Detection of and Recovery from Concurrency Errors Using Transactional Memory Techniques

by

Hassan Salehe Matar

A Thesis Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

Koç University

September, 2013

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Hassan Salehe Matar

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

Assoc. Prof. Serdar Taşıran (Advisor)

Prof. Dr. Attila Gürsoy

Prof. Dr. A. Murat Tekalp

Date:

To Ibrahim and my family at large,

ABSTRACT

We propose a technique to improve detection of concurrency errors of multi-threaded C/C++ applications and recovery of these applications from the errors using transactional memory (TM) technology. Transactional memory is an emerging parallel programming model which simplifies parallel program design and implementation, improves performance and protects applications from most concurrency bugs. Our approach uses TM to improve performance of detection of concurrency errors and provides a framework by which legacy C/C++ applications can benefit from concurrency error-freedom.

The current concurrent error detection approaches are either too slow or need extra modification to current processor architecture. The slowdown of these techniques stems from a number of reasons: instrumentation used to add them into application, necessary computations needed to detect errors, and cost of proper protection of error-detection-related data used by these techniques.

We propose a way to divide the instruction stream of each thread in a multi-threaded application into small transactions. We then use conflict detection to get fine-grain protection of concurrency error detection data to improve performance. We use transaction write buffers and rollback mechanism to recover from concurrency errors, data races in particular, and impose extra protection on erroneous data or portions of concurrent program.

Our approach works well on a number of multi-core benchmark applications and shows a significant performance improvement of concurrent error detection over conventional means. These improvements are encouraging initial results for the industrial usage of the proposed approach.

ÖZETÇE

Bu çalışmada, çok izlekli C/C++ uygulamalarındaki eş zamanlı programlama hatalarının tespitini geliştirmek ve bu uygulamaların hareket belleği (TM) teknolojisi kullanılarak kurtarılmasını sağlamak için bir yöntem öne sürüyoruz. Hareket Belleği, koşut program tasarımını ve uygulamasını basitleştiren, başarımı arttıran ve uygulamaları çoğu eş zamanlı programlama hatalarından koruyan bir koşut programlama modelidir. Bizim yaklaşımımız, eş zamanlı programlama hatalarının tespit edilme başarımını arttırmak için TM kullanmakta ve kalıt C/C++ uygulamalarının eş zamanlı programlama hatasızlıktan faydalanabileceği bir çerçeve sağlamaktadır.

Mevcut eş zamanlı hatası tespit eden yaklaşımlar ya çok yavaş ya da mevcut işlemci mimarisinde fazladan değişikliklere ihtiyaç duymaktadırlar. Bu yöntemlerin uygulamayı yavaşlatması birkaç sebepten kaynaklanabilir: bunları uygulamaya eklemek için kullanılan araçlar, hataların tespiti için gerekli hesaplamalar ve bu teknikler tarafından kullanılan hata tespitiyle alakalı verinin uygun biçimde korunması.

Çok izlekli bir uygulamadaki her izleğin komut akışını küçük hareketlere bölmek için bir yol sunmaktayız. Daha sonra, eş zamanlı hata tespit verisinin iyi taneli korunmasını elde etmek için çakışma tespitini kullanmaktayız. Eş zamanlı hatalarından, özel olarak veri yarışlarından, kurtulmak için hareket yazma arabelleklerini ve geri dönüş mekanizmalarını kullanmaktayız ve hatalı veriye veya eş zamanlı programın parçalarına fazladan koruma dayatmaktayız.

Yaklaşımımız birçok çok çekirdekli denektaşı uygulamada iyi çalışmakta ve uzlaşılmış yollara nazaran eşzamanlı hata tespitinde belirgin bir başarım artışı göstermektedir. Bu gelişmeler önerilen yaklaşımın endüstride kullanımı için cesaret verici öncül sonuçlardır.

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor *Assoc. Prof. Serdar Taşıran*. His experience, understanding and patience added considerably much to my graduate experience, and his invaluable comments and advices helped me in adapting academical research. I would also like to take this opportunity to thank my thesis committee.

My sincere thanks go to İsmail Kuru and Gokcen Kestor, the co-authors of the earlier work of my project, for their collaboration. I thank Roman Dementiev at Intel for his support on later stages of my project. I also appreciate anyone who contributed to my research in one form or another. I would like to thank Intel for pre-release equipment donations which fostered this work. I give special thanks to Microsoft® Research and TÜBİTAK as they sponsored this work.

I am grateful to former and present members of Multicore Software Engineering Research Center: Ömer, Can, Burcu, Erdal,İsmail, and Suha. I can not forget my friends once co-located in the center: Ayşe Nur, İdil, Özge and Salih.

I acknowledge anyone who contributed to the preparation of this thesis. Especially I appreciate Erdal Mutlu, Burcu Külahçioğlu Özkan, Suha Mutluergil and Can Bekar for valuable feedback and Turkish translation.

I would like to express my deepest gratitude to my wife and my whole family for patience throughout my research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

CPU	Central Processing Unit
RTM	Restricted Transactional Memory
STM^2	Software Transactional Memory for Simultaneous Multithreading
TSX	Intel(r) Transactional Synchronization Extensions

Chapter 1

INTRODUCTION

1.1 Motivation

Multi-core CPUs are everywhere; from supercomputers to servers, from desktops to smart phones. This has naturally forced software developers to write parallel and concurrent programs to benefit from parallel performance offered by the multi-cores. As is the case for any software it is almost impossible to write entirely correct software which meets specifications and adheres safely to the programming model of the machines in which that software is intended to run at first place. Therefore, there is every need for means to ensure the software behaves as expected and delivers the required and intended actions. There has been a number of techniques to ensure correctness of software developed. Some of them involve testing, model checking, and verification. As parallel programs involve execution of multiple threads simultaneously it is very hard to eliminate most of the program errors using classical testing approaches.

Appropriate ways of debugging parallel program are model checking and verification. This is because they put program reasoning into mathematical form which takes into account the general behavior of execution of the concurrent program. Verification reasons about program correctness from program input and output specifications and expected behaviors of the program and ensures that these properties are attained by the program when executed in a multi-threaded fashion. There are two types of verification approaches; static and dynamic. Static verification analyses program source code to determine if it conforms to certain specifications and parallel programming semantics. Dynamic verification monitors the software running on multi-threaded environment in multi-core machines. Therefore, it checks the program correctness from the program execution.

Static verification is complicated and very slow and may be infeasible to perform on large code base. Moreover, it has tendency to emit a lot of false alarms. Dynamic specification is in most of the time a preferred way of debugging multi-core software. Even though dynamic verification techniques may be a preferred way for correctness checking of concurrency errors, their current technology is still too slow to be used as debuggers running all the time during software development phase. There has been a number of researches working towards improving dynamic race detection. Some of them propose use of dedicated hardware component [31], introduction of new hardware instructions [5] special for dynamic software checking and use of heterogeneous multicores [1]. Others improve on existing approaches and form new algorithms [6].

While others are trying to improve performance of verification techniques, some have proposed parallel programming models that promise concurrent error-freedom and less hassle for design and implementation of parallel software. One of the notable models is transactional memory [10]; an idea which originated from database transaction semantics. This approach abstracts away all complexities of multi-core software and it provides a nice and simple way of writing software for multi-cores. Programmers specify regions they want them to execute in a transactional way. Transactions ensure atomicity; all operations in a transaction seem to execute instantaneously and their effects are seen by other threads when they are all successful. It also provides a simple reasoning of parallel programs by ensuring serializability. Serializability is a property where transactions executed by different program threads in concurrent fashion appear to produce results that are same to when they were executed one after another in some sequence. There are two types of transactional memory; software transactional memory and hardware transactional memory. Software transactional memory model is software implemented as a library or runtime and this software ensures transactional semantics are realized when software runs atop it. The library or runtime provides interfaces by which developers use to implement transactional memory-based concurrent software. On the other hand, hardware transactional memory is the one supported by or implemented in hardware. It involves modification to processors, caches and bus protocols [11]. Hardware transaction memory guarantees more improved performance than software transactional memory.

Transactional memory is not the best alternative for concurrency programming in all cases. Moreover, there are a lot of non-transactional memory software already running on various systems and it may be near to impossible to re-implement all them with transactional memory semantics. Some software developers are skeptical about programming using transactional memory. Therefore, there is still a need to improve dynamic verification techniques for current software development and running software that may need correctness checking. This thesis concentrates on runtime race detection, a common type of dynamic software verification.

1.2 Approach

This thesis proposes solutions for legacy software, and for non-transactional-software development. For legacy software we provide a way to ensure that it executes in a correct way as intended by developers and the system specification it is running on. Moreover, for both legacy software and for development concurrent software we provide a technique to improve dynamic verification.

For legacy C/C++ programs we propose [16] a way to divide program trace into transactional regions to execute as transactions. We present the PaRV tool for runtime detection of and recovery from data races in multi-threaded C and C++ programs. PaRV uses transactional memory technology [14] for parallelizing runtime verification and for buffering write accesses during race checking. Application threads are slowed down only due to instrumentation, but not due to the computation performed by runtime verification algorithms since the latter are run concurrently on different threads. Buffering writes allows us to recover from races and to safeguard against later ones.

For reducing slowdown of dynamic detection, we propose use of Intel(r) Transactional Synchronization Extensions (TSX). By dividing program traces for each thread and adding transaction instructions and calling race detection Performance is achieved due to reduce slowdown of the synchronization operations necessary to protect race detection meta-data.

1.3 Contribution

This thesis has an number of contributions.

We propose a way to reduce runtime race detection using helper thread per application thread adapted from STM^2 [14]. Moreover, using software transactional memory techniques recovery from the races can be achieved and prevented further by imposing extra protection on the racy data.

We propose an algorithm to instrument a program into small transactions for race detection. Splitting a program into small transactions does not change semantics of the program and it does not affect precision of race detection.

We propose use of TSXs to aid race detection and show that it improves performance. We don't simply protect the race-detection metadata using transactions. We break down a binary into proper-sized chunks in order to minimize overhead. We present a proper use of of TSXs to achieve precise race detection with much reduced slowdown while maintaining soundness and precision of race detection algorithm used.

1.4 Organization

Chaper 2 and 3 discuss necessary background. While chapter 2 presents technical details necessary to understand race detection, chapter 3 introduces transactional memory. Data race detection and recovery using software transactional memory is discussed in chapter 4. Improvement of race detection using hardware transactional memory is presented on chapter 5. Conclusions and related works are in chapter 6 and chapter **??**, respectively.

Chapter 2

CONCURRENCY ERRORS AND RUN-TIME RACE DETECTION

2.1 Concurrency Errors

Concurrent errors are run-time errors introduced due to presence of multiple processes, or preferably threads, executing in concurrent fashion and accessing shared resources like data structures and synchronization primitives such as locks. The commonly known concurrency errors are *deadlocks*, *data races*, *atomicity violations*, and *ordering violations*. A *deadlock* is a situation wherein two or more concurrent threads fail to proceed their executions because each waits for the other(s) to do something. A common example of a deadlock can be explained with with two threads A and B and two locks X and Y acquired by the threads, respectively. While holding these locks, the deadlock happens when A waits for lock Y to be released by B and B waits the lock X to be released by A.

