-based approach and performs all analysis at runtime. The second version adds a static program analysis phase to remedy the shortfalls of the first version. Both versions work directly on the program's executable and they require no presence of the program's source code. The third version, however, is radically different from the first two. It is based on source-code modifications to implement ToleRace.

• **Hardware implementation that leverages multicore components.** This work also investigates how the ToleRace mechanism can be realized in the now ubiquitous multicores. The proposed design extends the private cache of each core and requires minor modifications to the cache coherence protocol. In keeping with the spirit of ToleRace, the proposed hardware is transparent to the running executable, i.e., standard executables can run on top of it without any modification to the binaries.

CHAPTER 2

CHARACTERIZING ASYMMETRIC RACES

This chapter introduces the notations that will be used throughout this dissertation and characterizes asymmetric races.

2.1 Notations

Let *l*() and *u*() denote the atomic functions that acquire and release a specific lock, respectively. *r*() and *w*() represent two functions that read from and write to a specific variable. This chapter first considers cases when a single variable is protected and accessed in a non-nested critical section. The next chapter extends this theoretical framework to cover cases involving multiple variables and overlapped critical sections. **l**, **u**, **r**, and **w** denote the fundamental functions over that specific variable, and **x** denotes a "*don't care*" function that can be either a read or a write. **r**+ denotes a sequence of at least one read and **r*** indicates zero or more reads. The operators + and * are equally defined for writes. For a specific thread T_1, the sequence of critical operations using the above operators and fundamental functions are defined. For example, $T_1 = [\mathbf{l}_1\mathbf{r}_{11}\mathbf{w}_{11}\mathbf{r}+_{12}\mathbf{u}_1]$ denotes a thread that first locks a variable, then reads and writes exactly once, followed by at least one more read before it unlocks the variable. The first digit in the operation index denotes the thread index and the second digit, where present, distinguishes between sequences of operations of the same type. The following example shows one possible interleaved execution of critical operations of two threads $T_1 = [\mathbf{l}_1\mathbf{r}_{11}\mathbf{w}_{11}\mathbf{r}+_{12}\mathbf{u}_1]$, $T_2 = [\mathbf{w}_2]$: $S = \{\mathbf{l}_1\mathbf{r}_{11}\mathbf{w}_{11}\mathbf{w}_2\mathbf{r}+_{12}\mathbf{u}_1\}$. The sequence *S* specifies that the write from the second thread occurred between the first write and the second read of the first thread and thus causes a race condition.

2.2 Operation Interleavings and Race Cases

To characterize asymmetric races, this section investigates all possible interleavings between the operations of a correctly synchronized thread and a second, unsynchronized thread. The interleavings that result in races are grouped into four classes and the way ToleRace handles each class is explained. This study assumes a programming model with two types of threads :

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

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

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

This definition is agnostic to the threads' execution order. It does not presume any execution order among the threads, as long as they execute atomically with respect to their critical sections.

A thread (safe or non-safe) in T could execute but not affect the program's computation state. In this case, *Definition 1* can be relaxed to accept execution schedules as correct where a subset of the threads does not execute.

With the execution model defined above, ToleRace preserves the data-race-free-0 model semantics [2] when it tolerates all the occurring races. Such a model guarantees sequentially consistent execution [31] for all synchronization operations while allowing the underlying hardware to be weakly ordered.

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

operation interleaving				operation interleaving				operation interleaving			
T'_1	T_2	T''_1	race	T'_1	T_2	T''_1	race	T'_1	T_2	T''_1	race
R+	r+	R+	false	R+	wx*	R+	true	R+	r+wx*	R+	true
R+	r+	WX*	false	R+	wx*	WX*	true	R+	r+wx*	WX*	true
R+	r+	R+WX*	false	R+	wx*	R+WX*	true	R+	r+wx*	R+WX*	true
WX*	r+	R+	false	WX*	wx*	R+	true	WX*	r+wx*	R+	true
WX*	r+	WX*	true	WX*	wx*	WX*	false	WX*	r+wx*	WX*	true
WX*	r+	R+WX*	true	WX*	wx*	R+WX*	true	WX*	r+wx*	R+WX*	true
R+WX*	r+	R+	false	R+WX*	wx*	R+	true	R+WX*	r+wx*	R+	true
R+WX*	r+	WX*	true	R+WX*	wx*	WX*	true	R+WX*	r+wx*	WX*	true
R+WX*	r+	R+WX*	true	R+WX*	wx*	R+WX*	true	R+WX*	r+wx*	R+WX*	true

To understand how the safe and non-safe threads can interact, one needs to explore all interleavings where the non-safe thread T_2 executes between operations in the safe thread T_1. Note that there are only three ways in which a sequence of operations by a single thread can interact with a single variable: by reading it only (**r+**), by setting its value regardless of its prior (**wx***), and by setting its value based upon its prior (**r+wx***). Operations that follow a write by a particular thread are important semantically but do not affect the inter-thread interactions. Also note that **rw** could occur in two versions: (i) **w** is dependent upon the value retrieved by **r** and (ii) **w** is not dependent upon the value retrieved by **r**. Sequences where (ii) is true could be analyzed as independent manifestations of two sequences of type **r+** and **wx***.

Sequences where (i) is true demand special attention; thus, in the remainder of this work, when a sequence **rw** is issued by the same thread, case (i) is assumed.

Table 1 tabulates all possible interactions between a safe thread T_1 and a non-safe thread T_2. The safe thread is improperly intercepted by T_2 at a position that slices the operations of T_1 into two parts T'_1 and T''_1. The table evaluates the outcome of this interaction exhaustively. The following classification theorem results from Table 1.

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

I. XwR = $\{\mathbf{l}_1\mathbf{x}+_1\mathbf{w}_2\mathbf{x}*_2\mathbf{r}_1\mathbf{u}_1\}$. *This case specifies that any sequence of operations by* T_2 *that starts with a write and occurs after any operation but before a read in* T_1 *causes a race.*

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

III. RwW = $\{\mathbf{l}_1\mathbf{r}_1\mathbf{x}*_1\mathbf{w}_2\mathbf{x}*_2\mathbf{w}_1\mathbf{u}_1\}$. *When* T_1 *starts with a read followed by an arbitrary sequence of operations, and* T_2 *executes any sequence of operations that starts with a write just before* T_1 *writes back to this variable, a race will occur.*

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

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

Proof. Straightforward by combining cases from Table 1. •

There is previous work [33, 48] that also proposes enumeration of possible interleavings. However, it does not focus on race toleration as this work does. Section 3.1 describes how the classification from Table 1 is useful for this purpose.

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

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

Now, consider multiple safe threads among the K interacting threads. Because safe threads, by definition, hold consistent locks 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. •

CHAPTER 3

THE TOLERACE ORACLE

This chapter presents a theoretical framework, the ToleRace Oracle, and describes how it handles all the race cases specified in Theorem 1 in the previous chapter.

The core approach to managing the race condition cases specified in Theorem 1 is to replicate the protected shared state so that the thread that acquires a lock on the shared state has an exclusive copy (see Figure 3). This thread continues reading from or writing to this copy until it releases the lock. When the lock is released, the ToleRace runtime can employ a variety of software 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.

This chapter studies the effect of ToleRace on the cases described in Theorem 1, assuming an oracle determines which race has occurred.

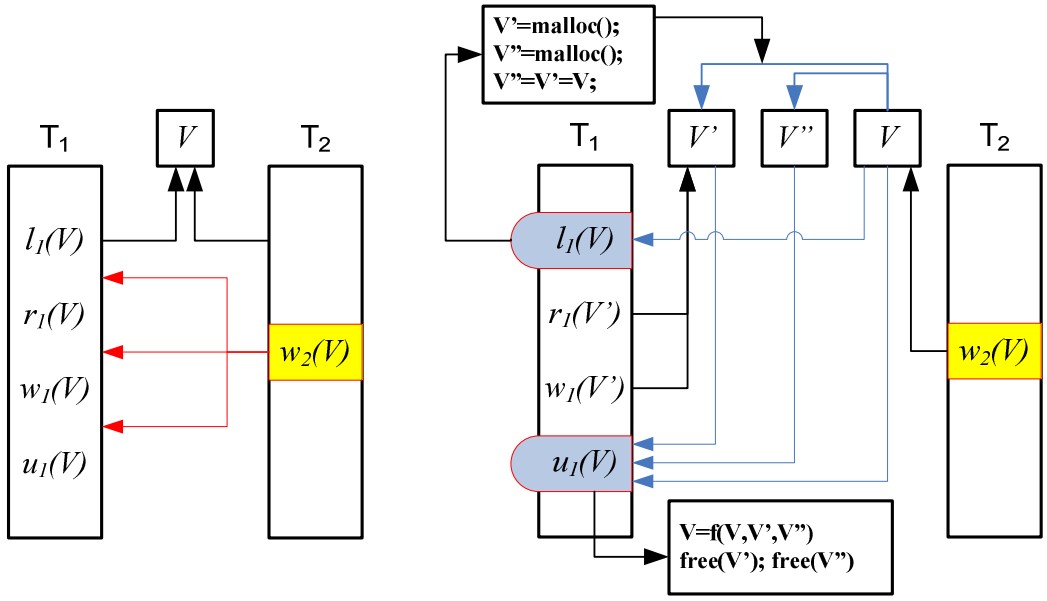
Initialization and Finalization: Assume that the binding of locks (x_V) to shared variables (V) is known before the critical section in T_1 is entered and that storage for two additional copies (V', V'') of variable V has been allocated. After the lock is released, the storage for the two copies is deallocated.

Lock (Entry): When lock x_V is acquired by T_1, copying V to V' and V'' ($V''=V'=V$) is performed atomically. Note that the copying and the lock acquire together may not be performed 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, not even 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.



(a) Without ToleRace (b) With ToleRace

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

3.1 Tolerating and Detecting Races with the oracle ToleRace

Combining the mechanism outlined above with the exhaustive interleavings

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

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

race type		V = V''	*f*(V, V', V'')	tolerable	
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	custom *f*'	maybe	N/A

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

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

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

IV_C: RrwW=XrwX – {RrwR ∪ WrwX}

The first column in Table 2 lists the race type based upon the classification from Theorem 1, the second column specifies whether *V* is equal to *V*'' at the point when *f* is called, the third column shows a resolution function *f* that allows ToleRace to tolerate the race, the fourth column indicates whether *f* provably succeeds in tolerating the race, and the fifth column presents , the schedule of threads that ToleRace's result represents. Table 2 shows that the ToleRace oracle tolerates all races except sequences of the form RrwW with the resolution function *f* defined by Table 2. Figure 4 gives an example that shows how ToleRace tolerates an RwR race, a sub-type of Type I race.

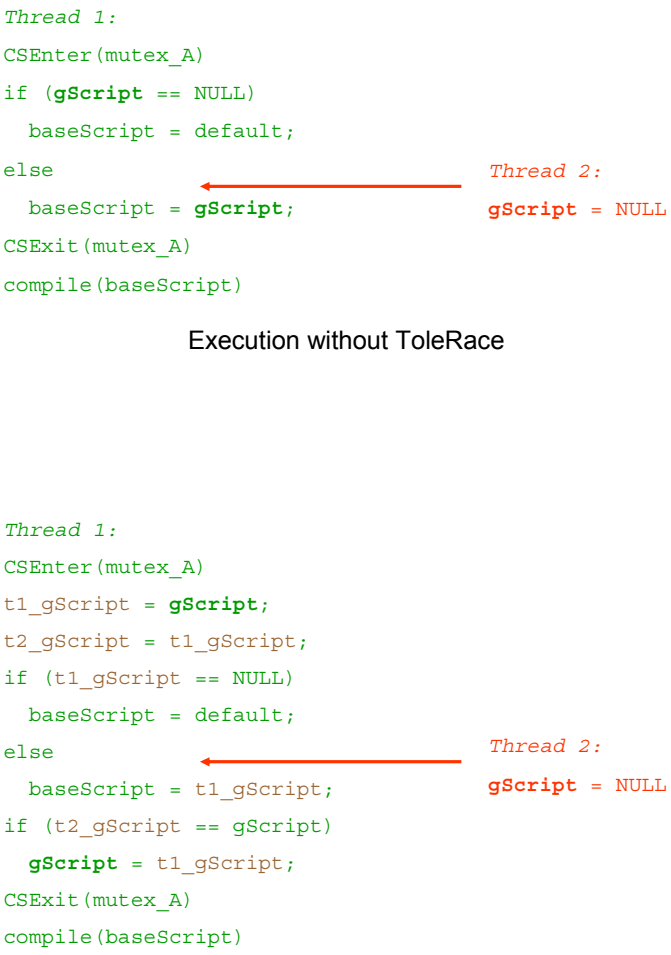


Figure 4: Tolerating RwR race with ToleRace

For races of type RrwW, the interleaving of reads and writes from T_2 breaks the program's sequential memory consistency. Here, T_1 and the interleaved part of T_2 both read the value of the shared variable once T_1 has entered the critical section, execute in parallel, and then join at the exit of the critical section of T_1. T_1 and T_2 see the same value returned by the read, which would not be possible if T_1 had executed its critical

section in isolation.

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

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

b) there is a WrW race. While ToleRace cannot detect this race, it can tolerate it.

3.2 Multiple Variables and Nested Critical Sections

So far, we have considered the oracle ToleRace in a multithreaded, single-variable, and non-nested critical section context. We now extend this framework to handle general cases, which involve multiple variables and nested critical sections. Local copies and the resolution function need to be made and executed atomically for multiple variables. Nested critical sections share their local copies with the outermost critical section. However, they have their own resolution function to resolve races for their protected variables. When dealing with these general cases, the race toleration mechanism employed in ToleRace may lead to inconsistent execution. In such circumstances, ToleRace cannot permit the reordering of the shared variable and acts as a race detector instead.

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

Inconsistent execution can potentially occur with ToleRace because it resolves races to each variable independently. Here I outline the proof of Theorem 3 and give an illustrative example. 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.

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 third column indicates whether ToleRace reorders the intervening operations to P and Q. This follows directly from the resolution function in Table 2 as just described. The fourth column lists if there is a dependency from P to Q. In general, when there is a write to Q and the accesses to P may contain a read, then Q may be dependent on P, and, hence, the operations must observe program order. The fifth column shows the ToleRace action for each combination, which can be deduced directly from the result in columns 3 and 4. ToleRace reverts to detection mode when it determines that there may be a dependency among variables and the resolution function allowed out-of-order execution.

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

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

Example 1. Let P and Q be shared variables with P = 1 and Q = 2. X is a lock variable and t11, t12, and t21 are all local variables. Consider a race that slices the operations in this critical section below into two parts:

```
Lock(X)
t11 = P;
Q = 5;
[Racing operations from a different thread]
t12 = Q;
P = 6;
Unlock(X)
```

Let the following be the racing operations coming from another thread:

```
P = 3;
t21 = P + 4;
Q = Q + t21;
```

From these three statements that form the racing operations, we observe that there is a dependency from P to Q where the local variable t21 is an intermediary. If there is no race, the possible final values of P and Q are:

1. P = 3 and Q = 12 if the critical section executes before the racing operations
2. P = 6 and Q = 5 if the critical section executes after the racing operations

With the race, ToleRace resolves the value of P and Q coming from the racing operations to have come after and before the critical section, respectively. Therefore, the final values of P and Q are 3 and 5, respectively. The ToleRace resolution makes it appear that it has reordered the write to P to have occurred after the write to Q. This P and Q value combination cannot be allowed as it would violate data dependence between P and Q in the racing operations. •

In general, we do not expect this extended framework to be invoked often. A recent study by Lu et al. [32] points out that the common cases for concurrency bugs, including races, involve only a single variable and no more than two threads.

Note that Theorem 3 does not disallow orderings that violate sequential memory consistency (SC). If a programmer wants SC to be honored on a weakly-ordered hardware, he or she has to place explicit synchronizations such as memory fence operations to restrict orderings to only those that conform to SC.

The oracle ToleRace described represents a theoretical framework that cannot be fully realized in practice. The next three chapters describe software implementations that approximate it. A hardware implementation is discussed in Chapter 7. Although the framework permits both software and hardware implementations, a software approach may be more appealing as it can be deployed immediately. Chapter 4 describes an initial implementation that is restricted and sub-optimal. It serves as a baseline for other implementations to benchmark against. Chapter 5 presents an improved version that addresses the shortfalls in the initial version. It approximates what would likely be deployed in practice. Chapter 6 investigates an idealized version of software ToleRace. It assumes an oracle compiler and the availability of the program's source code.

CHAPTER 4

SOFTWARE TOLERACE: A FIRST VERSION

This chapter discusses the initial version of software ToleRace [46] that is non-optimal and possesses some inherent restrictions. This first version makes all decisions at runtime and does not perform any static program analysis. It allows us to gauge an upper bound on the software ToleRace overhead. In the next chapter, I 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.

I implement ToleRace on top of Pin [35] running on x86 Linux systems. The parallel applications are written in C/C++ and use the pthreads library for synchronization operations. However, I believe the framework described here generalizes to other platforms and threading libraries. Unless noted otherwise, I apply software ToleRace to critical sections in the user code whereas critical sections in the library code receive no ToleRace protection. I 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 runtime via the procedure linkage table in the .plt section. The software versions in this and the next chapters work directly on standard Linux/X86 executables They also do not require information and analysis at the source code level; all the required information is obtained from the executable itself or during runtime.

