TOLERATING ASYMMETRIC DATA RACES WITH MINIMAL HARDWARE SUPPORT

BY

SHANXIANG QI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Professor Josep Torrellas

Abstract

Data races are a common type of concurrency bug in parallel programs. An important type of race that has not received much attention is Asymmetric data races. In these races, the state of well tested, correct threads is corrupted by racing threads from external, typically third-party code. Current schemes to detect and tolerate these races are software based, and have substantial execution overhead.

This thesis proposes the first scheme to detect and tolerate asymmetric data races in hardware. The approach, called Pacman, induces negligible execution overhead and requires minimal hardware modifications. In addition, compared to past software-based schemes, Pacman eliminates deadlock cases. Pacman is based on using hardware address signatures to detect the asymmetric races. Processor, cache coherence, and protocol messages remain unchanged. We evaluate Pacman for all the SPLASH2 and PARSEC applications. Our results show that Pacman is effective and has minimal overhead.

To Father and Mother.

Acknowledgments

I would like to thank my advisor Prof. Josep Torrellas, who teaches me how to do research in computer science. He gives me many brilliant ideas during these three years and my work wouldnt be possible at all without his help.

I also want to thank my colleagues. Norimasa and Lois spends a lot of time helping me for evaluation part and give me many good advice on my thesis work. Abdullah answers a lot of my questions and explains to me about his work on data race detection.

Last I want to thank my family, who always encourage me and support every decision I make. I love you forever.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the development of multicore technology, parallel programming becomes more and more important. While at the same time, concurrency bugs are likely to take on a higher profile and become an even costlier problem. Consequently, it is crucial to continue developing more effective techniques to detect and fix the concurrency bugs.

A key type of concurrency bug is the data race. A data race occurs when two or more threads access the same variable without any intervening synchronization and at least one of the accesses is a write.

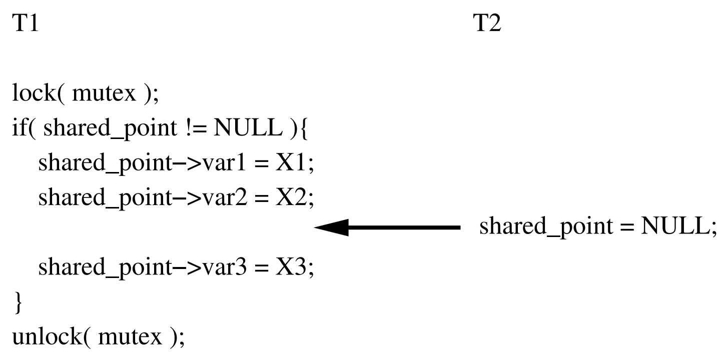
Since debugging data races is notoriously hard, there are a lot of works focusing on detecting the data race(e.g., [10, 5, 7, 9, 12, 13, 14, 15, 18, 19, 22, 24, 25, 27, 30, 31, 17, 11, 6]). However, not all data races would become bugs in the program, people may want to use data race to improve the performance of the program.

```
  T1                 T2

b=a;            while (flag!=true);

flag=true;           a=2;
```

Figure 1.1: Example for benign race

In the situation of Figure 1.1, even though a race occurs, it does not harm the correctness. Narayanasamy et al [12] showed that in Windows Vista and Internet Explorer, benign races happen rather common. With this fact, we only focus on data race which has high probability to harm the program, asymmetric race. We will give a detail definition in Chapter 2. Figure 1.2 shows an example of asymmetric data race, where T1 protects the shared_point in its critical section, and T2 modifies the shared_point variable without protecting it. When this situation happens, the program may lead to unexpected results.

Basically, we have two threads that access for the same shared variable from each: one thread is running in the critical section while the other one is not. Here, we define *safe thread* as a thread that accesses to shared variables with appropriate synchronization, and an *unsafe thread* as a thread that accesses to shared variables without lock protection or acquiring a wrong lock. Program developer can define which thread is



```
T1                                         T2

lock( mutex );
if( shared_point != NULL ){
   shared_point->var1 = X1;
   shared_point->var2 = X2;
                                   <-----  shared_point = NULL;
   shared_point->var3 = X3;
}
unlock( mutex );
```

Figure 1.2: Example of Asymmetric Data Race

safe and we will ensure the correctness of each critical section in safe thread. For the following discussion and experiments, we assume all the threads having critical section are safe threads which means we try to protect all the critical sections of the program.Note that we could only say one thread is a safe thread when it is executing a critical section.