Data races or race conditions occur when two different threads access the same memory location, at least one access is a write, and the accesses are not ordered by proper synchronization. There are many scenarios for data races. A typical example is when a shared memory is not protected by a common lock all the times it is accessed by any thread. Figure 2.1 shows a race condition due to use of different locks for different accesses on two threads for the same address.

In *Atomicity violations* the desired serializability among multiple memory accesses is violated. This happens when a program region intended to execute indivisibly in atomic way is not enforced during execution. This results from a lack of constraints on the interleaving of operations in a multithreaded program.

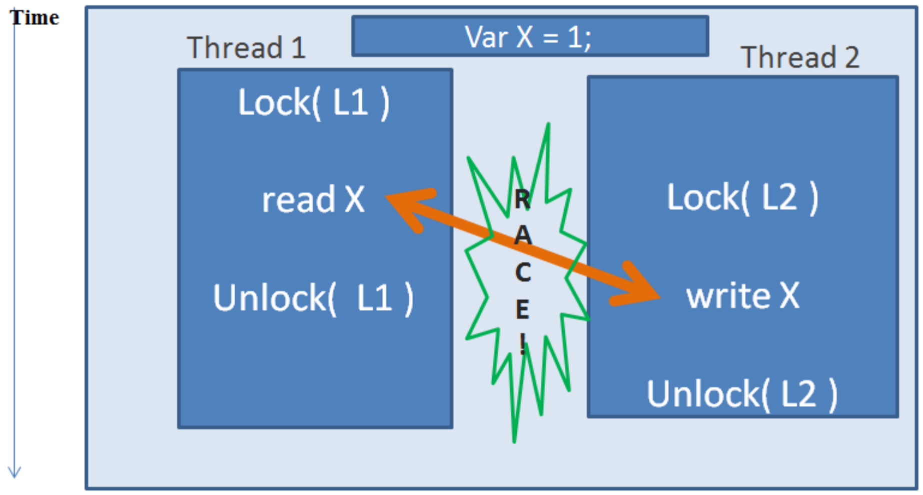

Figure 2.1: Example of race condition.

Ordering violation occurs when the desired order between two or more memory accesses is flipped. This involves two or more memory accesses from multiple threads that happen in an unexpected order, due to absent or incorrect synchronization. For example given an order that access X should always be executed before Y. The violation happens when the order is not enforced during execution in such a way that Y executes before X. This happens on sequential consistency memory models due to optimization purposes. On concurrent execution environment this could lead to non-determinism.

The analysis [19] of real-world concurrency bugs presents statistics about these errors on concurrent software. These bugs are very common in multithreaded programs. They are a major stumbling block to writing multithreaded programs as they generate complicated behaviors resulting from unexpected interaction of operations in different threads. Therefore, they are particularly difficult to diagnose and fix. This is due two main reasons. First, developers must reason about the interactions of many pieces of code executing in multiple threads. Observing one threads erroneous behavior is often insufficient for understanding the cause of that error, which may lie in another thread. Second, non-determinism in multithreaded execution complicates the process of interpreting executions that contain concurrency errors. The exact behavior of the application may differ, even from one erroneous program run to another, making it hard to spot the main source of the error. Therefore, it is important to study these errors and devise techniques to automatically detect them.

Our thesis concentrates on the study of data race condition detection for two reasons;

(a) Data race conditions are low level errors which violate program sequential consistency,

(b) They are symptoms of higher-level errors like atomicity and order violations.

2.2 Program Executions

To understand runtime race detection it is important to grasp knowledge of multithreaded program executions [26]. Rather than a completely general formalization that models all features of a multithreaded program, we opt for one that is simple and allows us to illustrate the key ideas for each runtime verification approach. We do not provide a formal syntax and semantics for multithreaded programs. Instead, only reason in terms of their executions is presented.

A multithreaded program consists of a number of concurrently executing threads, each identified with unique thread identifier `T` $\in$ `TID`.

The set of possible actions that a thread `T` can perform include:

- `rd(T, x, v)` and `wr(T, x, v)`, which read a value `v` from variable `x` and write a value `v` to `x`, respectively. The possible effects of the read and write actions on the local and global stores are given by the language specification and memory model and are left unspecified in this paper.

- `acq(T, k)` and `rel(T, k)`, which acquire and release a lock `k`.

- `begin(T, l, s, i )` and `end(T, l, s, i )`, which mark the beginning and end of one particular execution instance of a designated code block consisting of the program statement `s`. It might be the programmers intention to have this block be atomic, serializable, etc. The block label `l` uniquely identifies a designated code block, and the instance label `i` uniquely identifies the execution instance of the designated code block within the entire execution trace.

- `fork(T, T')` and `join(T, T')`, which create a new thread `T'` and wait until a thread `T'` terminates, respectively.

- `call(T,m, i )` and `ret(T,m, i )`, which represent the call and return actions of a particular execution of operation `m`. These actions are thread-local, i.e., they do not modify the global store. The instance label `i` uniquely identifies a particular execution of the operation `m` within the entire execution trace.

In code examples, we often omit the thread ids, block and instance labels when they are clear from the context or irrelevant, and we use more familiar syntax, such as `x = v`, for reads and writes. We use the function `TID(a)` to extract the thread identifier from an action.

A *trace* is a finite sequence of actions α = a_1, a_2, ..., a_n. A particular occurrence of an action (i, a_i) in a trace is called an *event*. The behavior of a trace α = a_1, a_2, ..., a_n is defined by the relation $\sum_0 \rightarrow^{\alpha} \sum_n$, which holds if there exist intermediate states $\sum_1$,..., $\sum_{n-1}$ such that $\sum_0 \rightarrow^{a_1} \sum_1 \rightarrow^{a_2} \ldots \rightarrow^{a_n} \sum_n$.

Event e_i = (i, a_i) is referred to using the type of action a_i is, e.g., a write or read event. The write predecessor of a read event `ej` where a_j = `rd(T, x, v)` in a trace is the largest `k` such that e_k is a write event that writes to `x`.

An *intended-atomic block* in a trace α is the sequence of actions starting with a `begin(T, l, s, i )` action and containing all `T`-actions up to and including a matching `end(T, l, s, i )` action. If an action `a` by thread `T` does not occur within an intended-atomic block for `T`, then the action `a` by itself is considered a (unary) intended-atomic block. This terminology was chosen to emphasize that atomicity is a specification (an annotation) and not a guarantee provided by the platform.

Similarly, an *operation execution* in a trace α is the sequence of actions executed by a thread t starting with `call(T,m, i )` and ending with the matching `ret(T,m, i )` action. For simplicity, we restrict our attention to executions where for all `begin(T, l, s, i )` actions (resp. for all `call(T,m, i )` actions), a matching `end(T, l, s, i )` action (resp. a `ret(T,m, i )` action) exists.

Two actions (or events) in a trace conflict if:

1. read or write the same variable and at least one of the accesses is a write;

2. they acquire or release the same lock; or

3. they are performed by the same thread.

If two actions (or events) do not conflict, they *commute*. A trace α is said to be an `f-permutation` of a trace β iff α and β both consist of `n` actions, and `f` is a permutation of $\{1,\dots, n\}$ such that, if $\alpha = a_1, a_2, \dots, a_n$, then $\beta = a_{f(1)}, a_{f(2)}, \dots, a_{f(n)}$. For each `i`, a_i and $a_{f(i)}$ are said to be corresponding actions and (i, a_i) and $(f(i), a_{f(i)})$ are said to be corresponding events. α is said to be a permutation of β if there exists such an `f` .

2.3 Run-time Race Detection

Reviews [27] infer that data race detection is the most well-studied problem in the area of verification of concurrent software Informally, a data race occurs in an execution whenever there are conflicting accesses to the same variable without proper synchronization. Absence of data races is highly desirable in concurrent programs when they are executed on multiprocessors, a situation that is increasingly common with the advent of commodity multi-cores. Detecting and eliminating data races is important for two reasons:

1. Read accesses may yield non-deterministic results in racy executions, which often result in unintended, surprising behavior, program crash and data corruption.

2. Data races are often symptoms of logical or programming errors. As an example of 1, consider the following program, where a programmer naturally expects at least one of tmp1 or tmp2 to be 42 at the end.

This expectation is based on the intuition that in each thread, an instruction that appears before another instruction in the program order is also executed before it; the operational semantics implied by this expectation is known as sequential consistency [18]. On a modern multiprocessor, due to the reordering of instructions by the compiler or the hardware, if this program starts from a state in which `X = Y = 0`, it is possible for it to end in a state with both `tmp1 = tmp2 = 0`. For performance reasons, typical runtime systems for concurrent programs do not provide sequential consistency for all executions. But consensus has emerged that sequential consistency should be ensured at least for data-race free programs [3, 22]. In most programming languages, data-race freedom cannot be checked precisely using static methods, and dynamic data-race detection remains the only alternative for eliminating surprising behaviors such as those described above. For these reasons, researchers have also suggested that data races be treated as a runtime exception similar to null dereference or array-out-of-bounds exceptions [6]. In this view, data-race freedom is used as a sufficient condition for sequential consistency. Later work has made a finer distinction between data races and sequential consistency. There is indication that it may be feasible also performance-wise to ensure sequential consistency and to avoid the nondeterminism due to races [2, 9, 20, 23]. If, in the future, platforms that ensure sequential consistency become widespread, this indirect use of data races and data-race detection may no longer be needed. However, the fact remains that unintended data races are often symptomatic of higher-level design errors in concurrent programs. With the exception of carefully crafted nonblocking data structures, where benign races are deliberately allowed for performance reasons, detection of races can serve as a debugging tool, determining executions which may have surprising behavior due to concurrency. This use of data races as a proxy for higher-level design and programming errors, and errors in proper use of synchronization policies in concurrent programs will continue even if race detection is no longer needed to ensure sequential consistency. We anticipate that formalizing race conditions for execution platforms, detecting and eliminating them will continue to be an active area of research and tool development

Defining a data race formally has been contentious with many competing definitions in the literature. The first popular algorithm for data-race detection was the Eraser algorithm [29]. This algorithm is based on the insight that programmers use locks to ensure exclusive access to shared variables; hence, it equates a data race with the absence of a consistent locking protocol. The following execution obeys the consistent locking protocol of always accessing `X` while holding the lock `Lck`.

However, the following execution has a data race because there is no common lock held by the two threads while accessing the variable `X`. To detect an inconsistency in the locking protocol, Eraser maintains for each variable `x`, a lockset `LS` initialized to the set of all locks in the program. It also maintains for each thread `t`, a variable `LH` containing the current set of locks held by thread t. At each access of x by thread t, `LS` is updated with the intersection of the LS and LH and a data race is reported if LS becomes empty. The emptiness of LS indicates that accesses to x have not been consistently protected by a single unique lock. The simplicity of Eraser and its intuitive appeal made it very popular. However, Eraser suffered from the problem of false alarms, i.e., it occasionally reported races that the programmer did not consider errors. There were two main reasons for it. 1. Some natural programming idioms change the locking discipline over time. For example, a variable may be allocated and initialized by one thread without holding any lock and then made available via a global pointer, which is protected by a lock. Another example is the common producerconsumer pattern in which the producer thread creates and initializes an object representing a work item without holding any locks and then puts it into a queue. The consumer then takes the object out of the queue and again accesses it without holding any locks. 2. Programmers often use custom synchronization primitives instead of locks to synchronize access to data.