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, this 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. I conservatively assume that all memory accesses in a critical section touch shared memory locations except for those touching the thread local stack. I 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 added code is generally referred to as 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 I 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, I 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. I will revisit this restriction in Chapter 5 when I 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.

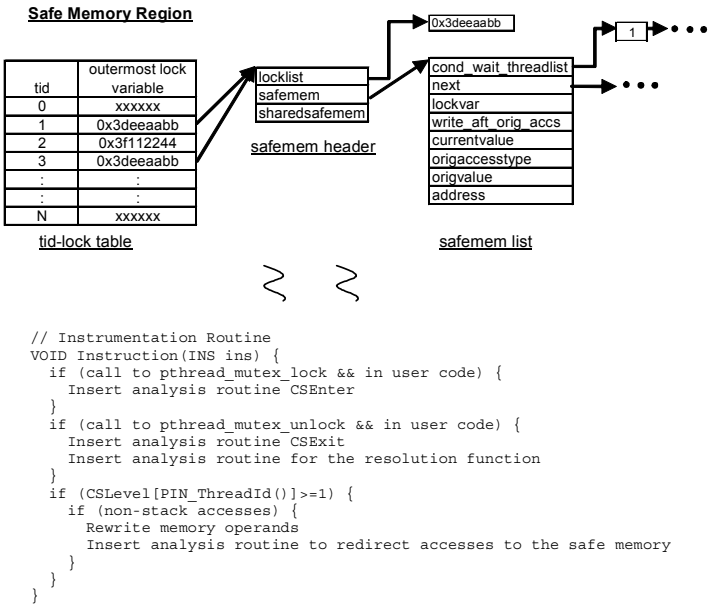


Figure 5: Pin-ToleRace framework.

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

As mentioned, the safe memory region is where the local copies of the shared memory locations reside. It contains three main data structures: the thread id (tid) lock mapping

table, the safemem header, and the list of safemem nodes. The lock mapping table size is determined by the maximum number of threads allowed in the system. The other two are dynamic structures, and their content is alive as long as the execution proceeds through the critical section. The content is created by the first instruction that accesses a shared memory location. The role of each of these structures is explained next.

4.2.2 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), I 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.

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 might reside in another safe memory region.

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

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. A 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. However, we must distinguish between a call to a user-defined and a call to a library routine. We only want to protect user code, and, therefore, do not want to redirect shared memory accesses in library code. Nevertheless, we cannot simply exclude accesses to the safe memory from libraries because a call to a library routine can pass pointers to shared variables as arguments. To handle this case, we allow the library code to access the existing nodes in the safemem list but not to create new nodes.

4.2.7 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.

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 resolving a race to multiple shared memory locations, one undesirable effect from Pin-ToleRace is that it may still violate sequential consistency even if all the necessary synchronization operations that disallow non-SC orderings are put in place.

To understand this subtle point, let us look at the following concrete example.

Example 2: Let the original program for thread 1 and thread 2 be as follows and let the values of P and Q before both threads execute these codes be zero.

```
Thread 1:                    Thread 2:
P = 1;                       Lock(mutex_X);
Fence();                     temp1 = P;
Q = 2;                       temp2 = Q;
                             Unlock(mutex_X);
```

The following is an ordering allowed by Pin-ToleRace.

```
Lock(mutex_X);
temp1 = P;
P = 1;
Fence();
Q = 2;
temp2 = Q;
Unlock(mutex_X)
```

When Pin-ToleRace resolves the race, the values of P and Q after thread 2 exits from the critical section are 1 and 2, respectively. The resolution function passes on the values of P and Q written by the racing thread (thread 1) as the only operations inside the critical section of the safe thread (thread 2) are read operations. However, the value for temp1, which reads from P, is zero and the value for temp2, which read from Q, is 2. Hence, from the standpoint of thread 2, the write to P appears to have

come after the write to Q even though the explicit fence synchronization disallows this. Note that the oracle ToleRace does not suffer from this problem. The oracle would have known to protect the shared variables P and Q when entering the critical section and would have proceeded to atomically copy the values of P and Q at that point. Pin-ToleRace does not have such oracle knowledge and determines all the shared location on-the-fly, so it can suffer from the above described scenario.

4.4 Evaluation

4.4.1 Benchmarks

I use 13 applications from the SPLASH2 [49] and PARSEC [9] benchmark suites to evaluate Pin-ToleRace. I 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. Four of the programs, cholesky, fft, lu, and radix, are kernels whereas the other four, barnes, ocean, radiosity, and water, are full applications. I replaced the SPLASH2 suite's PARMAC macros with a pthreads library implementation. For each of the eight program, the default inputs were used. I, however, increased some of the input sizes to lengthen the program runtimes.

I selected the five programs from the newly released PARSEC suite that use the pthreads library. One of these programs, dedup, is a kernel and the other four, facesim, ferret, fluidanimate, and x264, are real applications. The PARSEC suite aims to provide up-to-date multithreaded programs that focus on emerging workloads in recognition, mining, and synthesis. 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. I 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, 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, I 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. My test setup uses five non-safe threads and runs the three programs with 5M, 7.5M, and 10M iterations. In each experiment, we observe the correct values in all shared locations just before the critical section exit. We also see that after exiting from the critical section, the values of these shared locations change to the thread id of the non-safe thread that ran last.

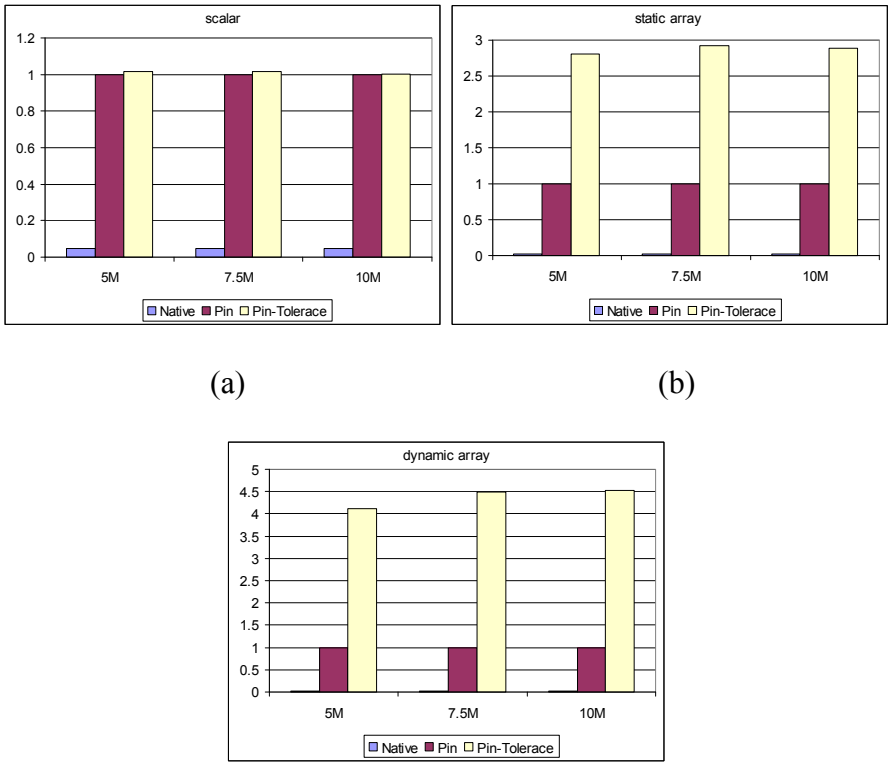


Figure 6: Normalized execution time of Pin-ToleRace for scalar (a), static array (b) and dynamic array (c) for different iteration counts.

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

while the Pin-ToleRace runs have all their races correctly tolerated.

For all three microbenchmarks, the overhead of Pin-ToleRace over native is very high—up to 80 times in the dynamic array case. The primary reason 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 are always busy tolerating races inside a critical section. The next section shows that real applications have critical section characteristics that are more benign and thus incur a much lower 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, I compiled the 13 benchmarks to use four processors, which corresponds to the number of cores on the evaluation platform. I then used Pin to collect the critical section statistics shown in Table 4. Note that I only study critical sections that reside in the user code, i.e., I 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%

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

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

critical sections, notably fluidanimate and radiosity, whose small critical sections are being looped over.

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

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

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

Pin-ToleRace Performance: This section studies the overhead of Pin-ToleRace on the benchmark applications. Given the results of the previous subsection, I 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. I chose this size to minimize the collisions in dedup and ferret.

Figure 6 presents the results. The timing measurements are normalized to the native runtime. Note that this is different from the normalization I used for the stress tests. The second bar shows the pure Pin overhead without instrumentation for each program. The third and fourth bars indicate the overhead of Pin-ToleRace with linked list and hash table implementations of the safe memory, respectively. On average, Pin-ToleRace incurs about a factor of two slowdown relative to the native runs and about 24% overhead relative to the Pin runs. I believe these performance degradations to be low enough to make Pin-ToleRace deployable in production environments. Moreover, by adding static analysis (see Chapter 5) or hardware support, it should be possible to reduce the overhead.

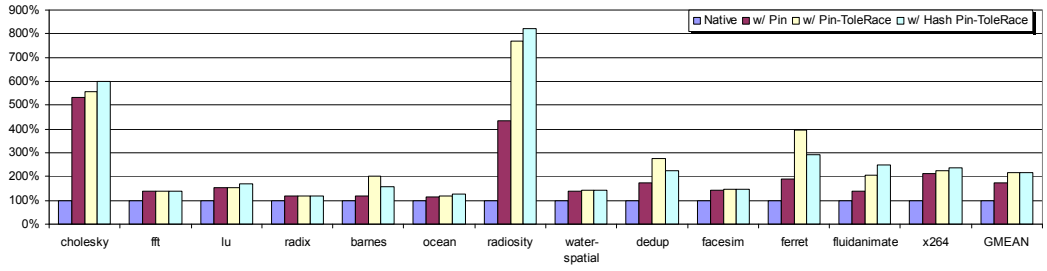


Figure 7: Normalized execution time of Pin-ToleRace.

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.

CHAPTER 5

IMPROVING THE INITIAL PIN-TOLERACE VERSION

In the previous chapter, I described a Pin-ToleRace system that performs all analyses at runtime. As noted, this system is used to gauge the upper bound on the overhead and is sub-optimal. To improve the efficiency of the system, we want to, among other things, tailor the safe memory structure to each critical section instead of treating every critical section uniformly. In addition, recall that the initial Pin-ToleRace made the assumption that all code segments that are sometimes reached from outside of any critical sections execute only after they have already been instrumented inside of a critical section. In this chapter, I clarify why this restriction is inherent in the initial Pin-ToleRace system.

There are, thus, two important objectives in this chapter: describing how to implement a more efficient Pin-ToleRace system and how to remove the above restriction. This chapter starts by discussing the sources of inefficiency in the previous system. It then explains the initial Pin-ToleRace restriction. After that, it describes how to achieve the two objectives by introducing a static program analysis phase that passes the necessary information to the Pin-ToleRace runtime.

5.1 Inefficiency in Pin-ToleRace

The sources of ineffectiveness 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 runtime, 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 operation 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: We have investigated two data structures for the safe memory implementation, namely, linked lists and chained hash tables. Recall that the linked-list structure is good for short critical sectionsr. This type of critical sections benefits from fast creation and destruction of the safe memory; the search operations are not critical as the number of accesses is small. On the other hand, the hash table structure is better suited for long critical sections. This type of critical section needs fast search operations; the cost of the creation and destruction of the safe memory can be amortized because of the high number of accesses. 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 8: An example illustrating how the assumption in the first version of Pin-ToleRace may be violated.

Every performance-oriented dynamic translator including Pin has a code cache whose primary goal is to speed up the translation by caching the already translated code segments and looking them up when they are encountered again. For Pin, the basic unit for the stored translated code is a trace, which is essentially a stream of instructions that ends with some control transfer instruction. Pin's code cache poses a complication. 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 in 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 directly accesses shared memory locations. In general, in the presence of a code cache, code segments that can potentially be executed both inside and outside of critical sections may cause incorrect runtime 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.

In order to tackle these restrictions, we need to identify up front what code may execute under critical sections. At runtime, we instrument such code whenever the code is discovered either inside or outside of the critical sections. The timing when the code gets instrumented is irrelevant as long as correct instrumentation routines are used. Recall from Section 4.2.5 that the initial Pin-ToleRace uses guards to disallow accesses to the safe memory when the code is executing outside of critical sections. To identify the code to instrument, we use a combination of static program analysis and runtime techniques. The former help us identify code segments such as statement 1 through 4 and function f() in Figure 7 whereas the latter allows us to cope with indirect call aliasing inside of critical sections.

As an aside note, Pin provides the instrumentation APIs INS_InsertIfCall and

INS_InsertThenCall that can be used in combination to selectively instrument instructions under a particular runtime execution condition. So, if we let the condition be that of executing in some critical section, we can solve the instrumentation problem mentioned above. However, these conditional instrumentations tend to incur higher overhead than the generic INS_InsertCall. Moreover, Pin does not have the capability to selectively perform memory operand rewriting, and, hence, our problems are not completely solved with these provided APIs.

One brute force way of tackling these problems all together is to flush the Pin's code cache every time a critical section is executed. The primary drawback here is that code cache flushing is a prohibitively expensive operation. Even though parallel programs generally spend only a small amount of their execution time inside critical sections, the cost of such flushing is still large. I experimented with the cholesky kernel from the SPLASH2 suite and found that the Pin-ToleRace overhead with full flushing is over 100X.

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 Pin-ToleRace. Figure 8 shows a block diagram of 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 traverses 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.

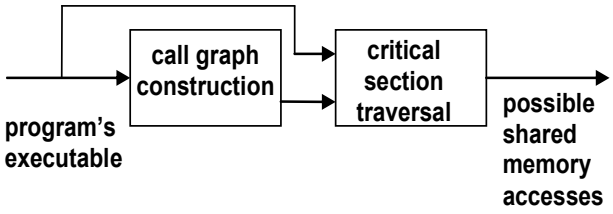


Figure 9: Static program analysis phase.

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 know 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 libraries in our case) are also part 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: The module uses a two-pass algorithm. During the first pass, it traverses the program's executable and collect target addresses and possibly names of

the user routines. It obtains this information from examining the .text section of the program. It eliminates certain routines that are not actually part of the program, but are included 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, it also gathers target addresses and possibly names of the shared libraries, including the pthreads libraries. This information is manifested in the procedure linkage table, which is contained in the .plt section of the executable. Note that it deals with x86/Linux platforms here; others may have different executable formats and conventions.

After the module has collected all the necessary information in the first pass, in the second pass, it traverses the .text section to build a call graph. It walks each routine in the section one by one. For a given routine, it traverses every instruction in the routine from start to finish in static program order and searches for calls to other routines. If a call is found, it checks its target and creates an edge from the current (calling) routine to the called routine. When examining each routine, it also gathers other information needed by the analysis in the next module (see output below). Note that it only deals with a call whose target is known at compile time. I discuss handling of indirect calls in Section 5.3.4.

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

The processing in this module uses Pin's API and its callback function IMG_AddInstrumentFunction when traversing the program and constructing the call graph. IMG_AddInstrumentFunction takes a pointer to a user-defined analysis function, our call graph construction module, as one of its argument. Note that I am

only concerned with analyzing the user generated image and, hence, do not analyze any dynamically loaded shared libraries.

Output: After processing, we have information about each routine in the .text section that 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 second module in the static analysis phase is called the critical section traversal module. 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 TraverseCS shown in Figure 9. This function gets invoked when a call to the pthread_mutex_lock routine is found while it traverses 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 1) the corresponding unlock is found or 2) a call that never returns back to the user mode such as exit() or assert() is encountered (if case 1). Otherwise, the recursive call falls into the other 8 cases and continues the recursion. Note that the Pin's INS_Next API in TraverseCS returns the next instruction after the current instruction in static program order.

```
VOID TraverseCS(INS instruct) {
  ins = INS_Next(instruct);
  Mark ins as visited;
----------------------------------------
  if (unlock || non-return calls) {
    return;                                  1
  }
----------------------------------------
  else if (lock) {
    nested CS recorded;
    TraverseCS(ins);                         2
  }
----------------------------------------
  else if (cond_wait or cond_broadcoast) {
    condition variables recorded;
    TraverseCS(ins);                         3
  }
----------------------------------------
  else if (other shared libraries) {
    library call recorded;
    TraverseCS(ins);                         4
  }
----------------------------------------
  else if (call to user routines) {
    user function call recorded;
    TraverseCS(ins);                         5
  }
----------------------------------------
  else if (indirect call) {
    indirect call recorded;
    TraverseCS(ins);                         6
  }
----------------------------------------
  else if (conditional branches) {
    ins_target = GetTarget(ins);
    TraverseCS(ins);
    if (ins_target not visited)              7
      TraverseCS(ins_target);
  }
----------------------------------------
  else if (unconditional branches) {
    ins_target = GetTarget(ins);
    if (ins_target not visited)              8
      TraverseCS(ins_target);
  }
----------------------------------------
  // instructions not affecting control flow
  else {
    Check if ins access non-stack memory;
    Hint to instrument if true;
    TraverseCS(ins);                         9
  }
}
```

Figure 10: The TraverseCS routine. This routine is invoked when a lock associated with a given critical section is found. It statically and recursively traverses the critical section to identify potential instruction candidates that may access shared memory locations.