Usually, asymmetric race will harm the correctness of the program since programmer put the shared variable inside one critical section. Besides, Microsoft also reports asymmetric races are common in software development projects [21]. They listed two reasons: First, usually a programmer's local reasoning about concurrency is correct. Errors due to taking wrong locks or no lock lie outside of the programmer's code, for example, in third party libraries. The second reason has to do with legacy code. For instance, a library may have been written assuming a single-threaded environment but later the requirement change to multithread programs. This requires all clients acquire locks before entry and release them on exit which could introduce races when some clients fail to put lock correctly.

Since asymmetric race is important and may harm the program, the goal of Pacman is not only detecting asymmetric race but also tolerating asymmetric race. In this scenario, Pacman ensures the correctness of the critical sections. Here, correctness means the value of shared variable inside the critical section should not be changed between the first access inside the critical section and the end of the critical section as shown in Figure 1.3

There are several previous approaches working on tolerating asymmetric races [21, 20]. However, Pacman is the first hardware approach to detect and tolerate asymmetric data races. Pacman overcomes the limitations of software implements such as deadlock, locking granularity, etc. Besides, with signature based hardware, Pacman can achieve very low overhead without changing processor, adding additional bits to cache, etc.

This thesis is organized as follows: Chapter 2 gives a background; Chapter 3 and 4 describe the Pac-

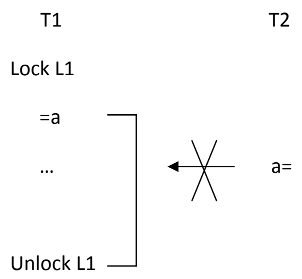


Figure 1.3: The goal of Pacman

man architecture and implementation; Chapter 5 evaluates Pacman; Chapter 6 discusses related work; and Chapter 7 concludes.

Chapter 2

Background

2.1 What is an Asymmetric Data Race?

Data races occur when two or more threads access to a shared variable without synchronization and at least one of them is a write. However, some concurrent applications intentionally insert symmetric data races into lock-free codes to get better performance as shown in Figure 1.1. In this thesis we focus on a harmful data race that may cause an incorrect behaviour in lock based programs: the asymetric data race.

An asymmetric data race happens in lock-based programs when the atomicity of a critical section in a safe thread is broken by an access to a shared variable from an unsafe thread. Figure 1.2 shows an example of asymmetric data race.

To characterize the situations where asymmetric data races may happen, we focus on the interleaving of accesses to a shared variable between a *safe thread* and an *unsafe thread* when the safe thread is in its critical section. We denote Rs and Ws as the reads and writes to the shared variable in a safe thread in its critical section, Xs as at least a read or write to the shared variable in a safe thread in its critical section and Ru and Wu as the reads and writes to the variable in an unsafe thread without synchronization.

The Xs-Wu-Xs interleaving produces an incorrect behavior in the safe thread. In this interleaving, an unsafe thread writes the shared variable between two accesses of the safe thread. This type of violations is dangerous, since the atomicity of the critical section is violated from the perspective of the safe thread. Note that if the unsafe thread writes the variable before the safe thread accesses for the first time to the variable (Wu-Xs) or after the last access (Xs-Wu), it is like the access was produced before or after the critical section, and the atomicity is not broken. Also, there is one exception in the interleaving Xs-Wu-Xs that not produce a race, the Ws-Wu-Ws interleaving.

There is other interleaving that produces a race, but it only affects to unsafe threads. In the Ws-Ru-Ws interleaving the safe thread executes correctly, but the unsafe thread reads an intermediate value of the shared variable that breaks the isolation of the critical section.

Also, there are some situations where the interleaving between safe and unsafe threads do not produce a race, such as *Xs-Ru-Xs*, with the exception of *Ws-Ru-Ws* interleaving of the previous paragraph.

In this paper we focus on preserving atomicity from the point of view of safe threads because the programmer reasons locally, and when establish the boundaries of the critical section, he hopes that the atomicity is preserved in his well synchronized code. We do not concern about the correctness of the unsafe threads because we only pretend to maintain the atomicity and correctness in critical sections.

Overall, to maintain the atomicity of the critical sections from the point of view of the safe threads, the key is taking care that unsafe threads do not write to shared variables while any safe thread is accessing to the same shared variable in its critical section.