Examples of such custom primitives are hand-crafted spin locks and double-checked locking. Researchers have tried to address the false alarms described above by adding embellishments to the basic Eraser algorithm. These embellishments usually take the form of a state machine attached to each shared variable [3,51,60]. A data race is reported only if the lockset becomes empty and the state machine enters a particular state. The state-machine approach has been unsatisfactory because, although the specification being checked becomes more complicated, the problem of false alarms is still not eliminated fully.

2.4 Happens-Before Relation

Recently, consensus has emerged around a precise definition of a data race based on the happens-before relation [17]. A large factor in the forming of this consensus was the use of this definition by the Java memory model [22]. The happens- before relation of an execution is a partial order on the events in the execution. Intuitively, the happens-before relation captures the causal relationships between the events in an execution; there is an edge from an event $\mathtt{e}_1$ to another event $\mathtt{e}_2$ if the execution of $\mathtt{e}_1$ enables $\mathtt{e}_2$ to happen. Formally, the happens-before relation is the transitive closure of the following set of edges:

1. the set of edges from one instruction to the next one executed by a thread.

2. the set of edges from a lock release to the subsequent acquiring of that lock.

3. the set of edges from the fork in one thread to the first operation of the forked thread.

4. the set of edges to the join operation in a thread from the last operation of the thread being joined with.

Recall that two accesses to a variable `x` are conflicting if at least one of them writes to `x`. Two conflicting accesses to a variable `x` in an execution form a data race if they are not ordered by the happens-before relation of the execution. The most well-known algorithm to check for the presence of data races defined using the happens-before relation is the vector-clock algorithm [24]. A vector clock maintains a clock, a non-negative integer, for each thread in the program. A vector clock $\texttt{VC}_1$ precedes a vector clock $\texttt{VC}_2$, if for each thread t, $\texttt{VC}_1(\text{t}) \leq \texttt{VC}_2(\text{t})$.

The algorithm maintains the following collection of vector clocks:

1. $\texttt{VC}_t$ for each thread `t`. The value of $\texttt{VC}_t$ is the vector clock for the last event executed by thread `t`. It is updated whenever thread `t` executes an event.

2. $\texttt{VC}_m$ for each lock `m`. If the last thread to release lock `m` was `t`, then the value of $\texttt{VC}_m$ is the vector clock of `t` when it released `m`.

3. $\texttt{VC}_{Wx}$ for each variable `x`. If the last thread to write variable `x` was `t`, then the value of $\texttt{VC}_{Wx}$ is the vector clock of t when the write happened.

4. $\texttt{VC}_{Rx}$ for each variable `x`. If the last thread to read variable `x` was `t`, then the value of $\texttt{VC}_{Rx}$ is the vector clock of `t` when the read happened.

The vector clocks are updated as events occur in an execution.

The vector-clock algorithm dynamically assigns to each event in the execution a particular vector clock preserving the following invariant:

For any pair of distinct events `e` and `f` , the vector clock assigned to `e` precedes the vector clock assigned to `f` iff the event `e` is ordered before the event `f` according to the happens-before relation.

Race detection is performed using vector clocks as follows:

1. When a thread t reads variable `x`, a race on`x` is declared unless $\texttt{VC}_{Wx}$ precedes $\texttt{VC}_t$.

2. When a thread t writes variable `x`, a race on `x` is declared unless $\texttt{VC}_{Wx}$ precedes $\texttt{VC}_t$ and $\texttt{VC}_{Rx}$ precedes $\texttt{VC}_t$.

Together, these rules serve to compute the happens-before relation indirectly.

2.4.1 Fasttrack Algorithm

Fasttrack [7] algorithm improves the happens-before discussed on section 2.4 to reduce slowdown while maintaining algorithm precision. FastTrack identifies some unnecessary large computations performed by vector-clocks algorithm by adapting a lightweight representation of vector clocks. This requires constant space and experimental results show that most of the operations do not the whole of a vector clock to maintain the happens-before relation.

To avoid unnecessary computations, FastTrack classifies a race condition as read-write race condition where the program trace contains a read that is concurrent with a later write to the same variable; a write-read race condition where a write is concurrent with a later read); or a write-write race condition which involves two concurrent writes).

Detection of Write-Write Data Races . For consecutive writes to a variable by various threads, it is revealed the the operations are totally ordered on that variable assuming that no races have happened so far on that variable. Therefore, in order to detect race on this situation it suffices to store the clock and thread id of the thread just wrote to the memory at the last time. The clock and id are stored on a small data structure call epoch. It takes constant time to check for write-write race conditions using the clock and the thread id from the epoch. This reduces slowdown of the algorithm largely.

To summarize, epochs reduce the space overhead for detecting write-write conflicts from O(n) to O(1) per allocated memory location, and replaces the O(n)-time vector clock comparison with the O(1)-time comparison operation.

Detection of Write-Read Data Races . Detecting write-read races under the new representation is also straightforward. On each read from variable `v` with current thread vector clock $\texttt{VC}_t$ (refer to section 2.4), FastTrack checks that the read happens after the last write via the same O(1)-time comparison of the epoch's clock and the clock from $\texttt{VC}_t$.

Detection of Read-Write Data Races . Detecting read-write race conditions is complicated. Unlike write operations, which are totally ordered in the case that no race conditions detected so far for a variable, reads are not totally ordered even in race-free programs. Thus, a write to a variable `v` could potentially conflict with the last read of `v` performed by any other thread, not just the last read in the entire trace completed so far. Hence, FastTrack may need to record an entire vector clock $\texttt{VC}_{Rv}$, in which $\texttt{VC}_{Rv}$(t) records the clock of the last read from `v` by thread `t`.

FastTrack is precise and reports data races if and only if the observed trace contains concurrent conflicting accesses.

2.5 Dynamic Instrumentation

Dynamic Instrumentation is technique for observing the target application's execution trace. It is used to gather the necessary data on the execution of a program for race detection. There exist several industry strength frameworks for dynamic instrumentation of executable program. Some of them are Intel(r) Pin [21], Valgrind [25], DynamoRIO [4] and RoadRunner [8]. Intel(r) PIN instruments C/C++ binary programs buy running tools called Pintools on PIN runtime with the the binary as parameter. Pintools can be used to perform program analysis on user space applications in Linux and Windows operating systems. Thus, it requires no recompiling of source code and can support instrumenting programs that dynamically generate code. Pin provides a rich API that abstracts away the underlying instruction set idiosyncrasies and allows context information such as register contents to be passed to the injected code responsible for race detection as parameters. Pin automatically saves and restores the registers that are overwritten by the injected code so the application continues execution from previous snapshot.

Chapter 3

TRANSACTIONAL MEMORY

Chip Multithreading (CMT) processors promise to deliver higher performance by running more than one stream of instructions in parallel rather than by increasing the processors frequency. CMT processors may come with different architectures: Chip Multi-Processor (CMP), Simultaneous Multi- Threading (SMT), or a combination of them. To exploit CMTs capabilities, users have to parallelize their applications. Unfortunately, efficiently parallelizing applications is not trivial. Several proposals focus on how to reduce the effort of parallelizing applications on CMT machines. Fine-grained locking techniques provide good performance but pose challenges to programmers. Consequently, automatic parallelization techniques and innovative programming models have been proposed to reduce programmers effort. Transactional Memory [11] (TM) is one of such novel programming models. The main goal of TM is to simplify synchronization by raising the level of abstraction and composition, breaking the connection between semantic atomicity and the means by which that atomicity is achieved. Programmers indicate atomic section in the source code by using language constructs such as atomic blocks, or using macros such as TM_BEGIN and TM_END without explicitly locking individual shared memory locations. An underlying TM system executes such transactions concurrently whenever possible, generally speculatively or optimistically, rollbacking when conflicts encountered. Transactions commit or abort atomically, i.e., either all memory locations modified during the transaction are updated (committed) or nothing is modified (abortted). Transactions are allowed to commit only if they have no conflicts or all their conflicts are resolved positively. There are commonly three types of transactional memory. These are Software Transactional Memory, Hardware Transactional Memory, and, a combination of these two.

3.1 Software Transactional Memory

TM semantics are implemented on software as libraries or run-times for Software Transactional Memory. Appropriate data structures for read- and write-sets per transaction are implemented on software. Software Transactional Memory is typically slower.

3.1.1 STM^2 [14]

STM^2 is the first parallel STM system that uses secondary hardware threads to leverage STM overhead.

STM^2 is essentially a parallel STM system where transactional operations are divided between application threads (computation) and auxiliary threads (STM management). With STM2, application threads optimistically perform their computation with minimal support from the underlying STM system. All synchronization and STM management operations are performed by the paired auxiliary threads. This means that application threads experience minimal overhead. Auxiliary threads, instead, validate read-sets, maintain transaction states and detect conflicts in parallel with the application threads computation. STM^2 detects conflicts as soon as they occur (eager conflict detection). If a conflict is detected, the auxiliary thread interrupts its corresponding application thread and aborts the transaction. If no conflicts arise during a specific transaction, the auxiliary thread commits the transaction and updates the modified shared memory location (lazy update). Communication between application threads and their corresponding auxiliary threads is performed through a lock-free circular buffer and simple atomic state variables.

3.2 Hardware Transactional Memory (HTM)

HTM involves modification to processor architecture, caches and bus protocols in order to handle buffering, conflict detection, validation and other mechanisms of TM. HTMs are generally faster than STM.

3.2.1 Transactional Synchronization Extensions (TSXs)

Intel TSXs provide an interface of instruction set extensions which allow programmers to specify regions of code for transactional synchronization. Programmers can use these extensions to gain the performance of fine-grain locking while actually programming using coarse-grain locks [12, 13].

With transactional synchronization, the hardware can determine dynamically whether threads need to serialize through lock-protected critical sections, and perform serialization only when required. This lets the processor expose and exploit concurrency that would otherwise be hidden due to dynamically unnecessary synchronization.

At the lowest level with Intel TSX, programmer-specified code regions (also referred to as transactional regions) are executed transactionally. If the transactional execution completes successfully, then all memory operations performed within the transactional region will appear to have occurred instantaneously when viewed from other logical processors. A processor makes architectural updates performed within the region visible to other logical processors only on a successful commit, a process referred to as an atomic commit.

These extensions can help achieve the performance of fine-grain locking while using coarser grain locks. These extensions can also allow locks around critical sections while avoiding unnecessary serializations. If multiple threads execute critical sections protected by the same lock but they do not perform any conflicting operations on each others data, then the threads can execute concurrently and without serialization. Even though the software uses lock acquisition operations on a common lock, the hardware is allowed to recognize this, elide the lock, and execute the critical sections on the two threads without requiring any communication through the lock if such communication was dynamically unnecessary.

Intel TSX provides two software interfaces [12, 13]. The first, called Hardware Lock Elision (HLE) is a legacy compatible instruction set extension (comprised of the XACQUIRE and XRELEASE prefixes) that are used to specify transactional regions. HLE is compatible with the conventional lock-based programming model. Software written using the HLE hints can run on both legacy hardware without TSX and new hardware with TSX. The second, called Restricted Transactional Memory (RTM) is a new instruction set interface (comprised of the XBEGIN, XEND, and XABORT instructions) that allows programmers to define transactional regions in a more flexible manner than is possible with HLE. Unlike the HLE extensions, but just like most new instruction set extensions, the RTM instructions will generate an undefined instruction exception (#UD) on older processors that do not support RTM. RTM also requires the programmer to provide an alternate code path for when the transactional execution is not successful.