Case 2 is entered when a (static) nested critical section is discovered. Care must be taken to communicate back to the TraverseCS calling site that this nested lock call need not invoke TraverseCS again. Its associated critical section traversal will have already been done as part of the traversing of the outer critical section, which was initiated by the outer lock call. Case 3 to 6 are responsible for call instructions that divert the control flow to other user or shared library routines. Case 3 tells if there is a call to synchronization libraries specifically pthread_cond_wait, pthread_cond_signal, and pthread_cond_broadcast. Case 4 deals with all the other shared libraries. Case 4 calls can be initiated from either inside or outside of critical sections whereas case 3 calls can only be invoked inside critical sections. Case 5 handles user routine calls. For case 4 and 5, TraverseCS also records what the called routines are. Case 6 indicates if there is an indirect call present.

Cases 7 and 8 handle conditional and unconditional branches, respectively. When TraverseCS encounters a conditional branch, it traverses 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, it needs to only traverse the target path if the target instruction has not already been visited. In both cases, whenever it encounters a branch target address that is less than the current branch address, i.e., a back edge, it checks if a loop is formed and whether there are instructions potentially accessing shared memory locations in the loop.

Case 9 is for all instructions that do not affect the control flow. When this case is entered, it checks if the instruction may access shared memory locations. If so, it adds it to the list of the candidate instructions. At runtime, the operands of each candidate instruction will be rewritten and the instruction itself instrumented with an analysis routine that diverts the accesses to the safe memory region.

For a critical section that contains calls to user routines, it also needs to include the

candidate instructions from the called routines. It first consults the call chain of each called routine. Then, it obtains 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.

Like in the call graph module, Pin's callback IMG_AddInstrumentFunction is employed in the processing.

Figure 10 shows an example when TraverseCS processes a critical section. It also contrasts manifestation of the critical section at the source code level and at the executable level. As we can see, an optimizing compiler has transformed what looks like a simple straight line code critical section where a lock call appears before an unlock call into a cryptic looking one. TraverseCS starts traversing the critical section at address 804a12c. It then reaches a conditional branch (case 7 in the TraverseCS routine). At this point, it first traverses the fall through path until it hits an assert call, which is a non-return call (case 1). Then, it traverses the target path until the corresponding unlock call is found at address 804a110. While traversing and meeting with the instructions that may access shared memory locations, highlighted in Figure 10, it collects their addresses, which will later be passed on to the runtime system. This critical section at the executable level is effectively defined by two ranges of addresses, one from the lock call to the assert call, and the other from the *lea* instruction to the unlock call. Notice that the unlock call appears before the corresponding lock call in static program order although, in actual dynamic execution order, the lock call happens before.

```
pthread_mutex_lock(&__intern__);
assert(__threads__<__MAX_THREADS__);
pthread_create(&(__tid__[__threads__++]), NULL, (SlaveStart), NULL);
printf("==> created thread %d\n", __threads__-1);
pthread_mutex_unlock(&__intern__);
```

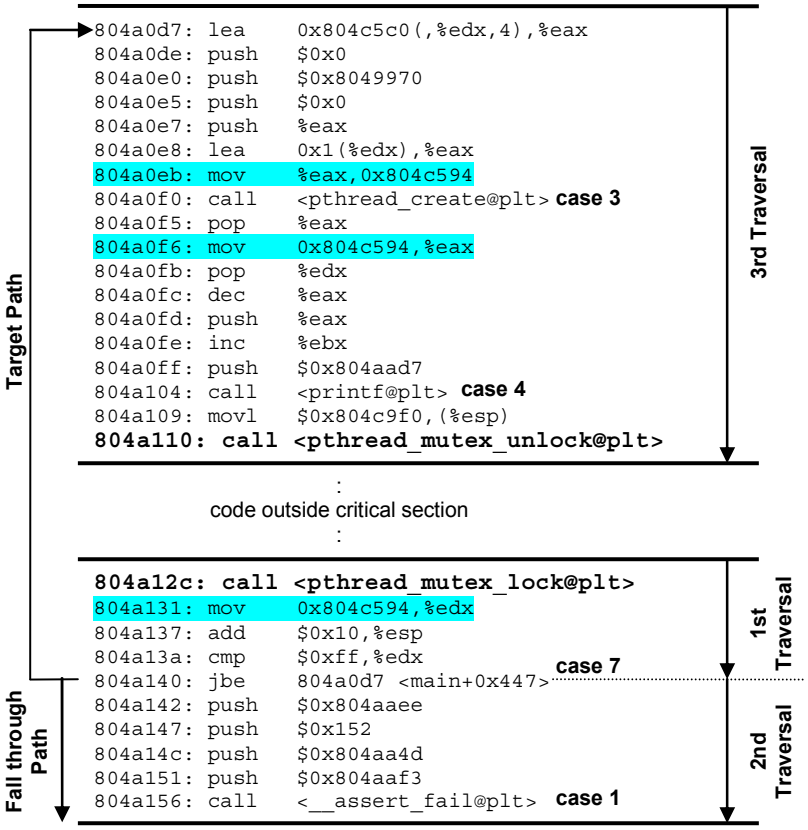


Figure 11: Manifestation of a critical section in the FFT kernel from the SPLASH2 suite. Its source level manifestation is shown in the box on top whereas its executable manifestation is shown directly below. Also shown here is the processing of this critical section at the executable level.

Output: After it traverses every critical section in the program, it produces a list of addresses of instructions that may execute inside of critical sections and access shared memory locations. It also obtains 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

The processing in this module together with the output of the analysis from the previous module completes the whole program analysis. The previous module reveals the kind of inter-procedural relationships among user routines, i.e., the call chain information. This module analyzes critical sections within individual user routines, and, hence, may be viewed as performing intra-procedural analysis. If there is never a call to another user routine from a given critical section, such intra-procedural analysis is sufficient. However, when a call to another user routine is encountered, a thorough analysis requires inter-procedural information obtained from the previous module.

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 Pin-ToleRace. First, let us address the inefficiency.

Addressing provisions for generality: The inefficiency is caused by uniformly implementing the full safe memory structure in every critical section. Now, given the knowledge of each critical section obtained from the static 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 operation. In particular, we

need to know whether a given critical section contains condition variables, overlapped critical sections, or nested critical sections. Choosing the path through the decision tree in Figure 11 from the root down to a leaf according to the analysis result gives the necessary structure needed for the safe memory. 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 indirection, which improves the efficiency of the safe memory accesses.

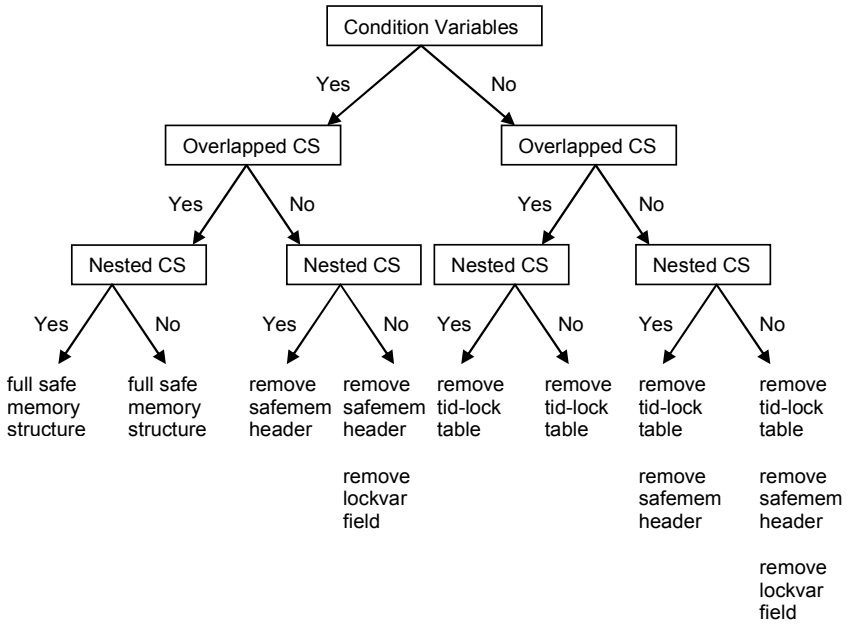


Figure 12: Decision tree for tailoring the safe memory structure for a given critical section.

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 of 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 bound, 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 that may never loop over shared memory accesses, we eliminate malloc/free calls, i.e., use 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 Pin's code cache. It instruments the code segments in question independent on whether they first execute inside or outside

of a critical section. 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.

However, when encountering indirect calls inside of a critical section, i.e., the critical section and the routines in its associated call chain contain indirect calls, static program analysis alone cannot deal with this situation thoroughly. We simply do not know the targets of such indirect calls until runtime. 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 add a call to 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.

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 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 presented in Table 4. Apparently, certain critical sections in some applications never get executed, for example, we statically found 43 critical sections in radiosity, only 36 of which are executed (see Table 4) with the given input.

Table 6: Critical sections properties 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

None of the programs in the two benchmark suites we consider 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 like, 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 pipeline-style parallelism. facesim is the only benchmark with a statically nested critical section. All critical sections in fluidanimate 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.

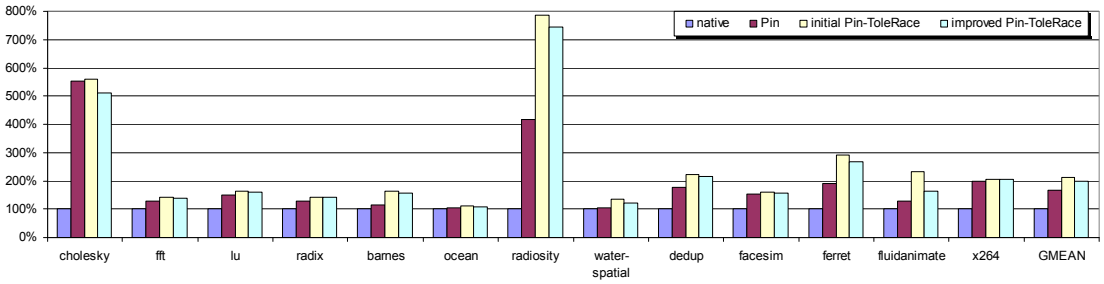


Figure 13: Normalized execution time for the improved version of Pin-ToleRace.

Figure 12 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 allocation instead 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.

CHAPTER 6

IDEALIZED SOFTWARE TOLERACE

This chapter concludes the discussion on software ToleRace implementations by investigating an ideal version of software ToleRace. It describes the third implementation, which is fundamentally different from the previous two schemes explained in Chapter 4 and 5. Whereas the first two implementations do not require accessibility to the program's source code and are based on a dynamic instumentator, this implementation presumes that the source code is accessible and ToleRace functionality is added by modifying critical sections directly at the source level.

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. In contrast, the previous two versions of Pin-ToleRace infer all the shared memory locations on-the-fly, thus, they yield an upper bound on the overhead.

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

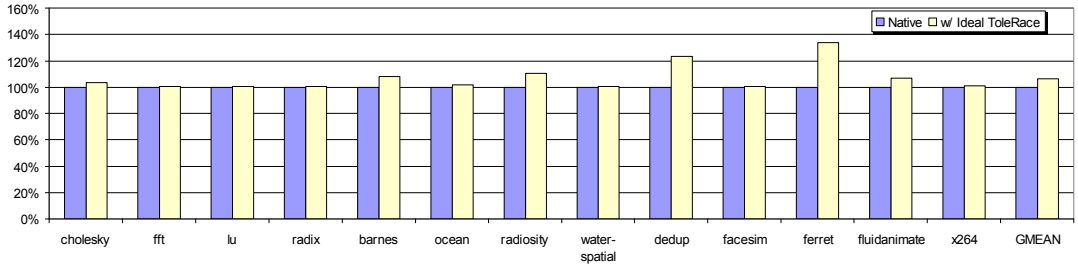


Figure 14: Normalized execution time of ideal software ToleRace.

After I incorporated ToleRace into the critical sections in this manner, I recompiled and ran these applications. Figure 13 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 runtime 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.

On average, when compared with around 2X overhead of Pin-ToleRace, the 6.4% overhead of the idealized software ToleRace is minuscule. This is understandable as most of the time, Pin-ToleRace is riding on the overhead of Pin itself. The ToleRace runtime overhead relative to the Pin overhead is about 24% on average. This is still about 4 times the average of the idealized software ToleRace. The drawback of this idealized ToleRace approach is that it requires accessibility to the program's source code. Thus, this may limit ToleRace's clients to programmers who write the softwares especially for commercial software where the source is usually not disclosed to general users. The Pin-ToleRace's approach, however, does not suffer from this disadvantage. It works directly on standard executables without any modifications and

commercial software is generally shipped in such binary form.

Suppose we want to adopt a runtime-based approach to ToleRace, like Pin-ToleRace, and we have access to the program's source code. We can take advantage of this situation to reduce the overhead of the runtime system. During the compilation process prior to runtime, the system performs source-level analysis and gathers as much information as possible about shared variable accesses inside of critical sections. However, not all information can be obtained at compile time. For ToleRace, the existence of dynamically generated shared variables necessitates runtime analysis. Nevertheless, for critical sections whose shared variable accesses can be inferred completely at compile time, e.g., those containing only scalar global variables, we want to modify those critical sections to incorporate ToleRace at the source level, compile them directly to the executable, and flag the ToleRace runtime to ignore them when executed. This should reduce the burden on the runtime system as it now only needs to handle those critical sections that truly require runtime analysis, i.e., those containing accesses to dynamic shared variables. For all other critical sections, the ToleRace mechanism is encoded directly in the executable. This combined approach should perform better than the Pin-ToleRace approach especially when the running application contains critical sections that mostly access static shared variables and are looped over several times.

CHAPTER 7

HARDWARE TOLERACE IMPLEMENTATION

7.1 Introduction

The previous chapters present software implementations of ToleRace. This chapter describes a hardware implementation of ToleRace.

As a consequence of Moore's Law, chip real estate continues to be abundant. However, microprocessor vendors find it progressively harder to turn the extra transistors into improved single-thread performance. They face extreme challenges on three microarchitectural fronts: the Instruction Level Parallelism (ILP), the memory wall, and the power wall. Single-thread performance started to level off in the year 2004. Realizing that the performance improvement trend, which had fueled the computer industry's growth for more than two decades, is in jeopardy, the chip vendors answered with a radically new way to revive the growth. This new way is, of course, multicore processors, which are already becoming ubiquitous.

Multicores are in essence symmetric shared-memory multiprocessors that share a common medium (a bus) for inter-processor communications. To date (July 2009), microprocessor manufacturers ship up to 16 general-purpose cores on a single chip. Users can take advantage of multicores in two fundamental ways: multiprogramming and multithreading. The former provides users with increased throughput when they run more than one program simultaneously. The latter can potentially allow single-thread performance to continue rallying but is much more difficult to exploit. It involves extracting parallelism out of sequential programs and turning them into parallel ones; an active research problem that we do not yet know how to solve in

general.

Given the ubiquity of multicores, it seems expedient that we implement hardware ToleRace with this microprocessor trend in mind, i.e., we want to investigate implementations that target multicore chips. As we shall see not much additional hardware is needed for a hardware implementation of ToleRace. Preexisting multicore components lend themselves well to embrace ToleRace's functionality. Recall that the ToleRace mechanism involves making thread-local copies of shared variables and operating on those copies. The private data cache in each core naturally serves as thread-local storage. Thus, the central idea behind the design of hardware ToleRace is to use existing cache components in multicores and leverage the already present cache coherence mechanism that maintains coherency among the private caches of the individual cores. In addition, we want the underlying hardware to be transparent to the running program. The program's executable should be able to run on top of it without requiring any modification to the binary or any user intervention.

Before we delve into the details of hardware ToleRace, we shall first characterize the multicore cache performance of the benchmark applications.

7.2 Cache Miss Characteristics of the Benchmark Programs

To measure the cache performance of a multicore, I wrote a lightweight Pin tool that simulates the cache operations in each private cache. The tool and all the benchmark programs run on a 32-bit x86-linux system and are compiled with a high optimization level (see Section 4.4.2). The simulator takes into account all x86 memory operations, including memory operands whose size is greater than 4 bytes, implicit memory operands (e.g, push and pop stack operations), and multiple read operands (e.g., in cmps instructions). In addition, it also accounts for unaligned accesses, i.e., an unaligned access may need to read two cache tags to determine whether the access is a

cache hit or miss.

Table 7: Cache miss ratios of the benchmark applications for different cache and block sizes

	32K		4M		16M	
	64 bytes	128 bytes	64 bytes	128 bytes	64 bytes	128 bytes
cholesky	1.59%	1.45%	0.25%	0.14%	0.17%	0.09%
fft	2.21%	1.47%	1.05%	0.53%	0.96%	0.48%
lu	1.84%	1.82%	0.18%	0.11%	0.02%	0.01%
radix	0.87%	1.14%	0.23%	0.14%	0.16%	0.09%
barnes	1.32%	1.82%	0.05%	0.04%	0.03%	0.02%
ocean	14.18%	12.55%	3.43%	1.73%	2.05%	1.03%
radiosity	0.87%	0.70%	0.02%	0.01%	0.02%	0.01%
water-spatial	0.55%	0.94%	0.00%	0.01%	0.00%	0.01%
dedup	1.37%	2.24%	0.08%	0.06%	0.06%	0.04%
facesim	1.51%	1.39%	0.39%	0.19%	0.17%	0.09%
fluidanimate	0.51%	0.60%	0.08%	0.07%	0.04%	0.04%
x264	2.82%	3.80%	0.12%	0.09%	0.06%	0.04%

The private caches of each core are tied together via a common bus. The bus is primarily a broadcast medium for inter-core communication. The tool simulates the MSI snoopy cache coherence protocol, which is an invalidation-based protocol. It treats the combined bus arbitration and transaction operations as happening atomically. Table 7 shows the cache performance of a 4-core processor configuration. All applications are designated to run with 4 threads to match the number of simulated cores. The numbers in the table indicate the overall cache miss ratios. Each core's private cache is direct-mapped. The results shown are for varying configurations, with cache sizes of 32 Kbytes, 4 Mbytes, and 16 Mbytes. For each size, the miss ratios for two block sizes, 64 bytes and 128 bytes, are measured.

As expected, the miss ratios reduce drastically across all the benchmark programs when the cache size increases from 32 Kbytes to 4 Mbytes for both block sizes. In

general, for an x86 architecture whose number of logical registers is small, we expect a lot of memory accesses due to register spills, and, hence, increasing the cache size can considerably reduce conflict misses. When the cache size is increased from 4 Mbytes to 16 Mbytes, the benefit gained is smaller. This indicates that the application might have already reached its working set size at the lower size cache. In some cases, e.g., cholesky, fft, radix, ocean, and radiosity, the 4 Mbyte cache outperforms the 16 Mbyte cache. For example, for fft, the 4 Mbyte cache with 128 byte blocks has a 0.53% miss ratio whereas the 16 Mbyte cache with 64 byte blocks incurs a 0.96% miss ratio. This can occur due to the following reasons. First, if an application has good spatial locality, large block sizes generate fewer cold misses. Second, the application accesses two large arrays alternately in a consecutive manner, i.e., good spatial locality accesses, that constantly conflict with each other. In this case, a smaller block size generates more conflict misses than a larger one. Of all the applications, ocean has the highest miss ratio for a given cache configuration, reaching as high as 14.2% for a 32 Kbyte cache with a 64 byte block size.

For the two large cache sizes, 4 and 16 Mbytes, almost all of the benchmark applications with the exception of water-spatial benefit from the larger block size. This indicates that these applications have significant spatial locality. For the smallest cache size, 32 Kbytes, the applications cannot fully exploit spatial locality because of the dominant conflict misses, which tend to increase as the block size gets larger. As we can see the miss ratios shoot up significantly when going from a 64-byte to a 128 byte block size in radix, water-spatial, dedup, and x264.

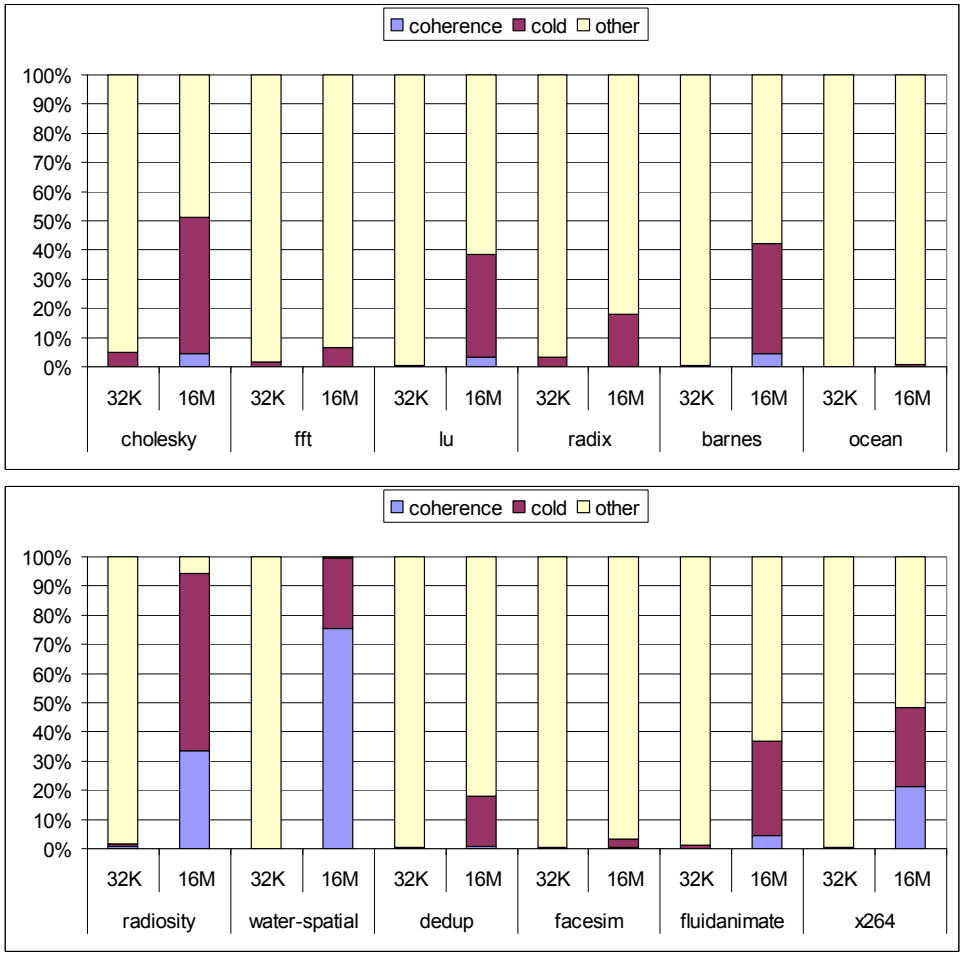


Figure 15: Breakdown of cache miss types in whole program execution. The top and the bottom panels show different sets of benchmark applications.

We will now take a closer look at the types of cache misses. Figure 14 shows the results with two cache sizes, 32 Kbytes and 16 Mbytes, each with a 1-byte block size. Because the block size is one byte, the coherence miss component is due only to true sharing. The category "other" includes both capacity and conflict misses. In this experiment, we consider all memory accesses performed throughout the entire program execution. As we can see from Figure 14, for the smaller cache size, conflict and capacity misses dominate and the coherence component is almost invisible. This

is because the shared blocks do not often remain in a private cache long enough to generate coherence activities. They are usually evicted because of conflict misses before they receive an invalidation message from another core. However, when the cache size is large enough to lessen the effect of conflict and capacity misses, the coherence miss component becomes significant. This is especially true for radiosity and water-spatial. In addition, for a number of applications, including cholesky, lu, radix, barnes, radiosity, water-spatial, dedup, fluidanimate, and x264, cold misses represent a significant fraction of the total misses.

Figure 15 shows results similar to Figure 14 but only considers memory accesses generated when the programs execute inside of critical sections. Here we see that coherence misses are the major miss component for the large cache size (16 Mbytes) in almost all of the benchmark programs. This is hardly surprising as the majority of the shared memory accesses happen within critical sections. They are the only accesses that generate coherence traffic, which in turn produces coherence misses. For the smaller cache size (32 Kbytes), conflict and capacity misses still dominate. Nevertheless, the coherence miss component is much more pronounced than it was when we considered all memory accesses.

For the programs that spend little time in critical sections, such as fft, radix, and ocean, we are only able to discern the coherence component for both cache sizes when we focus on memory accesses inside of critical sections. The conflict and capacity misses arising from accesses elsewhere dominate the total miss count. water-spatial is quite special, however. It also executes rarely inside of critical sections, but coherence misses suddenly become the main miss component when increasing the cache size from 32 Kbytes to 16 Mbytes even when we consider all accesses from the whole program run. This is because conflict and capacity misses are removed almost completely when the large cache size is used. All that is left, therefore, is cold and

coherence misses. We can see from both Figure 14 and Figure 15 that, for this application, the miss component profile when considering all accesses is largely the same as that when only considering accesses in critical sections for the large cache size.

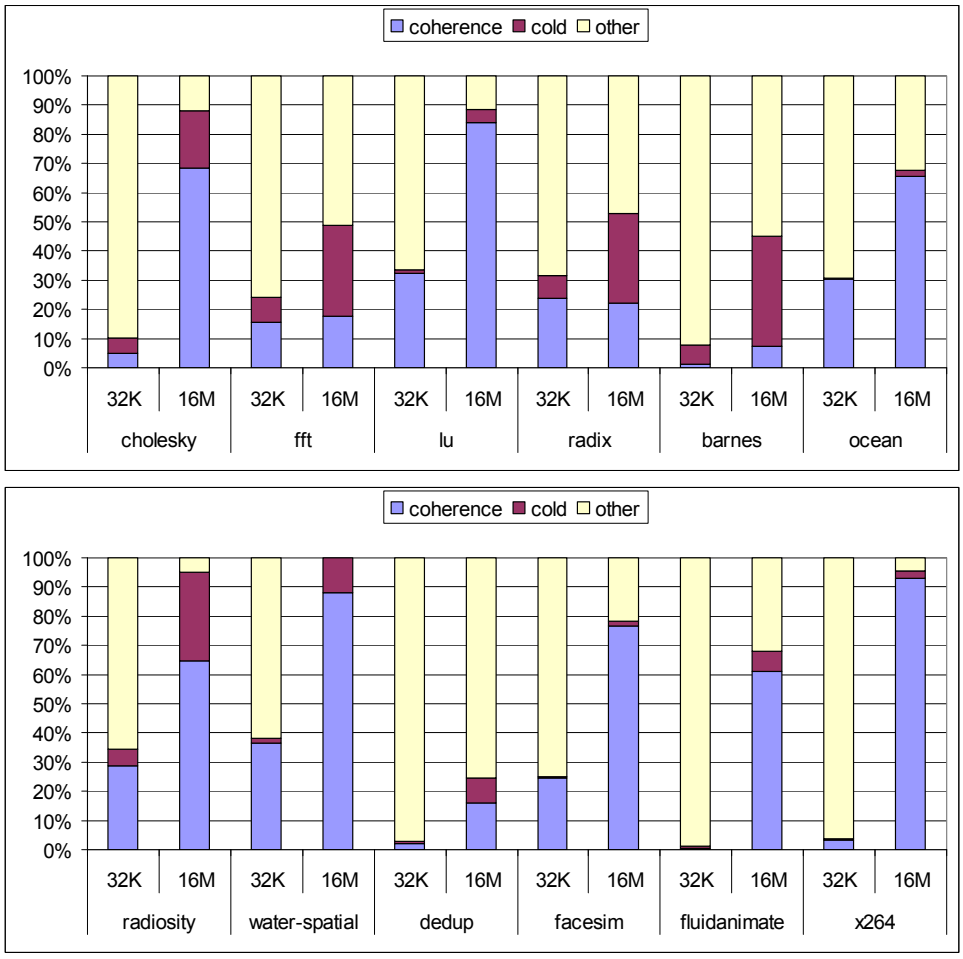


Figure 16: Breakdown of cache miss types inside of critical sections. The top and the bottom panels show different sets of benchmark applications.

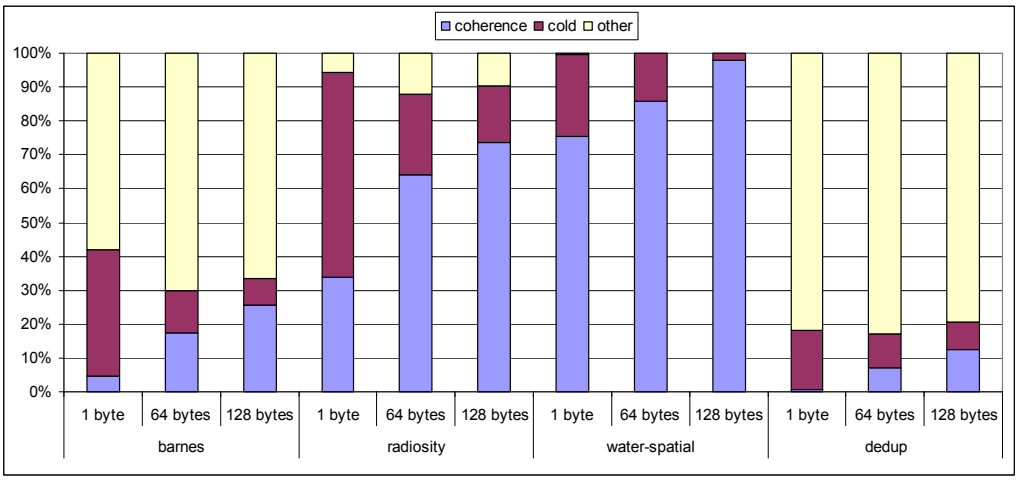


Figure 17: Effect of increasing block size on the composition of cache misses.

Next, we measure the effect of increasing the block size while keeping the cache size constant. The results are shown in Figure 16. In this experiment, each core has a 16 Mbyte direct-mapped private cache. We consider all accesses from the entire program execution and show the results for the four benchmarks, barnes, radiosity, water-spatial, and dedup, whose coherence miss components are discernible. We investigate three block sizes, 1, 64, and 128 bytes. The 1-byte size reflects the situation where only coherence traffic due to true sharing occurs. On all four applications, we see that increasing the block size leads to an increase in the contribution of the coherence miss component. This effect is largely attributed to the presence of more coherence messages due to false sharing as the block size increases.

7.3 Hardware ToleRace Design

This section describes the design of hardware ToleRace. It builds on top of the existing private caches in a multicore. The proposed design assumes write-back caches and leverages the MSI snoopy invalidation-based cache coherence protocol. To

support the ToleRace functionality, the cache coherence protocol is augmented with extra states and transitions. I believe that invalidation-based protocols are preferred over update-based protocols and write-back caches are more favorable than write-through caches in a multicore because both consume markedly reduced inter-core communication bandwidth relative to their counterparts. Hence, I propose the design of hardware ToleRace per the above assumptions.

7.3.1 Basic Design and Operation

The basic structure of the proposed hardware ToleRace is shown pictorially in Figure 17. The main components represent the safe memory region, which extends the private cache region in each core. This safe memory is placed at the cache level that receives coherence traffic from other cores. Its structure is similar to that of a victim cache [28]. For this basic design, the effect of context switching will not be considered. We will revisit this issue in Section 7.3.6.

Note that the design does not require any structural changes to the existing private cache, which still contains the standard components, arrays of valid bits, tags, and the cache blocks themselves. However, as we shall see later, the cache coherence protocol needs to be augmented so that the private cache can work with the safe memory in a synchronized fashion.

There are four basic components in the safe memory: the evicted cache block region, the active bit, sets of Access Bit Vectors (ABV), and two status bits.

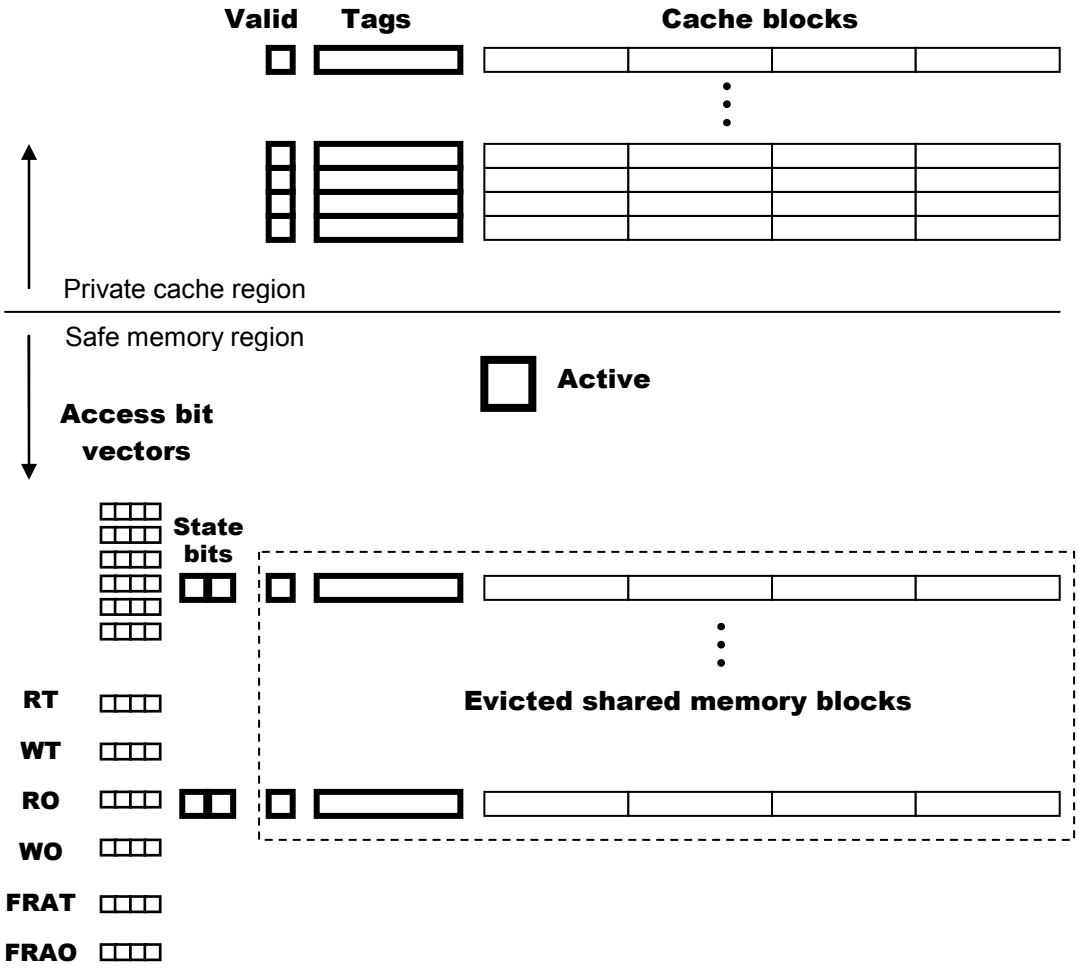


Figure 18: Basic structures of the proposed hardware ToleRace.