Chapter 4

PARALLEL RUNTIME VERIFICATION TOOL (PARV)

We present the PaRV tool for runtime detection of and recovery from data races in multi-threaded C and C++ programs. PaRV uses transactional memory technology for parallelizing runtime verification and for buffering write accesses during race checking. Application threads are slowed down only due to instrumentation, but not due to the computation performed by runtime verification algorithms since the latter are run concurrently on different threads. Buffering writes allows us to recover from races and to safeguard against later ones.

4.1 Introduction

We present PaRV, a tool for runtime detection of and recovery from data races in multi-threaded C and C++ programs. We use components from transactional memory (TM) implementations in order to parallelize runtime verification and to buffer write accesses until they are determined to be free of races.

Concurrently with each application thread, a sibling thread in the style of [14] performs race detection using the Fasttrack algorithm [7]. This approach to parallelized runtime verification minimizes application slowdown. The application thread only experiences slowdown due to instrumentation. Once the sibling thread determines that the accesses within a block are free of races, the accesses in the buffer are committed to memory. If a race is detected, the block is rolled back, and extra synchronization is performed on variables experiencing races, which allows the execution to continue without race conditions. In its current form, our approach allows race-free execution of application binaries at a modest overhead even for legacy applications. With the availability of TM hardware in upcoming microprocessors and with a large number of cores expected to be available on processor chips, we expect our approach to have further reduced performance overhead and wide applicability for legacy applications.

4.2 Transactional Memory and Runtime Verification

Runtime verification slows down applications. For instance, race detection slows down C/C++ programs by 100 times or more. High overheads make post-deployment use of such runtime monitoring techniques infeasible. Even during pre-deployment testing and runtime verification, such high overheads make it unlikely that runtime verification techniques will be used continuously during all runs.

Transactional memory implementations contain highly optimized mechanisms for logging and buffering events, and, in the case of parallelized implementations of transactions, for efficient inter-thread communication between threads working on the same transaction. Hardware vendors have started providing hardware support for transactional memory, which will make approaches using TM more efficient in the near future.

A related approach is that of log-based architectures (e.g., [5], [28]) which provide on-chip hardware resources for reducing the runtime overhead that comes from monitoring executions. Differently from log-based approaches, our tool does not need any additional hardware support but can benefit from it. Differently from how [28] makes use of hardware TM, we do not make use of conflict detection and version number management – parts of a TM implementation that incur significant computational overhead. We use the high-performance FastTrack algorithm instead.

One important way our approach will benefit from hardware support for TM announced or provided by processor vendors is by obviating the need for software-based synchronization to protect race checking metadata, such as vector clocks. With compiler support available for TM hardware, our approach will immediately enjoy the benefit of improved performance due to TM hardware.

STM^2 [14] is a novel, multi-threaded STM design, where each application thread has a dedicated auxiliary ("sibling") thread performing STM operations such as validation of read-sets, bookkeeping and conflict detection. The communication between application and auxiliary thread is provided by a communication channel and atomic status variables. The communication channel is implemented with a single-producer/single-consumer, circular, lock-free queue where the application thread (producer) posts read and write messages that the auxiliary thread (consumer) retrieves and processes them. We use STM^2's queue to communicate read, write and synchronization operations from the application thread to the sibling thread carrying out race detection. We also use STM^2's write buffering and transaction commit mechanisms to delay writing to memory of writes until they are shown to be bug-free. Specifics of these are explained in the next section.

4.3 Tool Architecture and Implementation

Figure 1 shows position of PaRV relative to DynamoRIO-Dynamic instrumentation tool. The high level organization of the tool is as follows. Instructions performed by each application thread are instrumented using the dynamic DynamoRIO binary instrumentation framework [4]. Between every application thread and its corresponding sibling thread, there is a FIFO queue (figure 2) in the style of the STM^2 circular buffer that the application thread writes to and the sibling thread reads from. On the application thread, read and write accesses and synchronization operations are instrumented such that for each of these instructions executed, an *event* is placed on the FIFO queue. The sibling thread removes events from the queue and is able to carry out race detection for the sequence of instructions carried out by the application thread in this way.

The sequence of instructions performed by each thread are divided into non-overlapping portions called *consistency block*s using DynamoRIO binary instrumentation. Every synchronization event is in a consistency block by itself. The sequence of instructions performed by an application thread between two synchronization events constitute a block otherwise. The application thread and the sibling thread synchronize at consistency block boundaries. The application thread buffers all write accesses

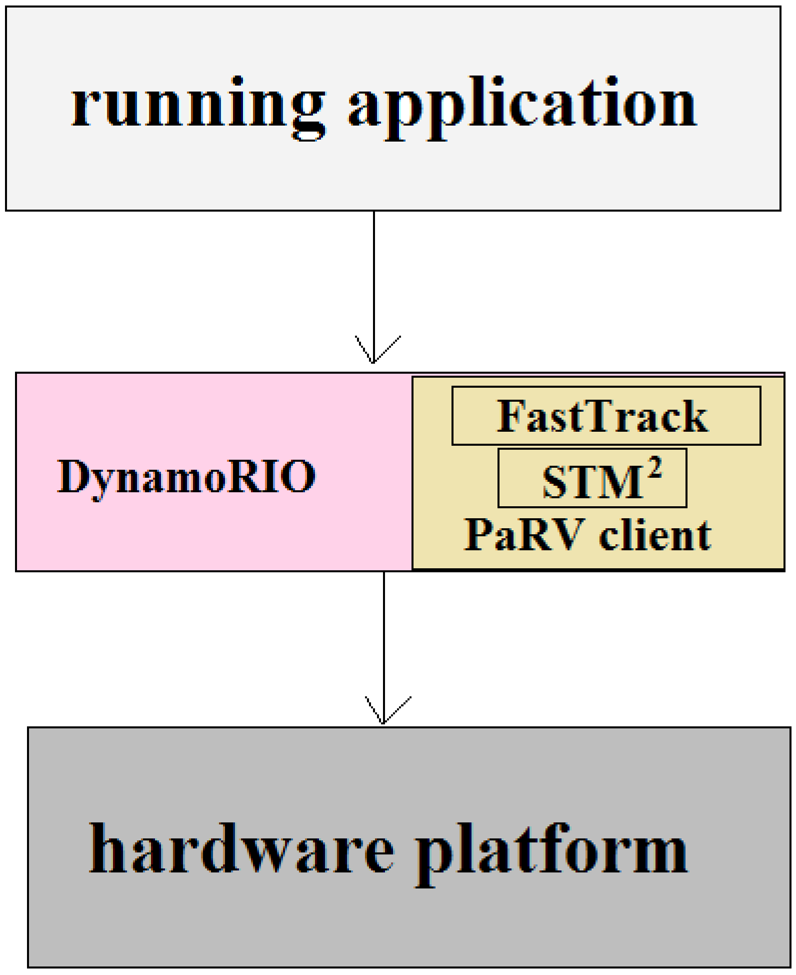

Figure 4.1: Architecture of PaRV.

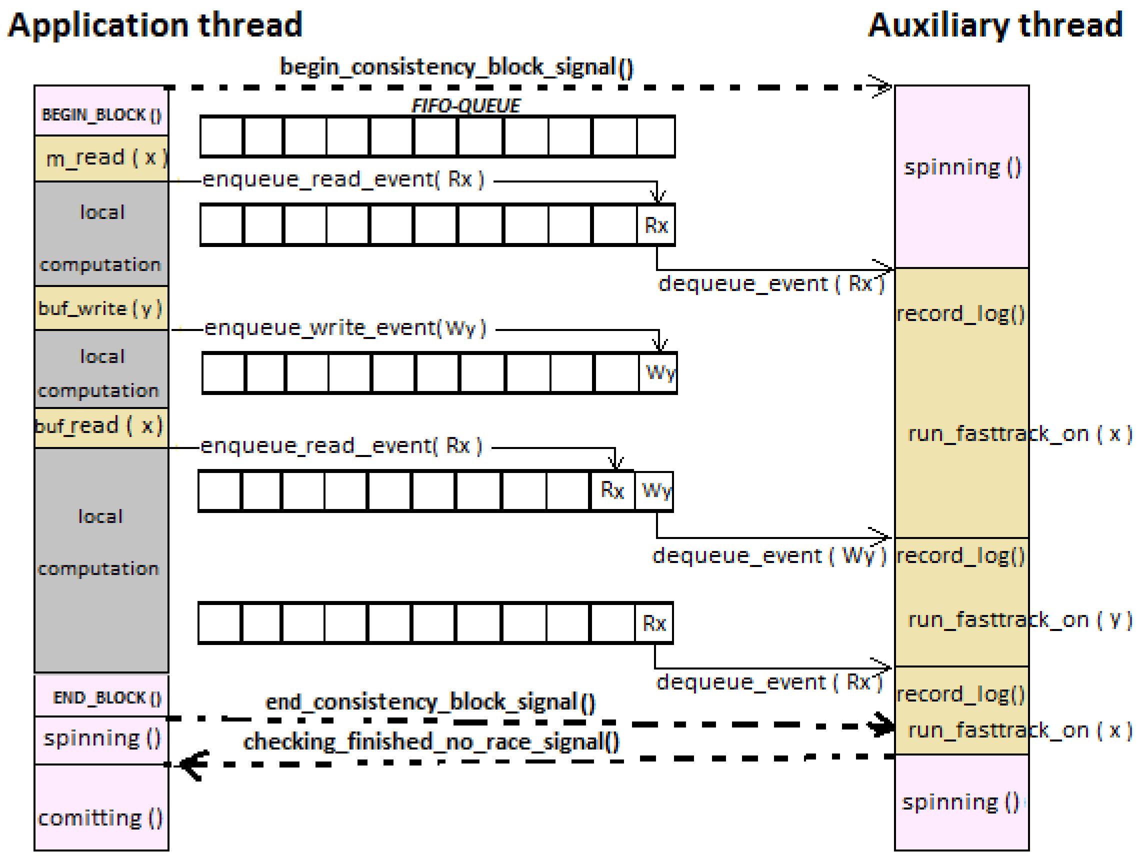

Figure 4.2: Application-auxiliary thread interaction

it performs. For consistency blocks that do not contain synchronization operations, when the sibling thread signals to the application thread that the processing of the block is complete, and detects no concurrency errors, the application thread commits the writes in the buffer to memory. For consistency blocks consisting of synchronization operations, the application waits for the sibling thread to complete processing the consistency block before it actually performs the synchronization operation. This is necessary for the runtime verification carried out by the sibling threads to have the same happens before relation as the execution produced by the application threads. Before using DynamoRIO to realize the implementation we tried our approach with PIN. However,with PIN we could not do some of the approaches discussed.

4.3.1 Runtime instrumentation with DynamoRIO

Using DynamoRIO, the write and read accesses and synchronization operations performed by application threads are modified.

A write access (store instruction `ins`) is instrumented to implement the following steps. First, the address and the value to be written are extracted from `ins`. Then, an entry is written into the write buffer of the application thread, and an event corresponding to the write access is placed on the FIFO queue. The write instruction is then skipped. This is necessary, since we only want to commit to main memory writes determined to be free of concurrency errors. The write buffer implementation is borrowed from STM[2]

```
addr = get_destination ( ins );
val = get_value ( ins );
write_to_buffer( addr, val);
enqueue_write_event( addr);
skip_instruction ( ins );
```

A read access (load instruction `ins`) is instrumented so that a variable that was written to earlier by the current consistency block gets its value from the write buffer. Other reads get their value from the main memory.

```
read_from_local_write_buffer_or_mem ( ins );
addr = get_memory_operand ( ins );
enqueue_read_event ( addr );
```

When a consistency block ends, the application thread waits for the sibling thread to set an atomic signal to indicate completion of runtime verification for the curent consistency block. At that time, the write buffer is committed to main memory. Unlike the commit phase of an STM implementation, we do not need to acquire locks for the variables in the write buffer, since we are not carrying out conflict detection between consistency blocks in the sense of TMs. If no race has been detected by the sibling thread, then write buffers can safely be written to memory, since there are no concurrent racy writes. This is a significant factor in reducing the instrumentation overhead below what an STM would experience.

Before every synchronization operation, we end the ongoing consistency block if there is one, and then start a new one. We put on the FIFO queue an event representing the synchronization operation. When the sibling thread is done processing this event and notifies the application thread by setting an atomic variable, the application thread continues, performs the synchronization operation, and starts the new consistency block.

4.3.2 Detecting and Recovering from Races

Each sibling thread applies to the stream of events it receives from the event FIFO queue the FastTrack race detection algorithm [7]. FastTrack is an efficient, precise race detection algorithm. The algorithm is described by providing the updates and checks performed by each thread for each memory access or synchronization operation. In our tool, differently from the original FastTrack, the application thread only records the events in the FIFO queue. The race detection computation is performed on the sibling thread for each event as it is removed from the FIFO queue. We implemented FastTrack in C based on the original implementation. The shared variables (e.g. vector clocks and epochs) used by FastTrack are protected by mutual exclusion locks. The sibling thread notifies the application thread of races or race-free completion of consistency blocks by setting atomic variables.

By buffering write accesses until the end of a consistency block, we are able to prevent racy writes from being written to memory, and racy reads from affecting later code. At the end of a consistency block, if the sibling thread signals a detected race condition, the consistency block is aborted (the write buffer discarded) and retried. The sibling thread notifies the application thread of the set of variables that experienced a race condition during the last execution of the consistency block. The application thread, when retrying the block, wraps each access to a racy variable x by an acquire and release of the lock that protects vc_x, the vector clock of x. Since the last access by another consistency block to x was followed by the sibling thread's access to vc_x, this ensures a happens-before relationship between the accesses and prevents a race condition. After a race is detected on x, all later accesses to x by application threads are protected by vc_x. By doing this for only variables that experience a race, we keep the performance overhead of our approach low.

4.4 Related Work

PaRV builds on research in the areas of transactional memory and dynamic race detection. It also bears similarities to approaches in the architecture literature for instrumenting and logging program executions, parallelizing dynamic monitoring, containing and recovering from errors encountered. In the following, we contrast PaRV with these approaches.

ParaLog [32] extends work on log-based architectures [5] provide hardware support for instrumenting, logging and monitoring executions of multithreaded programs. Techniques in ParaLog not only reduce the application slowdown due to instrumentation and logging, but also allow, similarly to PaRV, parallelized monitoring algorithms to be run on separate resources from the application, thus further reducing slowdown. ParaLog involves significant changes to processor and memory architecture. It accomplishes efficient tracking of ordering of events from different threads by monitoring cache coherence traffic. PaRV works on currently available, stock microprocessors, but If a platform provides LBA support, PaRV would incur much less slowdown as well.

Race-detection depends critically on, and almost entirely consists of tracking inter-thread dependencies precisely, and the multiple threads in the monitor accessing the per-address and per-thread metadata atomically. The hardware support in ParaLog directly targets efficient implementations of these operations. Taking an alternative approach, PaRV aims to reduce race-detection slowdown as much as possible in the absence of hardware support for monitoring. Differently from ParaLog, PaRV uses TM technology to prevent races, and explicitly inserts extra synchronization into the program for avoiding later races.

The authors in [1] present the KUDA tool, which, similarly to PaRV, separates race detection from application execution threads using kernel threads in the GPU as helper threads. Differently from KUDA, PaRV synchronizes the application and helper threads so that race detection does not lag behind. This is essential for prevention of and recovery from races, two more features that distinguish PaRV from KUDA. KUDA also parallelizes race detection further than one helper thread per application thread in order to make use of the high degree of parallelism provided by the hundreds of cores on a GPU.

Veeraraghavan, et al in [30] present the Frost tool that addresses detection and prevention of data races by running multiple replicas of an application using complementary schedules. Races are detected by comparing states reached by different replicas, instead of processing event sequences. While providing significant reduction in slowdown, this approach suffers from two key weaknesses. First, for an application with faulty synchronization, it is quite possible that no schedule leads to race free execution. PaRV addresses this problem by adding synchronization to the program as needed. Second, race detection in Frost is imprecise. PaRV uses the FastTrack algorithm for precise detection of races.

Chapter 5

IMPROVING RUNTIME ERROR DETECTION USING TSXS

Due to increase in necessity to program in multicore environment, it has become prominent to design tools to simplify complexities of parallel programming. Concurrent software for multicore environment can be prone to concurrent errors like deadlocks, atomicity violations and data races. Detecting these errors in such software can be challenging. We present a way to improve checking of data races using Hardware architectural assisted transactional memory to improve race detection. We show that using Intel's TSXs we can achieve up to 4X speedup over legacy race detection using FastTrack [7]; a fast and precise race detection algorithm.

5.1 Introduction

There are two race detection techniques; static and dynamic. Static race detection produces false alarms and very slow. Dynamic race detection performs better in terms of precision but it is still slow and it can not be turned on all the time during software development. Between these two approaches there is trade-off between precision and performance. Along these works there are proposals on use of a special, extra hardware to accelerate race detection. Our approach relies on improvements of microachitecture without need of extra hardware. Moreover precision of data race detection maintained. Our approach improves over FastTrack algorithm; the fast and precise race detection algorithm. Performance results from Splash-2 [33] benchmark indicate that up to 4x speedup is achieved.

Data races are a symptoms of high level concurrency errors which can lead to various consequences from program crash to loss of millions of dollars, to lives. Elimination of races from multicore-software is crucial along software development. A problem of performance slowdown of dynamic race detection is that it imposes a slowdown of about 100X. Basically, if it was fast enough, people would turn on race detection full time while running all of their tests.

The components of the slowdown dynamic race detection introduces;

- slowdown due to instrumentation of instructions,
- slowdown due to the computation that the race detection algorithm performs,
- slowdown due to the fact that there needs to be synchronization to protect the meta-data that the race detection algorithm uses.

We propose a solution that improves performance of race detection up to 34X speedup over the fastest race detection algorithm using HARDWARE transactional memory technology [12].

Our solution involves using architectural supported transactional memory to benefit from fine-grain concurrency of race detection meta-data. Our contributions are:

- We propose use of TSXs to aid race detection and show that it improves performance
- We not only simply protect the race-detection metadata using transactions, but also we break down a binary into proper-sized chunks in order to minimize overhead.
- We present a proper use of of TSXs to achieve precise race detection with much reduced slowdown
- We propose an algorithm to instrument a program into small transactions for race detection
- We show that splitting a program into small transactions does not change semantics of the program and it does not affect precision of race detection.

The following sections discuss the design and implementation details and evaluations of our approach.

5.2 Design and Implementation

5.2.1 Division of Program into Transactional Blocks

A transactional block – sometimes called transactional region – is a group of program actions/operations preceded by a *"transaction begin"* instruction and succeeded by *"transaction end"* instruction. These instructions form a transaction block which execute in a transactional manner.

The instrumentation in section 5.2.3 wraps a bunch of actions(sometimes refereed to as operations) and race detection function calls(from now on we will call them callbacks) in transactional blocks. A typical block starts with `XBEGIN`, contains a mixture of one or more actions and race detection callbacks, and ends with `XEND`.

`XBEGIN` wraps *xbegin* hardware instruction with multiple tries in case it fails or aborts. `XEND` wraps *xend* hardware instruction which commits the transaction if the transactions executes successfully.

The intention of existent transactional block is to execute the block speculatively in order to gain fine-grain concurrency performance. Ideally no lock should be used to protect any shared data inside the transactional block since transactional memory ensures safety through conflict detection and safe commits. Nevertheless, TSXs [12] do not guarantee that a block commits due to a number of reasons including but not limited buffer overflows and presence of un-friendly instructions. Therefore, it is important to provide an alternative path in case transaction fails to execute/commit. In our case of race detection the alternative fall-back is re-execution of the block non-transactional way. In this case the FastTrack meta-data are protected by locks as described in section 5.2.2.

The first part of our approach is to divide a given program into small transactional blocks. The aim of this is to achieve fine-grain locking performance of the race detection algorithm inside the transactional block. In order to maintain the original semantics and structure of the program, we design a way to consistently divide the program into blocks using program actions presented in section 5.2 and other new actions we introduce in the following paragraphs. For simplicity, we group these actions into sets of actions and refer to these sets while we describe our algorithm for formation of the transactional blocks.

TRANSACTION_BARRIER This set includes operations performed by a thread but serve as a means to divide program into transactional regions. They are used for this purpose because some for program split because they can not be inside a transactional region as they mail fail it or a source of transactional size controll. The actions or operations that constitute this set can be detailed in the following paragraphs

function call, function return, system call A function call creates a new execution stack and hence a new block. It involves completely moving the stack pointer to a new index. System calls include illegal instructions that can cause all-time transaction aborts. In order to control the transaction sizes, the previous transaction is ended before the call is commenced. The call precedes a begin of a new transaction.

Function return follows similar trend; execution claims previous stack pointer at function return. This way the return succeeds an end of transaction block.

input and output operations Transactions do not support input/output actions as they are irrevocable operations. Therefore each of these operations succeeds a transaction block and precedes a new block, staying outside transaction blocks.

conditional or unconditional jump A conditional / unconditional jump transitions execution sequence to a new stack pointer state. To limit transactional size from program loops these operations mark end of transaction and resume of a new block.

GLOBAL_OPERATION In this context load and store operations represent shared data retrieved from and stored to memory shared among concurrent threads, respectively. These operations account for data race detection. These operations constitute part of a transaction region.

SYNCHRONIZATION Synchronization operations include thread creation, thread join, lock acquire, unlock release, and barriers. These are high level operations as they may include (encapsulate) other operations like system calls, and synchronization operations. They might also contain load and store operations and experience benign race conditions. These operations are either kept in their separate transaction blocks or completely outside of the transactional block.

thread creation, thread join and barriers There actions involve system calls or loops. They mark the end of the currently executing transaction if there is any. Moreover, due to vulnerability to transactional memory as they may contain illegal instructions, they are executed outside of the transactional memory. As we will discuss, they have their corresponding race detection callbacks. In order for them to benefit from transactional memory concurrency, they are kept in new separate transactional regions.

lock acquire, unlock release operations We have experimented with these operations inside the transactions and they work friendly. We keep them on separate transactions together with their race detection callbacks. This has two implications. First, we make sure that execution semantic is maintained and it ensures a thread clears execution of the lock operation before moving to next transactional region. Some *pthread* libraries [15] implement lock operations which do not really acquire or release the lock when executed in transactional memory. This serves more performance improvement.