Evicted cache block region:

The general structure of this component is the same as the private cache. Each block contains a valid bit, a tag, and the actual data. The key difference is that an entry is searched associatively and not by indexing using parts of the address bits. Therefore, the tag for each entry contains the tag bits of the original block in the private cache as well as the index bits. Basically, a block is transferred from the private cache to this

evicted region in the safe memory whenever the CPU of the core in consideration accesses potentially shared memory locations that reside in this block. I will describe this evicted cache region more when I discuss the basic operation of the proposed hardware ToleRace. In Section 7.3.2, I will present a quantitative evaluation of this design. The main purpose of this evaluation is to be able to appropriately size the number of entries for this evicted cache region.

Active Bit:

The active bit specifies whether there are valid entries in the safe memory. When this bit is on, the hardware ToleRace system signals to the cache controller to also look for cache entries in this region. Whenever the controller receives a request from the CPU or from the bus, it first searches the private cache area. If a miss results, it continues the search in the safe memory before going to the lower levels of the memory hierarchy.

Table 8: Description of each Access Bit Vector.

Access Bit Vectors (ABV)	Descriptions
Read This (RT)	Bit i of RT is set when a **read** access from **this** core touches the i^{th} byte
Write This (WT)	Bit i of WT is set when a **write** access from **this** core touches the i^{th} byte
Read Others (RO)	Bit i of RO is set when a **read** access from the **other** cores touches the i^{th} byte
Write Others (WO)	Bit i of WO is set when a **write** access from the **other** cores touches the i^{th} byte
First This Access Read (FTAR)	Bit i of FTAR is set when the **first access** that touches the i^{th} byte from **this** core is a **read**
First Others Access Read (FOAR)	Bit i of FOAR is set when the **first access** that touches the i^{th} byte from the **other** cores is a **read**

Access Bit Vectors (ABV):

There are six cache block bit vectors that serve as bookkeeping mechanism for the types of accesses to each byte on an evicted cache block. These are Read This (RT), Write This (WT), Read Others (RO), Write Others (WO), First This Access Read

(FTAR), and First Others Access Read (FOAR). The width of each ABV is equal to the size of the cache block. The purpose of each ABV is given in Table 8. "This" refers simply to this core; "Others" denotes all the other cores.

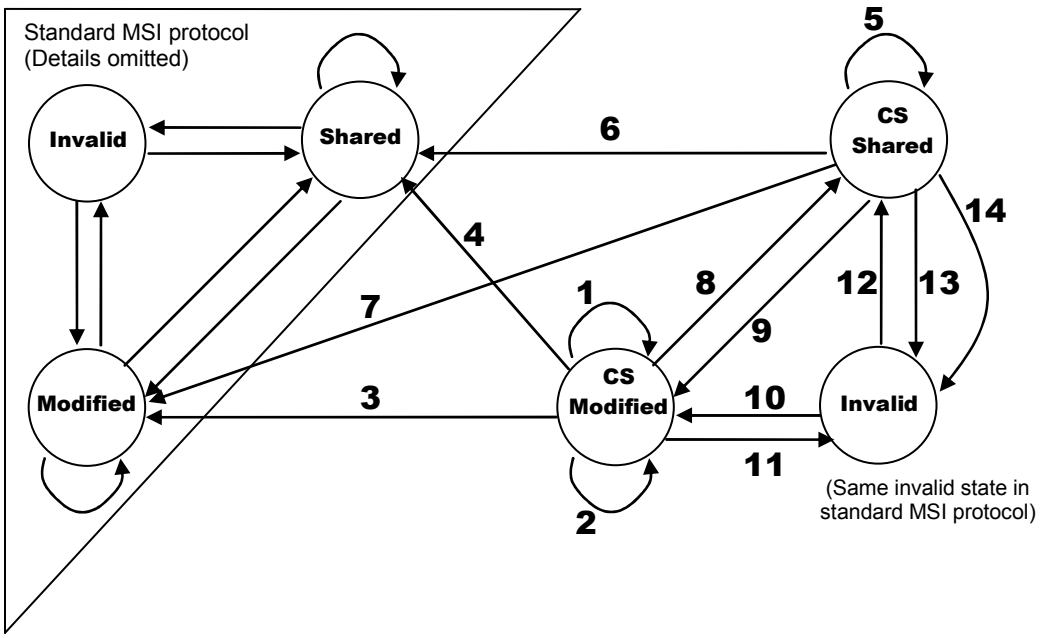
State Bits:

The two state bits for each cache block entry indicate the state of this cache block that exists in private caches of other cores. Their use will become clear when we discuss the changes to the coherence protocol and the basic operations of hardware ToleRace.

7.3.1.1 Modifications to the Standard MSI Cache Coherence Protocol

As mentioned before, I assume that the multicore uses the standard MSI cache coherence protocol. The proposed hardware ToleRace requires a few changes to this standard protocol. I believe that the modification I am presenting here can readily be applied to other similar invalidation-based protocols such as MESI. As we will see, the modification is not overly complicated. It involves 1) adding a new Modified (M) and a new Shared (S) state called Critical Section Modified (CSM) and Critical Section Shared (CSS), which are almost exact replica of the M and S states, respectively, 2) modifying some state transition actions between and within the original M and S states, and 3) adding a new set of state transition actions between the two new CSM and CSS states that are analogous to what the M and S have.

The modifications are depicted graphically in Figure 18. The details pertaining to the standard MSI protocol are omitted for clarity and interested readers are refered to, e.g., Culler et al. [14]. The underlined actions are those that directly relate to ToleRace's functionality. The key points to note in this augmented protocol are:



1: CPU read hit; **place read message on bus**
2: CPU write hit; **place invalidation message on bus**
3: CPU write miss; **place write miss on bus and write back**
4: CPU read miss; **place read miss on bus and write back**
5: CPU read hit; **place read message on bus**
6: CPU read miss; **place read message on bus**
7: CPU write miss; **place write miss on bus**
8: **Read miss for block received; write back; place read message on bus**
9: CPU write hit; **place invalidation message on bus**
10: CPU write miss and block found in safe memory; **place write miss on bus**
11: **Write miss for block received; write back**
12: CPU read miss and block found in safe memory; **place read miss on bus**
13: **Write miss for block received**
14: **Invalidation for block received**

Figure 19: Augmenting the standard MSI cache coherence protocol to enable ToleRace. Bus requests are in bold fonts whereas CPU requests are in normal fonts. Underlined actions are the key differences from the standard MSI protocol. All write backs are to the shared memory. When receiving a read miss message on the bus, another core wants to read from the block in CSM state. When receiving a write miss message on the bus, another core wants to write to the block in CSS state. That other core has the block in Invalid state.

1. It adds the two new states, CSS and CSM, as mentioned above.
2. It adds the new transitions to go from the Invalid (I) state to the two new states, CSS and CSM (transitions 10 and 12 in Figure 18).
3. It dictates that a read hit while in the CSS state propagate a read message on the shared bus (transition 5). Recall that in the original S state, a read hit never generates additional bus traffic and is confined only to the private cache of the core in consideration.
4. It requires that a read hit and a write hit while in the CSM state place the corresponding read and invalidation messages on the bus (transitions 1 and 2). The corresponding transitions in the original M state never propagate these read and invalidation messages on the bus.

The changes noted in points 3 and 4 above allow the safe memory in a given cache core to snoop for intervening reads or writes that would otherwise be confined locally to the private caches of the other cores. Note that the cache coherence engine only runs in the private cache region; it does not run in the safe memory region.

In addition to the changes above, this augmented protocol requires that all invalidation and read messages for a given access contain the full address, the number of bytes being accessed, and the cache block state. The last information is used to set the two state bits. Only two bits are required for we are only concerned with the I, CSS, and CSM states.

7.3.1.2 Basic Operation

Having described the major components and the required modification to the standard cache coherence protocol to accommodate ToleRace's functionality, I will now discuss the basic operation of the proposed hardware ToleRace.

Initially the safe memory in each core is not active; all the ABVs, and the valid bit for each block in the evicted cache block are cleared. When the core's CPU detects that:

1. the program executes in a critical section and
2. it accesses possibly shared memory locations, i.e., all non-stack accesses,

the CPU evicts the cache block being accessed from the private cache region and send the block to an entry in the evicted cache region in the safe memory. Once the block is placed in the safe memory, the hardware ToleRace broadcasts an invalidation message to nullify all other copies of this cache block in this and all other cores. It also sets the Active Bit in the safe memory.

When the Active Bit is on, there are two cases to consider:

1. subsequent accesses to this block come from this core, and
2. subsequent accesses to this block come from another core

In the first case, the following happens:

1. The accesses never bring the block back to the private cache; they always take a miss there and only consult the block in the safe memory. As a consequence of this, every time when the Active Bit is on, any misses in the private cache need to search the safe memory for a matching entry first before going to the next level of the memory hierarchy.
2. The accesses set the appropriate bits in each of the RT, WT, and FTAR bit vectors. Recall that these sets of ABVs are for this core and are set based on this core's CPU requests.

In the second case, we may potentially have a race. The following happens:

1. The accesses observe the augmented cache coherence protocol described previously as they are accessing the private cache.

2. The safe memory in each core with its Active Bit on snoops on the bus for messages that may be relevant, and sets the appropriate bits in each of the RO, WO, and FOAR vectors.

Upon exiting from a critical section, the ToleRace hardware resolves races according to the table in Figure 19. The race resolution happens on a per byte basis using the information from the six Access Bit Vectors. The information from the RT, WT, RO, and WO bit vectors are sufficient to infer some race types. However, as we can see in the table in Figure 19, for the entries labeled A, B, and C, we need the information stored in the FOAR and FTAR bit vectors to pin down the exact race type in these cases. In order to resolve the race correctly, the ToleRace hardware retrieves the latest block copy that resides outside of the safe memory. This block should be in one of the three states, I, CSS, or CSM. To do so, the hardware consults the state bits. If the block is in either I or CSS, it gets the latest copy from the main memory; otherwise it gets the block from the core that has the block copy in the CSM state. Up on receiving the latest copy, the hardware inspects the block byte by byte. Based upon the race resolution in each byte as per Figure 19, it determines if the copy of each byte to pass on should be coming from the block in the safe memory or the block outside. In the former case, the hardware masks out, i.e., replaces the respective byte in the outside block with the copy from the safe memory block; for the latter case, the converse happens.

RT	WT	RO	WO	Race type	Toleration strategy	
0	0	0	0	No race	N.A.	
0	0	0	1	No race	N.A.	
0	0	1	0	No race	N.A.	
0	0	1	1	No race	N.A.	
0	1	0	0	No race	N.A.	
0	1	0	1	No race	N.A.	
0	1	1	0	Wr+W	r+WW	
0	1	1	1	Wr+wx*W, Wwx*W	r+wx*WW, no race	A
1	0	0	0	No race	N.A.	
1	0	0	1	Rw+R	RRw+	
1	0	1	0	No race	N.A.	
1	0	1	1	Rr+wx*R, Rwx*R	RRr+wx*, RRwx*	B
1	1	0	0	No race	N.A.	
1	1	0	1	Xw+X	XXw+	
1	1	1	0	Xr+X	r+XX	
1	1	1	1	Wr+wx*X, Rr+wx*W, Wwx*X, Rwx*W	r+wx*WX, intolerable, WXwx*, RWwx*	C

Case A:

FOAR	Race type
0	Wwx*W
1	Wr+wx*W

Case B:

FOAR	Race type
0	Rwx*R
1	Rr+wx*R

Case C:

FTAR	FOAR	Race type
0	0	Wwx*X
0	1	Wr+wx*X
1	0	Rwx*W
1	1	Rr+wx*W

Figure 20: Hardware ToleRace resolution table based on Access Bit Vectors.

After successfully resolving the race in each byte for this block, the ToleRace hardware processes the next valid block in the safe memory until it visits all the valid blocks in the evicted cache region. Then, it resets all the valid bits, the ABVs, and the Active Bit. Note that the hardware needs to ensure that this process happens atomically. A straightforward way of ensuring this is to lock the bus so that all the bus transactions made during this time only come from the core that is resolving the race. As I will show in the next section, critical section memory accesses constitute only a small fraction of the total memory accesses, so this simple but seemingly costly way of ensuring atomicity should work just fine. Furthermore, most of the time while executing inside of critical sections, the number of entries in the evicted cache region is small; in many cases, only one entry is accessed per critical section execution.

7.3.2 Quantitative Assessment of the Proposed Hardware ToleRace

This sub-section investigates quantitatively the characteristics of the safe memory

when running the benchmark applications. In particular, it focuses on the number of entries in the evicted cache of the safe memory that each application run generates while it executes inside a given critical section. As we see from the previous section, entries in the evicted cache region are searched associatively and the hardware ToleRace needs to resolve the race entry by entry upon a critical section exit. Hence, the smaller the number of entries is, the higher the performance.

Unless otherwise noted, for the rest of this section, I simulate each application run by taking into account both user and library codes. This is in contrast to the previous chapters on software ToleRace, which only considered user code. The reason for this change is that I assume there is no hardware support that is able to distinguish user code from library code on-the-fly.

Table 9: Fractions of all memory accesses that are inside of critical sections.

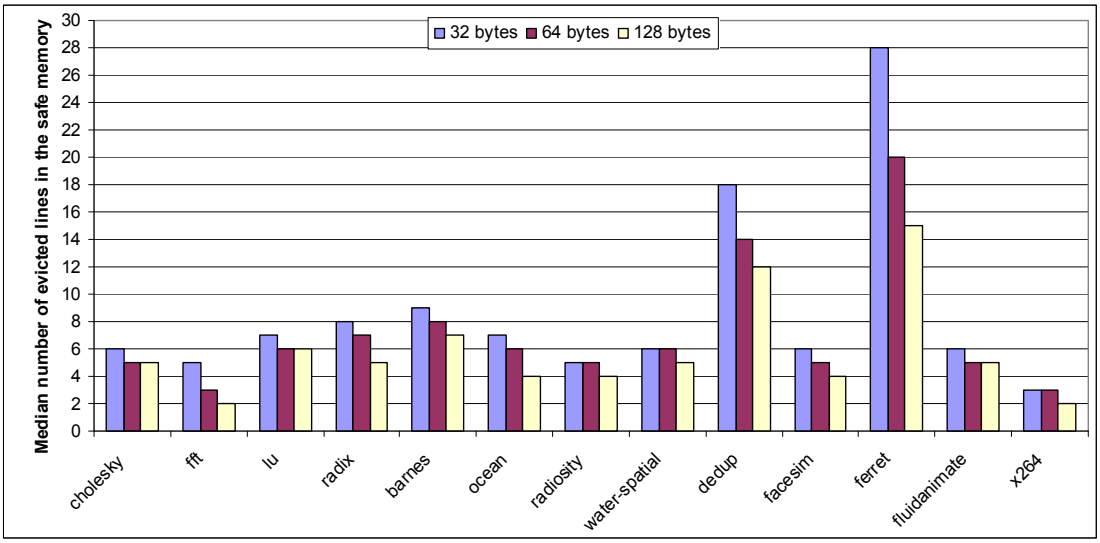
cholesky	0.6%
fft	< 0.001%
lu	< 0.001%
radix	< 0.001%
barnes	0.43%
ocean	< 0.01%
radiosity	0.49%
water-spatial	< 0.01%
dedup	0.73%
facesim	< 0.01%
ferret	1.22%
fluidanimate	1.36%
x264	< 0.01%

We first look at the memory accesses inside of critical sections as compared to the total memory accesses. Table 9 shows the percentage of the total memory accesses that are performed inside of critical sections. All of the benchmark applications except ferret and fluidanimate have less than one percent of critical section accesses. These

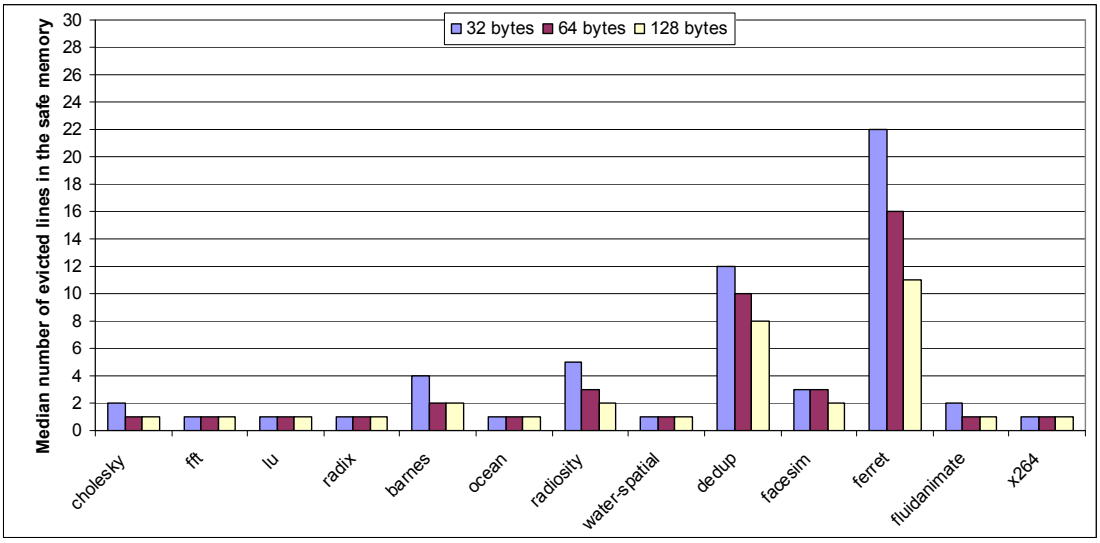
numbers are in agreement with the critical section characteristics of each benchmark, which we studied in Chapter 4. It confirms that critical section accesses are, indeed, not frequent events. This gives us some confidence that the performance penalty incurred by the proposed hardware ToleRace design should not be significant.

Next, I perform an experiment to collect the number of entries in the safe memory required for each critical section execution to successfully enforce the ToleRace mechanism. I assume an unbounded number of entries to begin with. The result in Figure 20 shows the median number of entries. For each application, I consider three block sizes, 32, 64, and 128 bytes.

Expectedly, as the block size increases the number of entries becomes smaller. When considering both user and library code (Figure 20(a)), the median number of entries is less than ten for all three block sizes and for all applications except dedup and ferret. With a 64-byte block size, most applications have a median number of entries of less than six. Recall that dedup and ferret execute loops inside critical sections and, hence, we expect that the number of entries will be higher for these two applications than for the others.

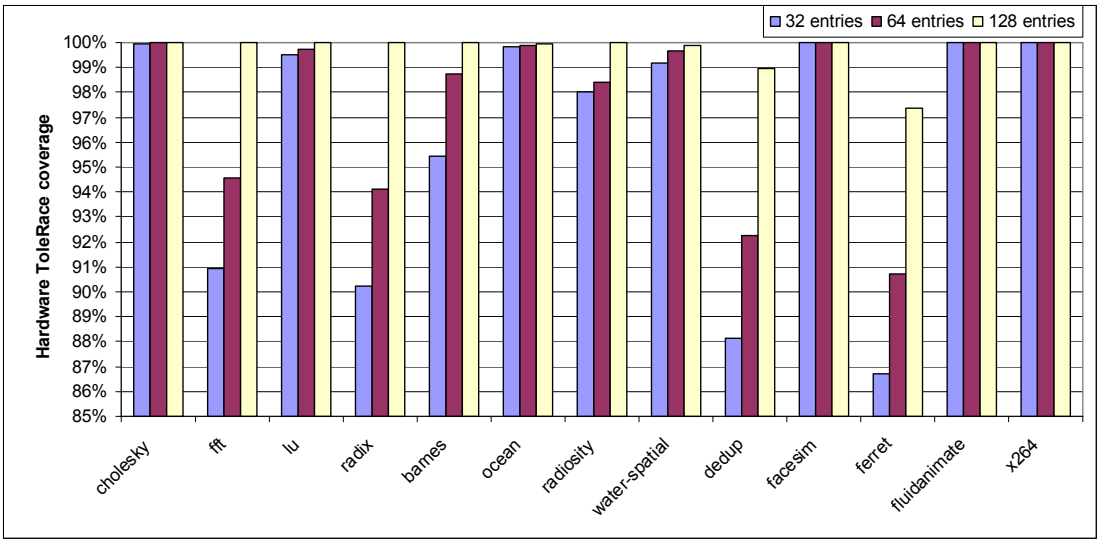


(a) considering both user and library code

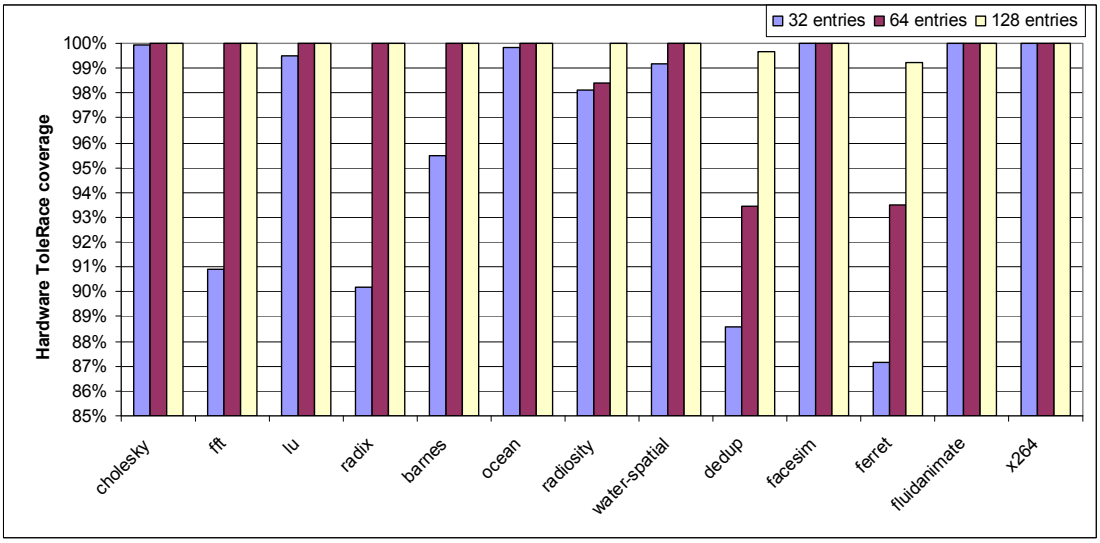


(b) considering only user code

Figure 21: The median number of entries in the evicted cache region of the safe memory, considering both user and library code (a) and user code only (b). The three bars in each application indicates three different block sizes of 32, 64, and 128 bytes.



(a) 32 bytes block size

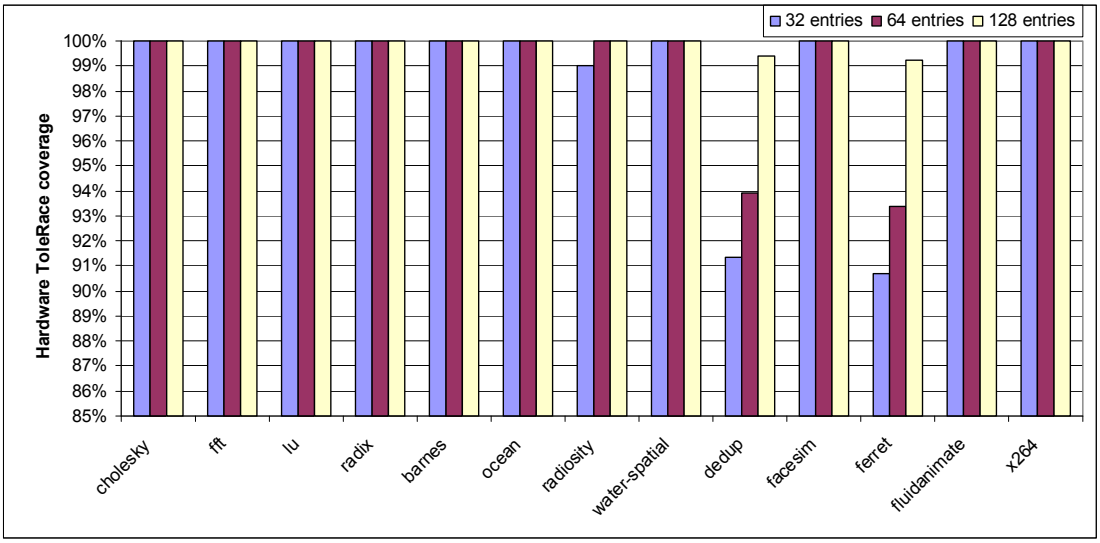


(b) 64 bytes block size

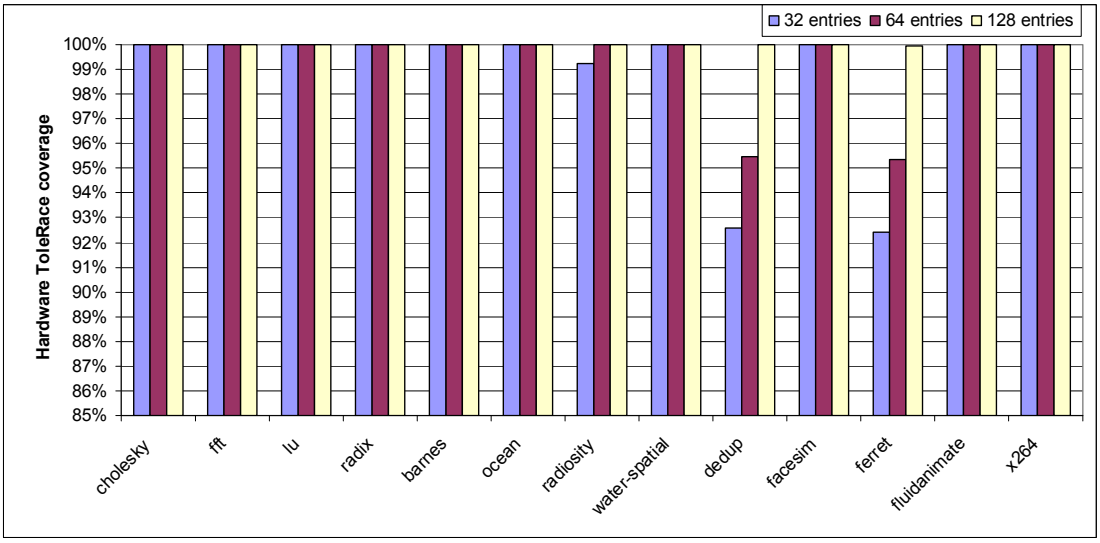
Figure 22: Hardware ToleRace coverage of critical section execution for 32 byte blocks (a) and 64 byte blocks (b). The three bars in each application indicate different numbers of entries, 32, 64, and 128, in the safe memory's evicted block region.

Suppose we have the ability to distinguish user code from library code. Then, the number in Figure 20(a) for each application can be further reduced. Figure 20(b) shows the results when the library code execution is excluded. With a 64-byte block size, a number of applications, namely, cholesky, fft, lu, radix, ocean, water, fluidanimate, and x264, have a median number of entries of only one. The numbers for dedup and ferret are reduced in this case as well, although they are still significantly larger than those of all the other applications.

In the next experiment, whose results are shown in Figure 21, I fixed the number of entries in the safe memory for each application at 32, 64, and 128, respectively. I then measure the fraction of critical executions that are fully covered by the proposed hardware ToleRace scheme, i.e., there is no overflow due to the finite number of entries in the evicted cache region. The results measured are for two block sizes, 32 and 64 bytes. As we can see, for 64 entries with 64-byte blocks, over 93% of all the critical section executions in each application are covered. For the same 64-byte block size with the number of entries doubled to 128, the coverage increases to 99%. Even though they do not execute long running critical sections, fft and radix have a significantly lower coverage than the other applications except dedup and ferret for 32 and 64 entries. This is because their critical section executions are almost negligible and missing one or two critical section executions contributes substantially to the drop of the coverage.



(a) 32 bytes block size



(b) 64 bytes block size

Figure 23: Hardware ToleRace coverage of critical section execution for 32 bytes block (a) and 64 bytes block (b) when considering user code only. The three bars in each application indicate different numbers of entries, 32, 64, and 128, in the safe memory's evicted block region.

Figure 22 shows the results similar to Figure 21 but with library code excluded. We can see markedly improved performance here. With the exception of dedup, ferret, and radiosity, all the applications have 100% critical section coverage at 32 entries with 64-byte block. For the three exempt applications, dedup, ferret, and radiosity, the coverage is 92.6%, 92.4%, and 99.25%, respectively.

7.3.3 Nested and Overlapped Critical Sections

The proposed basic ToleRace hardware is primarily designed to handle non-nested non-overlapped critical sections. The simplest way to extend the basic hardware to cope with nested critical sections is to add a simple counter that keeps track of the nesting level just like the CSLevel counter does in the software ToleRace system. When the value of this counter is greater than zero, accesses to shared memory happens inside of the safe memory. Race resolution takes place only when the counter reaches zero. This simple extension effectively flattens the inner critical sections. The hardware ToleRace disguises them by continually protecting shared accesses associated with the inner critical sections until the outermost unlock is encountered, i.e., exiting from the outermost critical section. Unfortunately, this simple scheme does not faithfully preserve the original lock-based program semantics for nested critical sections, which stops protecting shared accesses associated with an inner lock as soon as the corresponding inner unlock is found. In addition, it does not correctly handle overlapped critical sections.

To remedy this situation, we need to have the ability to perform race resolution at every critical section exit. Consequently, just like in the case of software ToleRace, we now need to associate a lock variable with each access. Since the hardware ToleRace resolves races at byte granularity, having a lock variable associated with each byte in a cache block is potentially costly. Nevertheless, if we take a less ambitious stance and

decide not to support arbitrary levels of nesting or overlapping, we can scale back the hardware considerably. To decide what the appropriate nesting level is to support in hardware ToleRace, I measured from the characteristics of the benchmark programs. From these programs, we see that hardly will we encounter a nesting level greater than two. Hence based on this empirical evidence, I propose to augment the basic hardware ToleRace to support nested and overlapped critical sections whose nesting level is at most four.

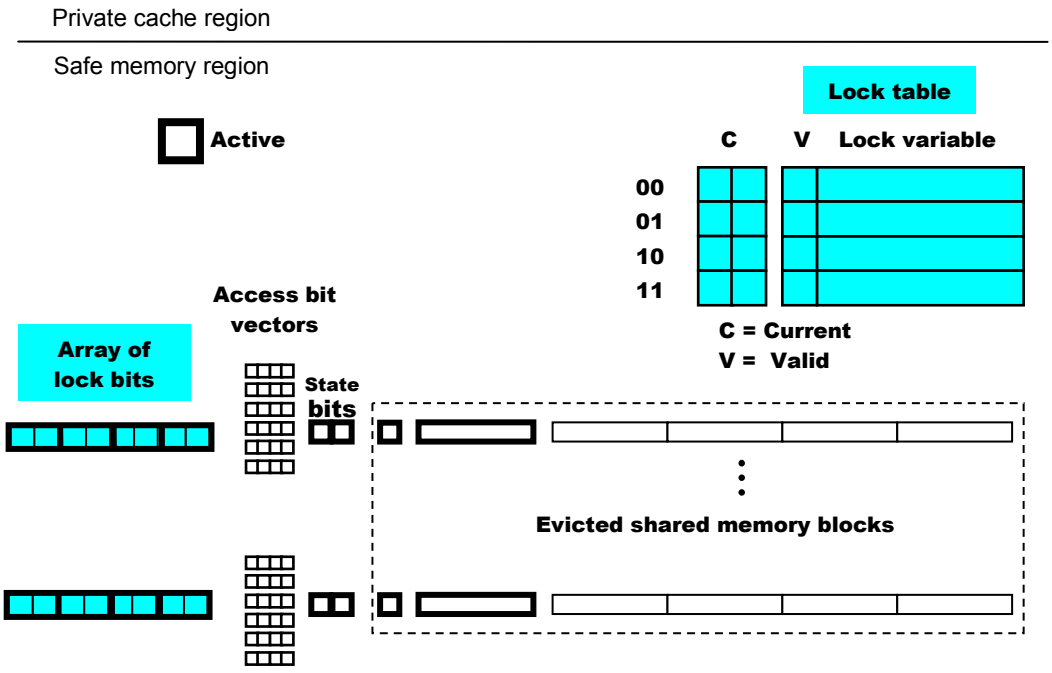


Figure 24: Augmenting the basic safe memory structure to accommodate nested and overlapped critical sections.

Figure 23 depicts additional structures to support nested and overlapped critical sections. The new structures, an array of lock bits and the lock table, are highlighted in the figure. Each byte in a cache block has two lock bits associated with it. The two

lock bits are mapped to a lock variable by simply indexing into the corresponding entry in the lock table. Note that the lock bits for each byte are interpreted only when either the corresponding RT or WT bit is set. Otherwise, they are disregarded since there are no shared accesses to this byte inside critical sections. For each entry in the lock table, the valid bit specifies whether the entry contains a valid lock variable. When the valid bit for an entry is set, a two-bit current counter value of zero indicates that the lock variable in the entry is the most current, i.e., the inner most; an entry with the highest current counter value belongs to the outermost lock.

With the additional structures, the basic operation is amended as follows. Initially, all current and valid bits in the lock table are cleared. When entering a critical section, the hardware searches for an "empty" entry whose valid bit is not set, registers the associate lock variable there, sets the valid bit, sets the current counter to zero, and increment all the current counters of other valid entries by one. When exiting a critical section, it searches for the corresponding lock variable in the table, clears the valid bit of that entry, and remaps the current counter values of all the other valid entries. During remapping, the entry with the lowest counter value is reassigned to zero, the next lower value to one, and so on. In doing so, all the valid counter values maintain a consecutive order. If we encounter only nested non-overlapped critical sections, the lock table grows and shrinks like a stack. The highest entry is at the top of the stack and its current counter value is zero.