Arithmetic operations perform on the data loaded into operating registers.

Formally a transaction block can be defined as follows

block := local+ | global+ | synchronization-operation synchronization-operation := lock | unlock

lock := arithmetic | reg-reg transfer

global := load | store := load, store+

5.2.2 Addition of Race Detection Callbacks

race_check_callback This is a function call to execute some part of FastTrack algorithm depending on the action succeeding it. For instance, the callbacks for lock action and unlock actions are *race_check_lock* and *race_check_unlock* respectively. While the callback for thread forking action is *race_check_thread_create*, the callback for thread joining action is *race_check_thread_join.* Moreover, callbacks for shared memory load is *race_check_read* and callback for shared memory store is *race_check_write.* These callbacks are summarized as follows.

- *race_check_thread_create*, this involves initialization of the vector clocks of the newly created thread.

- *race_check_thread_join*, updates the callers vector clocks according to the vector clocks of the terminated thread.

- *race_check_lock*, immediately called before the lock is actually acquired to copy the lock vector clocks into the thread vector clock.

- *race_check_unlock*, in this function vector clocks of a releasing thread are copied to the vector clocks of the lock.

- *race_check_read*, race detection is performed to see if the memory read is prone to race. Race checking involves the comparison of read and write vector clocks of the memory and that of the thread accessing it.

- *race_check_write*, race detection is performed to see if the memory read is prone to race. Race checking involves comparison of read and write vector clocks of the memory and that of the thread accessing it.

- *race_check_barrier*. executed by each thread at the barrier, it ensures that each vector clock of a thread contains maximum of clocks ever achieved by any thread at the barrier.

Any race detection callback resides inside the transactional memory block. It checks if the transactional block did not fail at trial. In case the transaction fails to initiate, the block is executed as an alternative path. Therefore, the callback acquires necessary FastTrack meta-data locks in order to protect them from data races within the FastTrack algorithm. Therefore any race detection callback extends to a general structure as in the figure 5.1. When FastTrack callbacks executed for race detection or updating FastTrack meta-data they test if called within an executing transaction they don't acquire any locks to protect FastTrack data. Otherwise, they acquire necessary locks to protect FastTrack data.

For comparison, all FastTrack synchronizations are fine-grain. That is, every vector clock/epoch for each of a thread, a lock, and and memory location is protected by its unique lock.

FastTrack implementation

Implementation of FastTrack mimics algorithms from [7]. For a fair comparison of FastTrack running under transactional support against normal FastTrack, we implemented FastTrack meta-data with fine-grain locking. Every vector clock or epoch for variables, locks and threads there is a single unique lock.

```
function race_check_callback* {
    if transaction failed to start
    then
         Acquire all necessary locks to protect FastTrack
         Do necessary race check and computations
         Release all necessary locks to protect FastTrack
         end
    otherwise
         Do necessary race check and computations
         end
}
```

Figure 5.1: Common structure of FastTrack functions

.

During race checking a thread acquires the lock of the vector clock or epoch it wants to access first.

5.2.3 "Instrumentation" Algorithm

Dynamic race detection requires instrumentation of the binary program to be checked. The instrumentation recognizes important program actions and adds necessary race detection function callbacks. In the case of our approach of using TSXs, instrumentation wraps small groups of actions into smaller chucks called transactional regions. This is achieved by adding TSX instructions to the executable using necessary instrumentation algorithm running on an instrumentation tool like PIN. We devise an algorithm 5.2.1 to instrument and add necessary operations to the executable. Details of the algorithm are on section 5.2.4.

The instrumentation algorithm is an adaptation of program instrumentation which recognizes all program actions discussed in part 5.2 in sequential ordered form of the executable before execution.

The provided algorithm produces a shadow executable from original executable by wrapping global actions and some synchronization operations into transactional blocks. It ensures that number of actions in each transactional block formed does not exceed the maximum size specified by `MAX_CAPACITY`. For example, if `MAX_CAPACITY` is 2 and there are 3 consecutive global operations. Then they comprise two transaction blocks; one with two actions and the rest with one action.

It should be noted that the algorithm forms transactional blocks depending on the program characteristic actions discussed on section 5.2. Regardless of the number of actions added to the currently formed transaction if the current action from instrumented executable is an element of SYNCHRONIZATION or TRANSACTION_BARRIER sets then the transaction ends immediately by the enclosure of `XEND` and new transaction initiates when necessary depending on the current action. Element actions of TRANSACTION_BARRIER set terminate the previous transaction and do not initiate new transaction a new one, while, actions from SYNCHRONIZATION class initiate new transaction immediately before that action or after.

Actions other than those discussed one section 5.2 just get appended to the shadow executable with any additional modification.

5.2.4 Important Parts of the Algorithm

Input and output The input of the algorithm is an executable program to be instrumented. This executable comprises various actions which are discussed on section [3.1]. For simplicity we have abstracted output is output as new shadow executable which contains all actions from input some of which encapsulated into transactional blocks and with an addition of race detection function calls. Figure 5.2 shows how input and output look like before and after instrumentation, respectively.

Initialization Algorithm initializes a new shadow executable output at line 1 as an empty list of ordered actions. Line 2 initializes *openTransaction* variable to false. This variable keeps track, along instrumentation, of an open transaction initiated with `XBEGIN` previously so that its corresponding `XEND` can be added when need at current or upcoming action during instrumentation.

currentTransactionSize is initiated at line 3. This is mostly used to keep number of actions in a currently open transaction. It helps to determine when a current transaction should be succeeded with *XEND* once its size becomes `MAX_CAPACITY` at line 21-25.

Loop This is between line 4 - 39. Algorithm loops through all actions present in the executable input. The executable input is modeled as an ordered list of actions, therefore, the algorithm checks one action before the other until they finish. In the algorithm loop it checks type of current action and performs insertions depending the type. The loop has three sub parts which execute depending on type of the current action.

- *Current action is element of TRANSACTON_BARRIER*

In the case that the current action being instrumented is belongs to TRANSACTON_BARRIER group of actions then if the formation of transaction block has stated that block finishes by insertion of `XEND` [lines 6 -12].

- *Current action is GLOBAL_OPERATION*

  Lines 13 -25. This initiates an new transactional block in case there is no open block. Otherwise, the action gets appended to current block. In case that the current block size has reached maximum block capacity the block is closed with `XEND`.

- *Current action is SYNCHRONIZATION operation*

  This ends previous open transaction in case there was any. It then creates a separate transactional block and wraps appropriate race detection callback in it. The action resides inside that block if and only if it is a lock acquire or release operation.

Algorithm 5.2.1 Algorithm to instrument a multithreaded program

Input: Executable *exec* executable to be instrumented.

Output: Instrumented executable: *newExec*

```
1:  $newExec := \{\}$
2:  $openTransaction := FALSE$
3:  $currentTransactionSize := 0$
4:  while $exec$ has next action do
5:      $action := getNextAction(exec)$
6:      if $action$ is element of $TRANSACTION\_BARRIER$ then
7:          if there is open transaction then
8:              $appendAction(newExec, XEND)$
9:              $openTransaction := FALSE$
10:         end if
11:         $appendAction(newExec, action)$
12:         $currentSize := 0$
```

```
13: else if $action$ is element of $GLOBAL\_OPERATION$ then
14:     if there is NO open transaction then
15:         $appendAction(newExec, XBEGIN)$
16:         $openTransaction = TRUE$
17:     end if
18:     $appendAction(newExec, race\_detection\_callback)$
19:     $appendAction(newExec, action)$
20:     $increment(currentTransactionSize)$
21:     if current transaction size equals $MAX\_CAPACITY$ then
22:         $appendAction(newExec, XEND)$
23:         $openTransaction := FALSE$
24:         $currentTransactionSize := 0$
25:     end if
26: else if $action$ is element of $SYNCHRONIZATION$ then
27:     if there is open transaction then
28:         $appendAction(newExec, XEND)$
29:         $openTransaction := FALSE$
30:     end if
31:     if $action$ is NOT lock or unlock operation then
32:         $appendAction(newExec, action)$
33:     end if
34:     $appendAction(newExec, XBEGIN)$
35:     $appendAction(newExec, race\_check\_callback)$
36:     if $action$ is lock or unlock operation then
37:         $appendAction(newExec, action)$
38:     end if
39:     $appendAction(newExec, XEND)$
40:     $currentTransactionSize := 0$
41: else
```

```
42:         $appendAction(newExec, action)$
43:     **end if**
44: **end while**
45: **return** $newExec$
```

5.2.5 Instrumentation Example

We provide an example to show how the instrumentation algorithm divides program into transactional regions and adds race detection callbacks.

Consider a simple program which interprets to sequence of actions on figure 5.2. The executable is a theoretical example of a program which lets two threads add 1000 to shared account balance *G_balance* each. Assume that the first thread forks the second thread at *line 1*. Then both threads execute the critical section between *lines 2 - 7*. The thread which updates *G_balance* last prints the final balance at *line 9* . With the provided algorithm the instrumentation can produce an executable of the form presented in figure 5.2.

The given example has 10 actions. We will describe how the instrumentation algorithm divides into transactional regions and add race detection callbacks. It does this by inspecting each action from action 1 to action 10 and producing output as shown on figure 5.2.

The first action is SYNCHRONIZATION and there is no open transaction already. Therefore, the algorithm at *lines 31- 40* appends the action to output and creates a complete, independent transaction by wrapping race detection callback for thread creation. The output is as shown on lines 1-4 of the output on Listing 2.

The second action is also a SYNCHRONIZATION operation. It constitutes a separate transactional region with this action enclosed together with race detection callback as as at lines 5 - 8 on Listing 2. The code which produces this portion of output is at lines 34 - 40. The similar output is experienced with the the lock release action at line 7 of input executable. Its output is between lines 20 - 23 of Listing 2.

Between lines 3 and 6 of the input executable there are three consecutive

GLOBAL_OPERATION actions. Since `MAX_CAPACITY` is set to 2, the algorithm, at lines 13 - 25, produces two transactional blocks as shown on lines 9 - 19 of Listing2. Since action 7 is a synchronization operation it ends the second transactional block making it contain only one global operation.

Actions 8, 9, and 10 are elements of TRANSACTION_BARRIER. Therefore, they reside outside transactional block. The portion of algorithm which handles this lies between lines 6 -12.

This can be transformed into the following code using the instrumentation algorithm with MAX_TSX_CAPACITY set to 2.

Assume that the sequence of actions were identified from a program in which a main thread forks a child thread at line 1 and both threads execute lines 2-8 at listing 5.1.

5.3 Evaluation

We evaluated our approach on five benchmarks from Splash-2 benchmark suite. The benchmarks we used are barnes, fft, lu_cb, lu_ncb, and radix.

As a proof of concept we manually annotated the benchmarks. We then applied the algorithm we proposed on section 5.2 to split program into consistency blocks and add TSX instructions and race detection function callbacks accordingly in a manner the algorithm operates. It should be noted that the same algorithm can be applied with the help of dynamic instrumentation tools like PIN.

Our instrumentation with annotation can be classified as coarse grain instrumentation as not every tiny piece of shared memory is instrumented in similar manner to dynamic instrumentation performed by tools like PIN or DynamoRIO. In most cases of the selected benchmarks input used was test or simsimall input.