When a cache block in the safe memory is accessed, each byte involved has its corresponding lock bits set to the entry of the most current lock, i.e., the entry with the current counter value is zero. At race resolution time, each cache block in the evicted region is inspected one by one just like in the basic operation. If, for a given block, each byte accessed has the same value of lock bits associated with the most current lock, the same procedures as in the basic operation apply and the lock table's data

structures are updated upon exiting the critical section as described above. However, if there are some bytes accessed whose lock bits are not associated with the most current lock, the hardware needs to retain this block in the safe memory and perform race resolution for the bytes associated with the most current lock. It follows the race resolution table in Figure 19, clears the corresponding ABVs, and writes back each byte involved with the appropriate copy. The copy of the byte in the block in the safe memory is the same as the copy in the shared memory.

7.3.4 Handling Condition Variables

Figure 24 shows the two additional structures, the conditional wait bit and the wait counter, to deal with condition variables. Recall that when we encounter a condition variable, a thread is allowed to sleep inside of the critical section, and, hence, more than one thread can be inside of the critical section at the same time. The conditional wait bit indicates that the safe memory is involved in a condition variable synchronization and the wait counter specifies how many threads are currently waiting inside of the critical section. Handling condition variables in hardware ToleRace is quite complicated. The basic operation needs to be modified such that each thread involved in a given condition variable sees the same consistent safe memory. Here are the operations that the hardware ToleRace needs to follow:

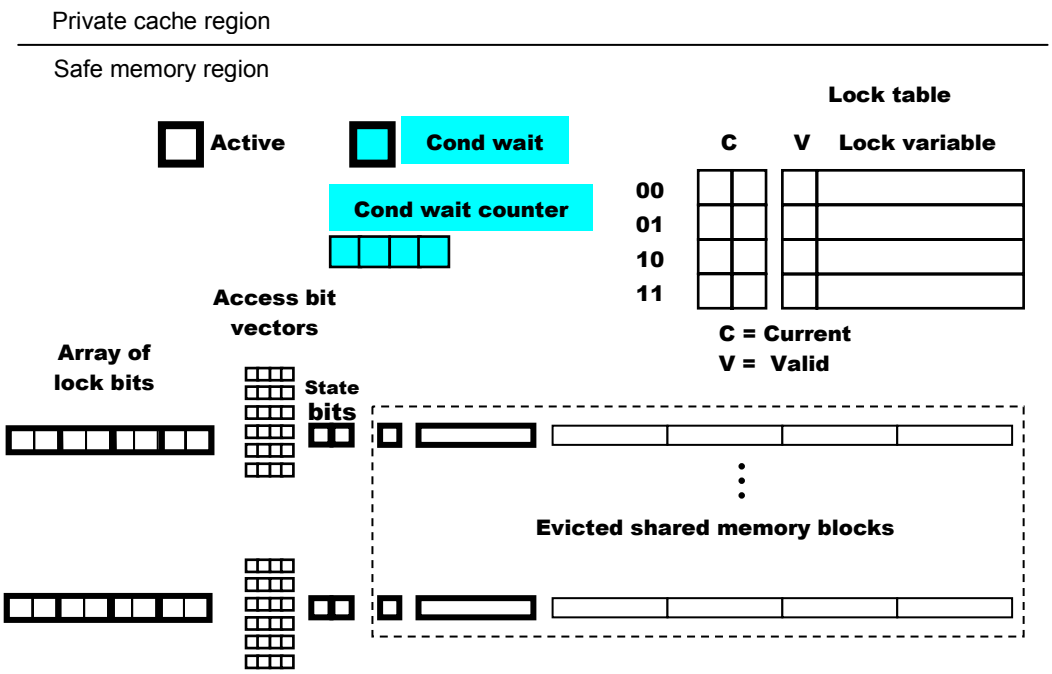


Figure 25: Adding the conditional wait bit and conditional wait counter to cope with condition variables.

1. When a thread accesses the safe memory in a core, the hardware ToleRace needs to broadcast the associated lock variable on the shared bus.
2. The core with the conditional wait bit set snoops on the bus for the lock variable message and compares the snooped variable with its current lock variable, i.e., by looking up the lock table.
3. If there is a match, the thread that has sent the message from another core is entering the critical section with the same condition variable. In this case, the hardware ToleRace proceeds to copy the safe memory from this core to that other core.
4. When accessing the safe memory with the conditional wait bit set, the hardware ToleRace uses an update-based protocol so that all writes to the current core are

reflected on all the other safe memories on other cores whose conditional wait bit is set.

5. Whenever a new thread enters the same condition variable critical section in a core, a message is broadcasted over the shared bus so that the wait counter in every safe memory that has its conditional wait bit set gets incremented. Similarly, if a thread exits from the critical section, all the wait counters must be decremented. The resolution of races does not happen until the wait counter becomes zero.

7.3.5 Resolving Issues with Existing Binaries and Load-Store Orderings

One major appeal of ToleRace is that it works on existing lock-based binaries without any modifications thereof. This section outlines a scheme to "transform" the original program binary to execute on ToleRace hardware. In addition, since hardware ToleRace is likely to be implemented on top of an out-of-order core, issues with load/store orderings need to be addressed.

As prerequisites, the ToleRace hardware must be able to recognize the beginning and end of a critical section. With out-of-order processors, it must further make sure that no load and store operations from inside of critical sections are moved outside. Similarly, load and store operations from outside of critical sections must not be moved inside. I propose the following scheme to satisfy these prerequisites.

In the first step, we use binary interception [27] to intercept calls to pthread libraries that define critical section enters/exits. Then, we rewrite the first few bytes of the relevant libraries to call detoured routines that contain special sequences of instructions. The ToleRace hardware is hardwired or programmed to recognize these special sequences so that it can tell when to start and stop enabling ToleRace. This

detoured routine must also return back to the original pthread library and, subsequently, the initiating call site. One of the instructions in the detoured routine needs to perform a fence operation. For example, the MFENCE x86 instruction accomplishes this. It inhibits any load or store operations across the fence.

7.3.6 Fallback Mechanism

When considering a hardware solution, it is essential to have a recovery mechanism in place in case the hardware resources are exhausted. This section describes one possible way to recover from such a scenario. Given that hardware is typically carefully sized for the common case, recoveries are expected to be rare. Therefore, the proposed mechanism is designed to be simple and safe but not necessarily the most efficient.

If the ToleRace hardware resources are exhausted while the program is inside of a critical section, there are two possibilities to consider:

1. a race has already been detected
2. a race has not yet been detected

In the first case, the fallback is simply to stop program execution and report the race. It is not advisable to continue program execution after a race occurs. Doing so would be tantamount to tolerating the race in the middle of a critical section, which is not a defined behavior in ToleRace.

Therefore, the focus of the fallback mechanism is on the second case where the program execution must be allowed to continue. Abruptly stopping the program execution is not an acceptable solution in this case. After all, the introduction of ToleRace should not interfere with a legitimate race-free run of a program. A clean way to recover from this scenario is explained next.

The basic idea is to reconcile the copies of each of the active cache blocks in the safe memory region and update the shared memory before program execution resumes. Consider each cache block residing in the safe memory region. Instances of these cache blocks may exist in:

1. one other processor in the CSM state or
2. one or more other processors in the CSS or invalid state

The recovery machinery needs to flush all instances of these cache blocks to memory and make sure that each byte in the blocks holds the latest value from the last write to this byte. To do this correctly, the machinery consults the two access bit vectors, Write This (WT) and Write Others (WO), which are associated with each cache block in the safe memory region.

Given that there is no race, it is not possible for a particular byte to have both its WT and WO bit set at the time the recovery takes place. If the WT bit is set, the update to the shared memory needs to come from the associated byte in the block residing in the safe memory. This holds regardless of whether there is a copy of this block in other processor cores since the other cores never wrote to this byte.

If the WO bit is set, the state bits need to be consulted. If the block is in the CSS or invalid state, no further update of this byte is required, because the memory already contains the latest copy. If the block is in the CSM state, the update for this byte must come from the core that has this block in the CSM state. Note that the fallback mechanism described also correctly handles cases involving condition variables. Recall that all the cores with waiting threads have their safe memory synchronized with one another. Of course, after having completed these recovery operations, all relevant bookkeeping structures such as the Active Bit and the ABVs need to be reset as well.

7.3.7 Tolerating the RrwW Race with Reexecution

So far, the race case RrwW has been considered intolerable. However, this case can be serialized as rwRW if we have the ability to reexecute the RW operations in the safe thread after we have seen the intervening rw operations from the non-safe thread. With hardware ToleRace, such reexecution becomes viable. This sub-section quantifies the potential to increase race tolerance through critical section reexecution on the benchmark programs. Note that reexecution capability can also be provided in Pin-ToleRace through PIN_SaveCheckpoint and PIN_Resume APIs, but is limited to only lightweight reexecution. I will explain what lightweight reexecution refers to below.

First, consider the reexecution capability of a particular critical section execution. A critical section execution is considered to be in one of the three categories, non-reexecutable, lightweight reexecutable, and heavyweight reexecutable. A non-reexecutable critical section always involves interactive I/O operations that cannot be undone and cannot be repeated. A critical section in this category executes calls to library functions that write to output devices or take input from users interactively. Example of such library functions are printf, getc, and gets. A lightweight reexecutable critical section does not contain calls to library functions that invoke calls to operating system services. Therefore, it can be reexecuted purely by checkpointing the register and memory reads in the critical sections in user code and address space. A heavyweight reexecutable critical section contains calls that invoke operating systems services that can potentially be undone by checkpointing the full memory states in both user and OS address spaces. These include calls to threading and synchronization operations in pthread libraries.

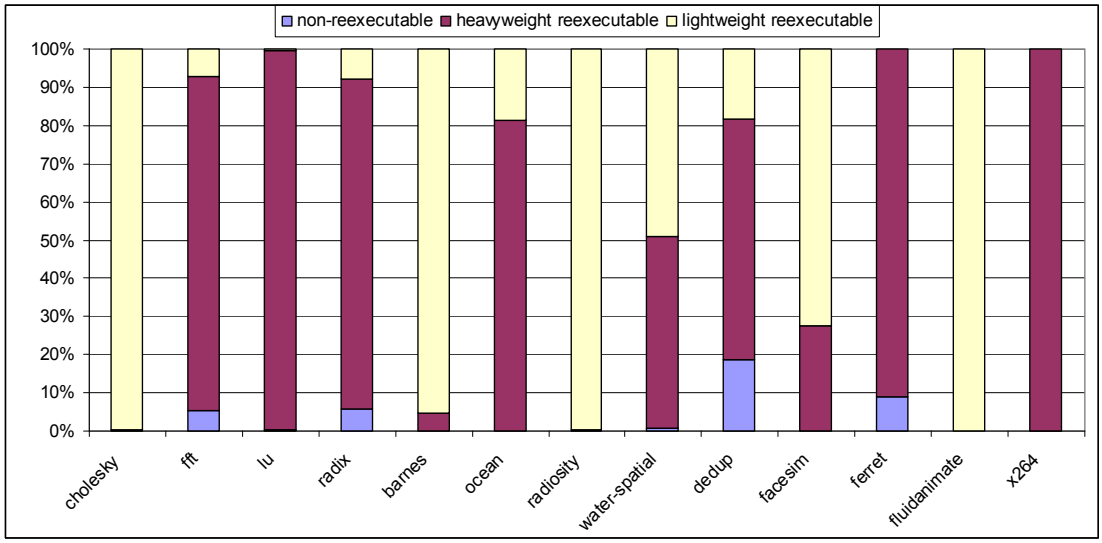


Figure 26: Critical section's reexecutability for each of the benchmark programs.

Figure 25 shows the breakdown of each critical section type for all the benchmark applications. All critical section executions are considered not just those with RW operations. To obtain the results in Figure 25, I wrote a Pin tool to track down all function calls generated when a critical section executes and categorized those calls to determine the type of the critical section in terms of its reexecutability. We can see a spectrum of characteristics here. cholesky, barnes, radiosity, and fluidanimate contains mostly lightweight reexecutable critical sections. On the other hand, heavyweight reexecutable critical sections are dominant in fft, lu, radix, ocean, dedup, ferret, and x264. water-spatial has an equal mix of the two categories. Non-reexecutable critical sections, although not prevalent, are visible in fft, radix, dedup, and ferret.

The next experiment measures the race toleration potential in each benchmark application. The results of this experiment when all critical section executions are considered are shown in Figure 26. The toleration potential is measured in fraction of the total memory accesses inside of critical sections. A 50% toleration potential means

that, on average, for a given memory location accessed inside of critical sections, half of the accesses are potentially tolerant to races, i.e., they can be serialized with the racing accesses.

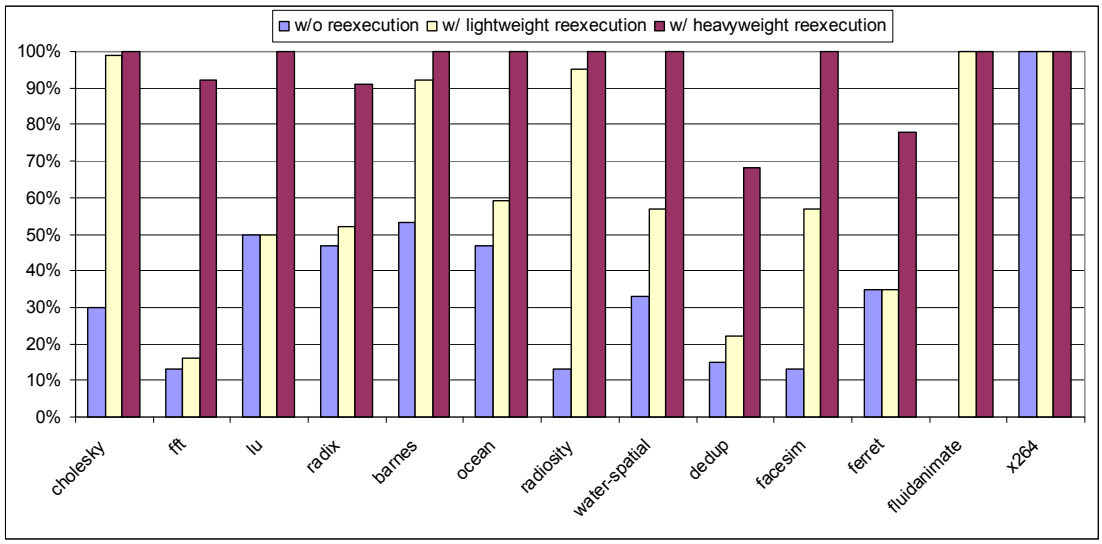


Figure 27: Race toleration potential for each benchmark application.

The first bar in Figure 26 for each application shows the race toleration potential without any reexecution. In essence, this bar indicates fraction of all accesses to a given shared memory inside of critical sections that do not start with a read that is later followed by a write (which is indicative of the RW operations). Recall that when this is the case, it guarantees that race case RrwW never happens, and, hence, all the racing accesses from other threads to this memory can always be tolerated. We can see that we have two extreme cases. fluidanimate has zero toleration potential whereas x264 has 100%. barnes hovers just above 50% whereas most of the benchmarks have less than 50% toleration potential. When considering lightweight rexecution, the toleration potential shoots up dramatically in a number of applications, particularly in

fluidanimate where it jumps from 0% to 100%. Other benchmarks with large improvements are cholesky, barnes, radiosity, and facesim. Note that the results shown here are compatible with the categories of critical sections we saw in Figure 25. When considering heavyweight reexecution, we see 100% toleration potential in all except those benchmark applications that contain non-reexecutable critical sections.

7.3.8 Context Switches

The discussion of the proposed hardware ToleRace in the multicore framework thus far has ignored the effect of context switching. After all, an application may create more threads than there are cores available. Therefore, any operating system that has some fairness policy in its scheduler will need to perform context switching to suspend some running threads and let other waiting threads run. Also, to better utilize the CPU cores, the OS will often suspend a thread that is idling waiting for a response from an I/O device and switch in a ready thread. Moreover, even if the application does generate as many threads as there are cores, the OS may still have reason to perform context switching. For example, it may want to perform thread migration, e.g., to switch a power-hungry thread from a hot core to a cooler core to prevent thermal run-away. Therefore, we must be prepared for the possibility that context switching happens while a thread executes inside a critical section. For our 13 benchmark applications, we expect those with long critical sections such as dedup and ferret to be more vulnerable to this scenario than others with short critical section, e.g., ocean and fluidanimate. In addition, as we have seen from Figure 25 in the previous sub-section, dedup and ferret execute I/O-calls in critical sections. This makes them even more susceptible to undergo context switching while inside of critical sections.

Because the safe memory is part of the thread context, to survive context switching without losing the ToleRace capabilities, we need to ensure that this context is kept

even though the actual thread that owns it is suspended, i.e., switched out of the running core. Here I explore a number of possibilities for surviving context switching.

First, assume that there are no condition variables involved. In this case, a number of simple solutions is available. The simplest solution is to delay context switching until the running thread exits the critical section. For applications with relatively short critical sections, this approach should be efficient enough. To cope with long running critical sections that contain I/O operations, the hardware ToleRace may allow context switches to happen inside of critical sections. However, it may elect to leave the safe memory context of the switched-out thread in the core, speculating that the new running thread will not overwrite the safe memory context of the previously running thread, i.e., the new thread will not be in any critical sections before the previous thread resumes. To do this successfully requires close co-ordination with the OS's scheduler to ensure that the previously suspended thread that owns the safe memory context is switched back into the same core. If all goes well with this scheme and the speculation is successful, the context switching overhead will be minimal since there is no saving and restoring of the safe memory context. This looks just like any regular context switches with no safe memory context involved.

If the new thread enters a critical section and is thus about to overwrite the existing safe memory context, the hardware ToleRace has the following options. First, it may trigger an interrupt to ask the OS for help. For example, it can try to suspend the current thread and resume the previous thread, thus allowing the previous thread to continue with its safe memory context until it exits from the critical section. Hopefully, by the time the previous thread resumes, it has received all the responses it was waiting for while being suspended. A simpler alternative is for the hardware ToleRace to simply stop the protection of the previous thread. This also requires co-operation from the OS to clear the old safe memory context before allowing the new

context in.

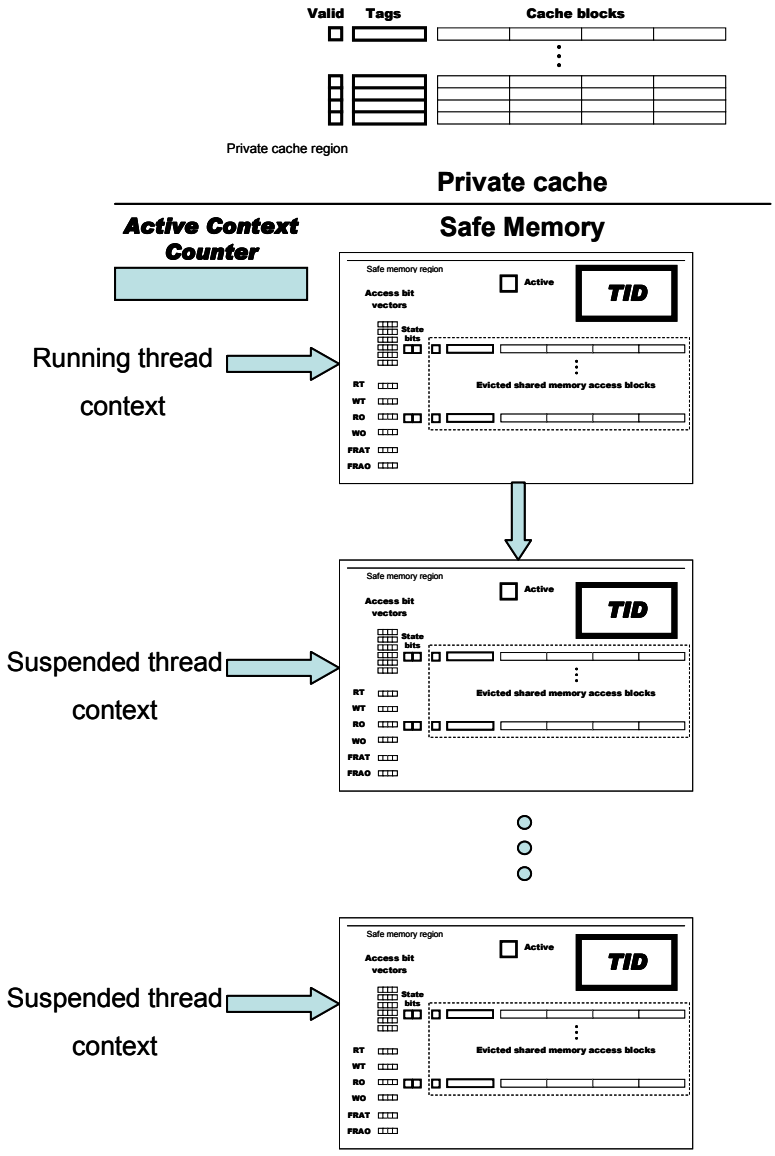


Figure 28: Organization of the safe memory to survive context switching.

Now, consider dealing with critical sections with condition variables. This situation is much harder to cope with. All the simple solutions just explored (except abandoning ToleRace protection) will not work here. The hardware ToleRace simply cannot disallow context switching inside of critical sections or the system faces a potential deadlock situation. This is because every thread running on each core might be

waiting on a condition variable, and the only way out is to switch out some of the running threads and let a thread that generates the conditional signal in to break the "deadlock". Switching out a thread that has a condition variable safe memory context is problematic because the hardware ToleRace will need to ensure that the suspended safe memory context is kept consistent with all the other safe memories.

One hardware intensive solution is to keep the safe memory in each core as a chain in a linked list as shown in Figure 27. The safe memory is augmented with a field that records the thread ID that owns the safe memory context. When a running thread is switched out while inside of a critical section, the corresponding core clears the current context to make room for the next thread, increments the Active Context Counter (ACC), and adds the suspended safe memory context to the safe memory list. When the ACC value is greater than one, even if the current core does not have a current safe memory context, i.e., it does not execute inside of a critical section, it is obliged to snoop on the bus on behalf of the suspended contexts. Organizing the safe memory in this linked list fashion allows for the associative search to start from the current context to all the suspended contexts in a particular core. When a suspended thread is ready to resume running, it needs to clear off the suspended context recorded in the core on which it previously ran. The OS needs to work with the hardware ToleRace to make this successful. Upon resuming the suspended thread, an invalidation signal containing the thread ID needs to be broadcast and intercepted by this core. The core then proceeds to delete the corresponding suspended context from the list, copies this context to the new core, and decrements the ACC accordingly.

Another possible solution is to have a centralized shared buffer that keeps the suspended safe memory context within. This buffer is connected to the common bus and performs all the necessary snooping operations to update the appropriate data structures in the safe memory as described in the Section 7.3.1. With this centralized

structure, we trade off extra cost of having a dedicated structure with faster context switching. However, as the number of threads and their corresponding contexts increases, such a centralized scheme will have poor scalability.

With the two proposed hardware supports for context switches I just described, there is always the possibility that the hardware resources are exhausted. When this happens, the hardware ToleRace, in co-ordination with the OS, has to resort to some recovery mechanism and stop supporting the ToleRace functionality.

7.4 Enabling ToleRace with Hardware Transactional Memory

The proposed hardware ToleRace bears some resemblance with hardware transactional memory. Given the ubiquity of multicore processors and the TM promise to enhance parallel applications developed for such processors, it is expected that manufacturers will soon start to ship processors with TM supports. Sun Microsystems is one such manufacturer who has announced publicly to have prototyped a processor with such support [16]. Given the arrival of hardware TM, it seems expedient to fit the ToleRace mechanism into this framework. This section investigates how to accommodate the ToleRace functionalities with hardware TM. It lists some necessary conditions for ToleRace to be incorporated correctly in hardware TM. Because doing so requires a minimal increase in hardware budget and complexity, processor manufacturers may be inclined to support both ToleRace and TM at the same time with the former geared towards enhancing reliability of existing lock-based programs and the latter towards newly developed transaction-based parallel applications.

As ToleRace is designed to work transparently with existing lock-based program binaries, in the first step, assume that there is a pre-processing system to “transactify” lock-based programs to transaction-based programs where lock-based critical sections

are transformed into atomic blocks marked by transaction constructs recognized by the hardware TM. Note that essentially all this system does is replace lock-based constructs with some constructs recognized by the TM hardware. Nothing about the ToleRace or TM semantics are incorporated at this point. An example of such a system is HyTM [15]. The following are some necessary conditions for ToleRace to function on top of hardware TM.

1. Use deferred (a.k.a. lazy) update

To be compatible with ToleRace, the hardware TM must not modify the shared data directly. Most proposed hardware TMs that buffer updates in private caches satisfy this requirement. However, there exists hardware TM systems such as Wisconsin's LogTM [36] that do not satisfy this requirement. LogTM uses an eager update protocol that is not compatible with ToleRace.

2. Support open-nested semantics

To accommodate nested critical sections and follow their lock-based program semantics faithfully, the hardware TM needs to provide support for open-nested semantics. Simple flattening is not sufficient to embrace nested critical sections as it may prevent forward progress in the original lock-based program. To make this more concrete, consider the following somewhat contrived example.

```
P = 4;
Lock(mutex_X)
Q++
Lock(mutex_Y)
P++
Unlock(mutex_Y)
do {} while (P < 5);
Unlock(mutex_X)
```

In this example, the lock variable mutex_X protects the update to the variable Q whereas mutex_Y protects the update to the variable P. The mutex_Y critical section is nested in the outer mutex_X critical section and, just before exiting from the outer critical section, there is a do-while loop that spins on the condition (P < 5). If the underlying hardware TM uses a deferred update protocol and flattens nested-transactions, when the example above is "transactified", it will keep spinning on the do-while loop after executing the inner critical section as the updated value of the shared variable P will not be made visible until the outer critical section completes.

Unfortunately, recently proposed hardware TM handles nested transactions by flattening them. Although there has been exploration in open-nested transactions [6], flattened transactions remain the preferred semantics for nested transactions because of their relative simplicity. Therefore, for the current generation of TM hardware to accommodate nested critical sections like ToleRace does, it may need assistance from special hardware like what depicted in Section 7.3.3. This hardware is rather intuitive and not particularly complicated, so I believe including it to support nested critical transactions is justifiable.

3. Disable concurrent transactions

Critical sections that are protected by the same mutex variable are mutually exclusive. Hence, for ToleRace, there is no notion of concurrent execution of such critical sections. To enforce this condition, the hardware TM must disallow concurrent transactions that originate from the same mutex variable. Disabling a given transaction should not be hard to accomplish as the TM hardware already contains structures to track a new transaction that is about to execute in the middle of an outstanding transaction. However, recognizing which transaction to disable requires special treatment. During the "transactification", an atomic block derived from a corresponding critical section needs to be augmented with the mutex variable that is associated with the critical section. When the TM hardware processes a given atomic block, it examines if the atomic block is a generic one, i.e., coming from purely transaction-based code, or a derived one, i.e., coming from lock-based code. It imposes no restriction on concurrent execution for generic atomic blocks, but prohibits concurrent execution of derived atomic blocks with the same mutex variable.

4. Detect conflicts from non-transactions executions

If the hardware TM is only concerned with conflicts among transactions, it needs to be augmented to detect conflicts from non-transaction execution in order to support ToleRace. The contention management hardware is not needed when in ToleRace mode because ToleRace always guarantees forward progress. However, additional resolution hardware is needed. This is a simple piece of hardware whose primary function is to propagate memory writes based on the type of conflict detected.

5. Support Retry/OrElse constructs

To mimic condition variable semantics, the hardware TM must support coordination

among transactions in form of Retry/OrElse constructs [22]. These constructs can be thought of as a generalized form of signal/wait operations where the transaction and the shared location names are not given explicitly.

7.5 Summary

This chapter describes a hardware implementation of ToleRace that leverages components in multicore processors. The proposed design augments the private cache in each core with regular structures that are reminiscent to those of a victim cache and requires only minor modification to the standard MSI cache coherence protocol. From the quantitative assessment using the benchmark applications, only a small amount of additional hardware is needed to support ToleRace functionality. With hardware support, tolerating the race case RrwW that would otherwise be difficult to do in software ToleRace becomes viable with reexecution.

CHAPTER 8

CONCLUSION

This dissertation introduces ToleRace, a novel runtime system that uses data replication for detecting and tolerating concurrency errors in lock-based multithreaded programs. ToleRace addresses asymmetric races, where one use of a shared variable is correctly protected with locks while other uses are not. This dissertation first presents a theoretical framework and explains why asymmetric races are the focus of this work. It then describes two software implementations on top of a dynamic instrumentation tools. The evaluation indicates that real applications can run on top of software ToleRace with acceptable overhead, about 2X slowdown. For reference, the third software implementation that mimics the effect of the oracle compiler is included. This version incurs only 6.4% overhead across the benchmark applications. Finally, this work proposes a hardware implementation that leverages components in multicore processors. The initial evaluation shows that only a small amount of additional hardware is needed to support ToleRace functionality and such hardware possesses a regular structure that is similar to that of a victim cache. This should be familiar to CPU hardware designer and make adoption of ToleRace easier.

CHAPTER 9

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 [39]) or defines new programming language semantics that help improve the static detection of races (e.g., Cyclone [20]). Static analysis techniques face several challenges. First, because many of the techniques are based on some form of model checking [23], 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 [18] and Houdini/rcc [19] address these issues by combining traditional static analysis with heuristics and statistical ranking to identify the most probable races. One inherent drawback of static analysis for race detection is that asymmetric races can occur in contexts where the source code for the component containing the error is not available for examination.

Eraser is a dynamic race detection system based on lock-sets [47]. 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 [3]. Combining locksets with a happens-before scheme results in higher precision dynamic race detectors [12, 17, 41, 50]. Even with refinements, the execution overhead of these approaches is typically larger than a factor of two. Recent work that affords a reasonable overhead proposes to implement the happens-before algorithm purely in hardware [38].

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 [26] and Sun's Thread Analyzer [25], which are commercial tools capable of locating data races in concurrent programs. Both tools suffer from a high memory footprint and runtime overhead and are, thus, primarily used for software testing.

Atomicity violation is another important class of concurrency errors. It can be addressed statically [5] or dynamically. The AVIO system [33] 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. [34] offers to tolerate some degree of atomicity violation with implicit atomicity by grouping consecutive memory operations into atomic blocks.

Vaziri et al. [48] classify harmful interleavings into 11 categories, which is more than the six race cases (with case IV subdivided) this work considered (Chapter 2). The extra categories address high-level data races at the object granularity, which this work does not consider. Their approach to race detection requires source-code annotation and targets safe language environments.

Kiena et al. [29] 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 this work, Rajamani et al. [42] propose a run-time 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.

The hardware ToleRace proposed is analogous to hardware transactional memory, which was first proposed by Herlihy and Moss [24]. The proposed scheme works on a fully associative transaction cache and leverages the cache coherence protocol to detect conflicts. Rajwar and Goodman proposed Speculative Lock Elision (SLE) [43], a form of optimistic synchronization that elides locks in parallel programs and, thus, allows multiple critical sections that were mutually exclusive to execute concurrently. Their follow-up work, Transactional Lock Removal (TLR) [45], extends SLE to address issues with starvation as well as transactional semantics. The advent of multicores around the year 2004 reinvigorated research in hardware transactional memory. Transactional Coherence and Consistency (TCC) from Stanford [21] is a hardware TM scheme that provides strong isolation and employs deferred policies for both update operations and conflict detections. In contrast to TCC, LogTM [36] from Wisconsin uses eager update and eager conflict detection protocols. LogTM's authors argue that commits happen more often than aborts so adopting all eager policies make the common cases fast. Ceze et al. [13] describes a hardware TM that does not rely on cache coherence to track and detect conflicting memory accesses. All the hardware TM schemes described thus far are bounded hardware TM. Cases for unbounded TMs have been described by Ananian et al. [7] and Rajwar et al. [44].

REFERENCES

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

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

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

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

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

[6] K. Agrawal, A. I. T. Lee and J. Sukha, *Safe open-nested transactions through ownership*, *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, Raleigh, NC, USA, 2009.

[7] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson and S. Lie, *Unbounded Transactional Memory*, *11th International Symposium on High-*

Performance Computer Architecture (HPCA'05), San Francisco, CA, 2005, pp. 316-327.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] L. Ceze, J. Tuck, J. Torrellas and C. Cascaval, *Bulk Disambiguation of Speculative Threads in Multiprocessors*, *Proceedings of the 33rd annual international symposium on Computer Architecture*, IEEE Computer Society, 2006.

[14] D. Culler, J. P. Singh and A. Gupta, eds., *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 1998.

[15] P. Damron, A. Fedorova and Y. Lev, *Hybrid transactional memory*, *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. 336-346.

[16] D. Dice, Y. Lev, M. Moir and D. Nussbaum, *Early experience with a commercial hardware transactional memory implementation*, *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ACM, Washington, DC, USA, 2009.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis and K. Olukotun, *Transactional Memory Coherence and Consistency*, *Proceedings of the 31st annual international symposium on Computer architecture*, IEEE Computer Society, Munich, Germany, 2004.

[22] T. Harris, S. Marlow, S. Peyton-Jones and M. Herlihy, *Composable memory transactions*, *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, Chicago, IL, USA,

2005.

[23] 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.

[24] 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.

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

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

[27] G. Hunt and D. Brubacher, *Detours: Binary Interception of Win32 Functions*, *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, 1999.

[28] N. Jouppi, *Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers*, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, Washington, 1990.

[29] 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.

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

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

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

[33] 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.

[34] 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.

[35] 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.

[36] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill and D. A. Wood, *LogTM: log-based transactional memory*, *The Twelfth International Symposium on High-Performance Computer Architecture (HPCA'06)*, Austin, TX, 2006, pp. 254- 265.

[37] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar and I. Neamtiu, *Finding and reproducing heisenbugs in concurrent programs*, *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, 2008.

[38] A. Muzahid, D. Surez, S. Qi and J. Torrellas, *SigRace: signature-based data*

race detection, *Proceedings of the 36th annual international symposium on Computer architecture*, ACM, Austin, TX, USA, 2009.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] R. Rajwar and J. R. Goodman, *Speculative lock elision: enabling highly concurrent multithreaded execution*, *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, Austin, Texas, 2001.

[44] R. Rajwar, M. Herlihy and K. Lai, *Virtualizing Transactional Memory*, *Proceedings of the 32nd annual international symposium on Computer Architecture*, IEEE Computer Society, 2005.

[45] R. Rajwar and J. R. R. Goodman, *Transactional lock-free execution of lock-*

based programs, *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ACM, San Jose, California, 2002.

[46] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal and K. Pattabiraman, *Detecting and tolerating asymmetric races*, *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, Raleigh, NC, USA, 2009.

[47] 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.

[48] 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.

[49] 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.

[50] 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.