For evaluation of approach we describe experimental setup, show the speedup results it achieves at best, and examine various factors which affect performance of our approach.

Listing 5.1: Example program before instrumentation

```
1 Create thread 2
2 Acquire AccountLock
3 Load G_balance to register local reg1
4 Add 1000 to reg1
5 Store reg1 value to G_balance
6 Load G_total to register local reg2
7 Release AccountLock
8 Compare reg1 and reg2 and jump to 10 if less
9 Call Print_func reg1
10 Return
```

Listing 5.2: Example program after instrumentation

```
1 Create thread 2
2 XBEGIN
3 race_check_callback
4 XEND
5 XBEGIN
6 race_check_callback
7 Acquire AccountLock
8 XEND
9 XBEGIN
10 race_check_callback
11 Load G_balance to local register reg1
12 Add 1000 to reg1
13 race_check_callback
14 Store reg1 value to G_balance
15 XEND
16 XBEGIN
17 race_check_callback
18 Load G_total to local register reg2
19 XEND
20 XBEGIN
21 race_check_callback
22 Release AccountLock
23 XEND
24 Compare reg1 and reg2 and jump to 29 if less
25 Call Print_func reg2
26 Return
```

Figure 5.2: Example of program actions before instrumentation and after instrumentation.

5.3.1 Experimental Setup

We performed experiments on an ordinary desktop machine with Haswell microarchitecture cores which have TSX instructions. The properties of the machine are as listed below.

Machine Properties Intel i5 Haswell Microarchitecture, 2 -core with hyperthreading, 4GB RAM, 2.4GHz clockspeed. Running Ubuntu 13.04 - 64bit Operating system.

Splash Benchmarks Used barnes, fft, lu_cb, lu_ncb, radix

Number of Runs, Input Size, and Program Threads For experimental results on section 4.2 and 4.3 are average of running hundreds of times setup benchmarks and their average values. We exercised with 2, 4, 8, 16 threads and transactional sizes ranging from 1 to 16.

5.3.2 Performance

We present performance with ideal number of threads as 4 on a averagely small program input. Graph 5.3 shows program slowdown bar graphs for three settings; first when the benchmarks execute without both the race detection and transaction instructions. Typically this is 1.0X slowdown. The second bar represents slowdown when it runs with race detection only and FastTrack meta-data are protected by individual locks. The third bar of each benchmark represents slowdown when benchmarks run in transactions and race detection executed simultaneously. Race detection meta-data in this case are not protected by individual lock, but rather by TSXs. For each benchmark there is bar for slowdown.

The second bars for each benchmark are tallest of all within any benchmark. This implies that protecting race detection meta-data imposes more slowdown. Use of TSXs utilizes necessary fine-grain protection of the data and necessary improvement is reached. With TSX we can achieve a maximum of 4.0X speedup as achieved at *fft* benchmark.

Benchmark	Adresses	Locks	Shared Reads	Shared Writes	Lock Operations	Barrier Operations
barnes	681634	68	64768018	24200474	34528	68
fft	1575331	2	20769077	13963385	8	28
lu_cb	66849	2	14560103	5681684	8	268
lu_ncb	65817	2	13463448	5694781	8	268
radix	2172676	2	13932783	7588112	8	44

Table 5.1: The statistics table showing execution properties of the benchmarks.

Table 5.1 shows the overview of operations statistics of the benchmarks used with their specified inputs. Addresses column shows number of unique memory addresses in a given benchmark. These addresses are accessed by 4 threads in this setting and total number of read and write operations on these addresses are shown on respective "shared reads" and "shared writes" columns respectively.

The locks column shows number of unique locks used by these the threads to protect some shared data depending on the benchmark. The lock operations column shows number of times in total these locks locks were acquired and released.

The last column the barrier operations shows number of barriers executed in the benchmark. Barriers are regarded as synchronization operations and have special effect on FastTrack algorithm.

5.3.3 Factors Affecting Performance

Transaction size Keeping number of application threads for each benchmark as four and increasing the number of program actions inside transactional block, the change in performance speedup is presented in figure 5.4. Ideally speedup slightly increases with increase in transactional size for most of the benchmarks. This is because as the size of transactions increase, total transactions per application decrease. This way the total cost of initializing transactions decrease. Initialization includes initialization of write buffers and write and read sets. However, if accessed data within a transaction is too sparse and from totally unrelated memory. The cache misses and false cache sharing increases aborts and this degrades performance as is the case for *barnes*.

When transactional size is one it means transactions contain single action each. The transactional size dominates the execution and hence performance comparably not high. However, increase in block size up to 4 almost there is linear increase in performance and increase is gentle.

However, transactions with sizes larger than 4 do not persist that linear increase in speedup because, due to program structure of the benchmarks by blocks contain

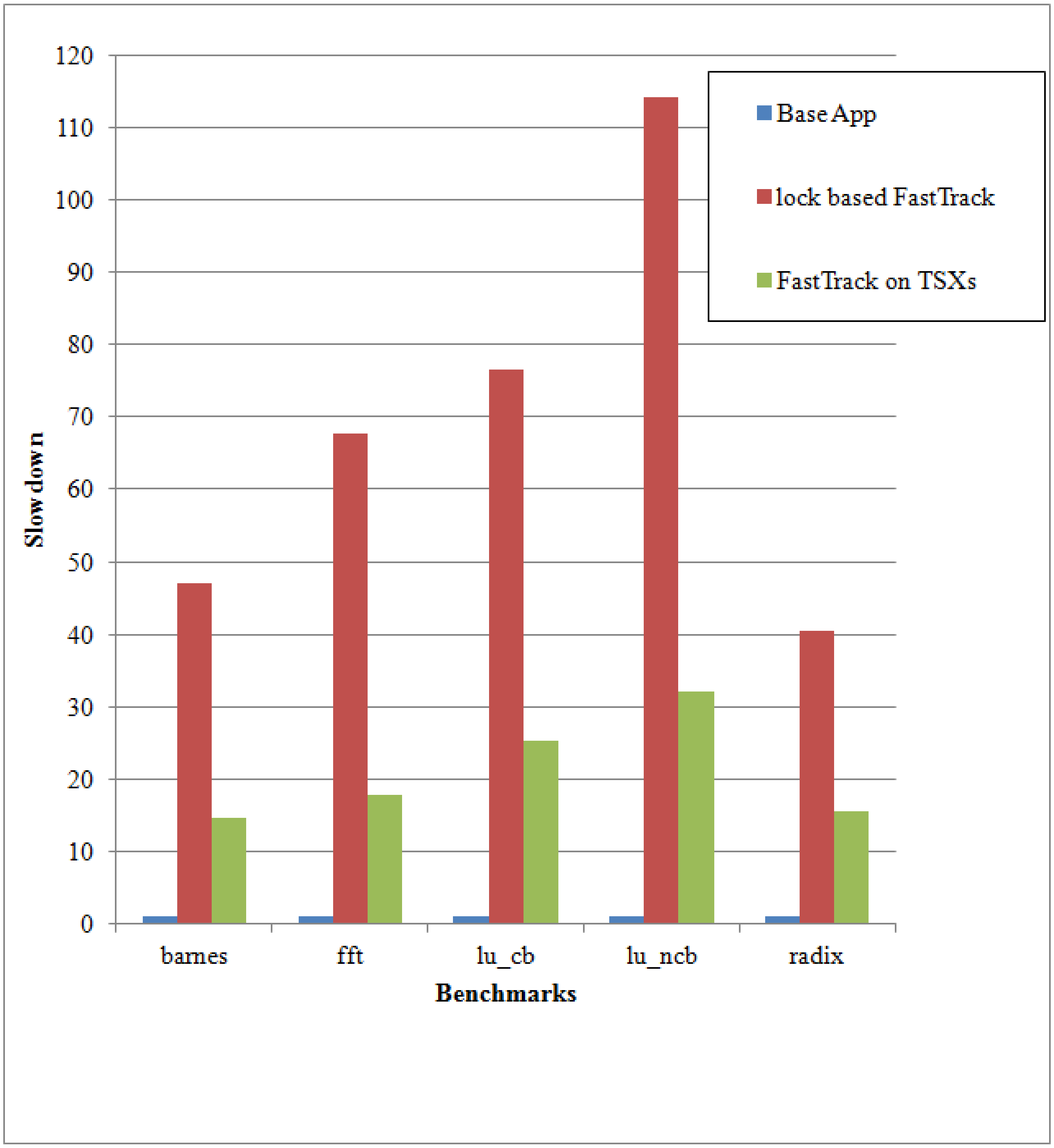

Figure 5.3: The graph showing performance slowdowns for race detection with FastTrack protected by fine-grain locking and FastTrack protected by Transactional synchronization extensions.

maximum number of actions.

Abort Rates Table 5.2 shows percentage of transaction executions that processors tried and the failed due to a number of reasons. In our implementation when a transaction fails it can be retried up to a fixed number of times before that block executes non-atomically. Therefore, the provided percentages show the rate at which transactions were tried and failed. These failing transaction trials contributed to slowdown therefore it is worth examining them. The higher the abort rate the lower speedup once can get when trying to gain performance of race detection using TSXs.

Number of Parallel Threads For a given application increase in number of parallel executing threads affect speedup of race detection. With fixed transactional block size to 4 we examined hour our algorithm performs on the benchmark when their number of executing threads increases.

Our results in figure 5.5 are partly influenced with the small number of cores of our experimentation machine. This means that an application with more than 4 threads context switches increase and so benchmarks aborts. That is why we experience increase in speedup up to 4 threads. Then we see a serious drop in speedup when application threads are greater than 4.

5.3.4 Correctness of our approach

On the algorithm we focus much on performance. We rely on the correctness of the FastTrack algorithm. Moreover, addition of TSX and fasttrack callbacks does not change program semantincs. At least this can be provide informally by the program output when benchmarks are run on three modes; without TSX and FasTrack, with FatTrack only, and with both TSX and FastTrack. Output from three modes show same results for all the benchmarks used.

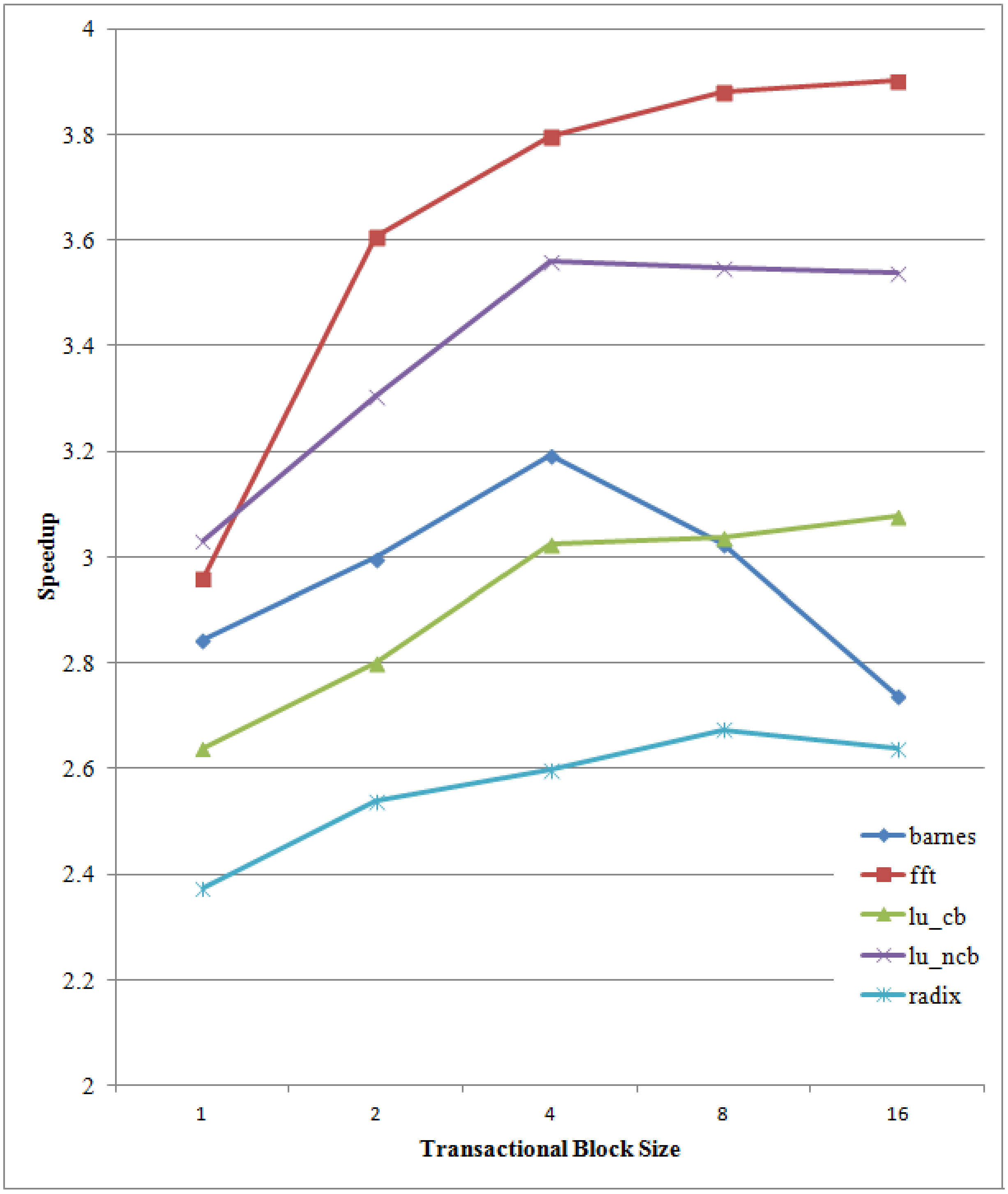

Figure 5.4: The graph showing effect in performance by changing size of transactional block

Benchmark	Transaction Block Size				
	1	2	4	8	16
barnes	0.484%	1.282%	1.392%	1.848%	2.582%
fft	0.182%	5.579%	20.492%	27.164%	1.334%
lu_cb	0.039%	0.133%	0.115%	0.133%	0.140%
lu_ncb	0.052%	0.286%	0.242%	0.239%	0.251%
radix	12.107%	66.274%	0.901%	11.430%	9.847%

Table 5.2: The statistics table showing abort rates by transactional block size.

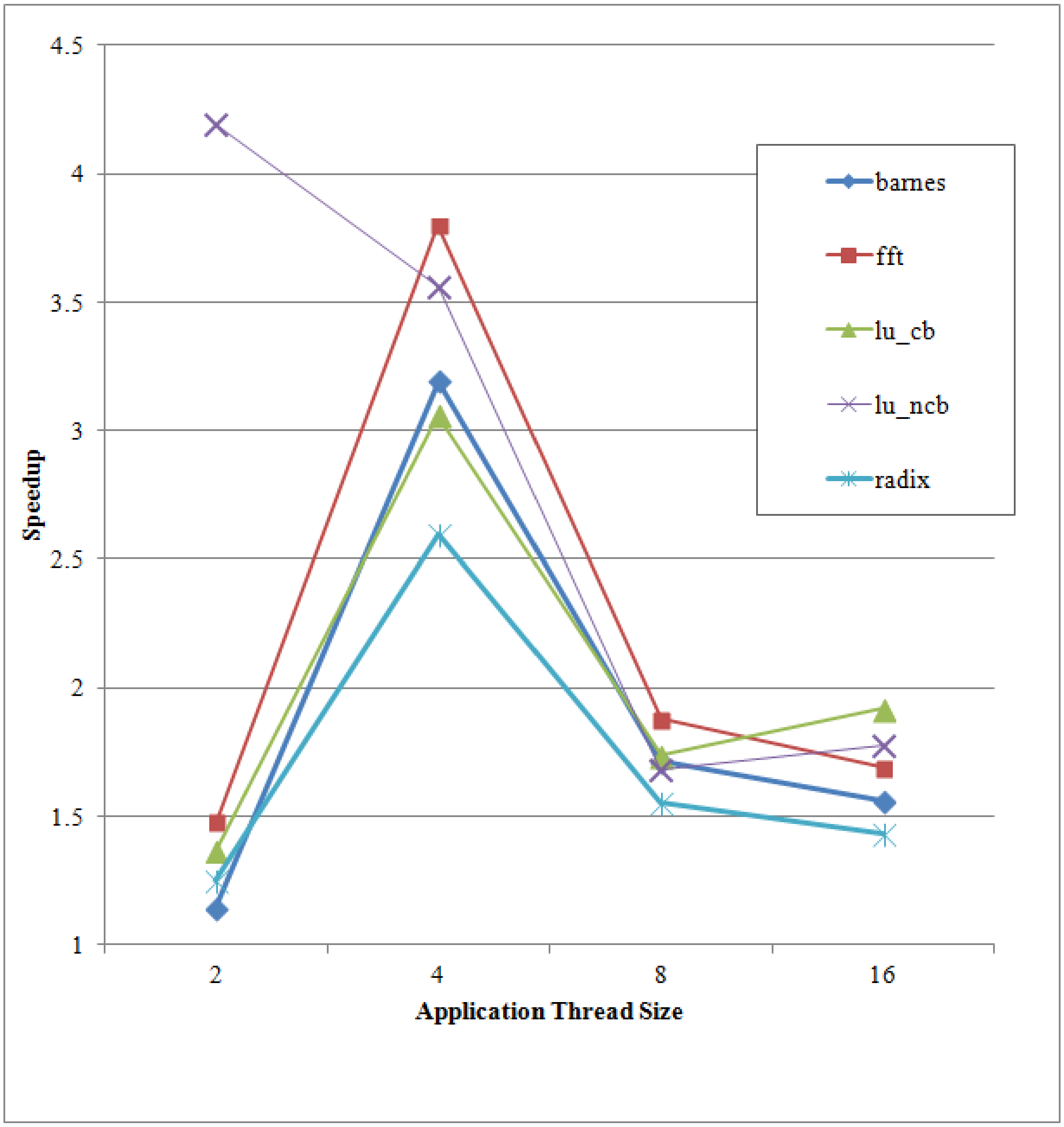

Figure 5.5: The graph showing performance speedup changes as number of application threads increase

Chapter 6

CONCLUSIONS

This thesis has has presented an number of contributions our project.

We have shown the approach to reduce runtime race detection using helper thread per application thread as adapted from STM^2 [14].

Moreover, we have showed that using software transactional memory techniques, recovery from the races can be achieved and prevented further by imposing extra protection on the racy data.

We have proposed an algorithm in chapter **??** to instrument a program into small transactions for race detection. Splitting a program into small transactions does not change semantics of the program and it does not affect precision of race detection.

We have proposed use of TSXs to aid race detection and have showed that it improves performance. We don't simply protect the race-detection metadata using transactions. However, we break down a binary into proper-sized chunks in order to minimize overhead. We present a proper use of of TSXs to achieve precise race detection with much reduced slowdown while maintaining soundness and precision of race detection algorithm used.

BIBLIOGRAPHY

[1] U. C. Bekar, T. Elmas, S. Okur, and S. Tasiran. Kuda: Gpu accelerated split race checker. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, London, England, UK, March 2012.

[2] E. Bodden and K. Havelund. Racer: effective race detection using aspectj. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 155–166, New York, NY, USA, 2008. ACM.

[3] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[4] D. L. Bruening. Efficient, transparent and comprehensive runtime code manipulation. Technical report, 2004.

[5] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proc. 1st Workshop on Architectural and system support for improving software dependability*, ASID '06, pages 63–65, New York, NY, USA, 2006. ACM.

[6] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 245–255, New York, NY, USA, 2007. ACM.

[7] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. *SIGPLAN Not.*, 44:121–133, June 2009.

[8] C. Flanagan and S. N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In S. Lerner and A. Rountev, editors, *PASTE*, pages 1–8. ACM, 2010.

[9] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is sc + ilp = rc. In *In Proceedings of the Twenty Sixth Annual International Symposium on Computer Architecture*, pages 162–171. IEEE Computer Society Press, 1999.

[10] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[12] Intel. Chapter 8: Intel transactional synchronization extensions. *http://software.intel.com/sites/default/files/m/9/2/3/41604*.

[13] Intel. Intel architecture instruction set extensions programming reference with intel tsx. *http://download-software.intel.com/sites/default/files/319433-014.pdf*.

[14] G. Kestor, R. Gioiosa, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. Stm2: A parallel stm for high performance simultaneous multithreading systems. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 221 –231, oct. 2011.

[15] A. Kleen. Lock elision in the gnu c library.

[16] I. Kuru, H. Matar, A. Cristal, G. Kestor, and O. Unsal. Parv: Parallelizing runtime detection and prevention of concurrency errors. In S. Qadeer and S. Tasiran, editors, *Runtime Verification*, volume 7687 of *Lecture Notes in Computer Science*, pages 42–47. Springer Berlin Heidelberg, 2013.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[18] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, C-28(9):690–691, 1979.

[19] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[20] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 37–48, New York, NY, USA, 2006. ACM.

[21] C.-K. 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. *SIGPLAN Not.*, 40:190–200, June 2005.

[22] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[23] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A case for an sc-preserving compiler. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 199–210, New York, NY, USA, 2011. ACM.

[24] F. Mattern. Virtual time and global states of distributed systems. In C. M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[25] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[26] S. Qadeer and S. Tasiran. Runtime verification of concurrency-specific correctness criteria. *International Journal on Software Tools for Technology Transfer*, 14(3):291–305, 2012.

[27] A. Raza. A review of race detection mechanisms. In *Proceedings of the First international computer science conference on Theory and Applications*, CSR'06, pages 534–543, Berlin, Heidelberg, 2006. Springer-Verlag.

[28] D. Sánchez, J. L. Aragón, and J. M. García. A log-based redundant architecture for reliable parallel computation. In *HiPC*, pages 1–10. IEEE, 2010.

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[30] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of*

the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, pages 369–384, New York, NY, USA, 2011. ACM.

[31] A. H. Vineeth Mekkat and A. Zhai. Accelerating data race detection utilizing on-chip data-parallel cores. In *International Conference on Runtime Verification (RV), 2013*, INRIA Rennes, France, 24-27 September 2013.

[32] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. Paralog: enabling and accelerating online parallel monitoring of multithreaded applications. In *Proceedings of the fifteenth edition of ASPLOS*, ASPLOS '10, pages 271–284, New York, NY, USA, 2010. ACM.

[33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.