The non-deterministic nature of the interleaving between accesses of different cores in a chip multiprocessor makes difficult to detect and debug asymmetric data races, and because of this is, a tolerant system could be very useful. Two proposals tolerate asymmetric races by replicating the data in a local copy with a software approach, Tolerace [21] and ISOLATOR [20]. They replicate shared data into local variables to isolate critical sections and prevent that its behavior is incorrect. The two mechanism detect and tolerate these races on the fly, due to a static analysis does not work with dynamic memory. These two approaches are essentially software approaches that introduce overheads due to replication and resolving races. In this work we approach the first hardware solution with low overhead to detect and tolerate asymmetric data races.

2.2 Hardware Signatures

Signatures are a low cost hardware solution to keep an unbounded number of addresses in a fixed space. The basic operations supported by a signature are to insert a new address, test for membership and clear. Signatures are composed by a long register that keeps a resume of addresses that are hash-encoding and accumulating with a Bloom filter [3]. Due to aliasing, a conflict can be detected in the test operation when no actual conflict exist (false positive), but no conflicts are missed. Signatures have been used in several multiprocessor designs to detect data dependences in thread-level speculation [4], transacional memory [29], race detection [10], code analysis [26] and others.

The first signature design [3] in Figure 2.1 is composed by a long register and several hash functions. A hash function gets an address as input, and returns a value in the [*0*,*m-1*] interval, where m is the size of the register. In an insert operation, each hash function sets one bit in the register. To test if an address conflicts with the signature, if the positions in the array getting by the hash functions are all 1, means that the address is in the signature. To clear the signature the register is reset.

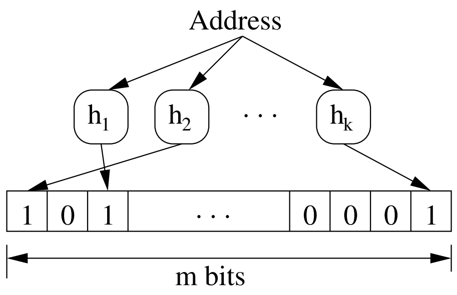


Figure 2.1: True Bloom Filter.

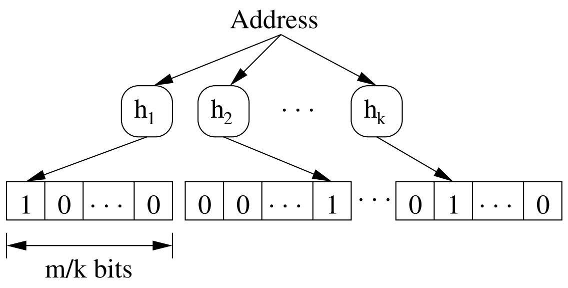


Figure 2.2: Parallel Bloom filter.

A more recent implementation of signatures that improve the bloom signature is the parallel bloom filters [23] in the Figure 2.2. The idea is similar, with the difference that each hash function operates in different ranges of the array, which permit separate the register in k single-port SRAMs instead of one multi-port SRAM. Then, this signature is composed by k Bloom Filters, each with a hash function. The insert operation hashes the address and inserts a bit in all k-filters. One address is represented in the signature if all bloom filters say it. The advantage of parallel signatures is that hash functions are more simple, and we can use simple-port SRAMS, and we can take up less space in die chip with similar performance.

In this paper we use hardware signatures with parallel bloom filter to detect and tolerate asymmetric data races.

Chapter 3

Tolerating Asymmetric Data Races

3.1 Overview of the Idea

All previous works on tolerating data races are based on software systems [21, 20]. Hardware approaches mainly focus on detecting the data races [10, 5, 14]. This makes Pacman the first hardware approach to not only detect but also tolerate the asymmetric data races.

The idea is to use hardware address signatures to automatically record the set of addresses accessed by the processor in a critical section. These signatures reside in a special hardware module called SigTable. We envision SigTable to be connected to the on-chip network. By using the cache coherence messages, SigTable updates the signatures, compares the signatures with the read/write operations from the other processors and occasionally sends NACK messages to stall some processor.

Pacman addresses several shortcomings of the existing data race tolerance schemes. First, from theoretical aspect, Pacman will not generate any new deadlocks or inconstancy behaviors that are not allowed by the original programs. Second, since we use hardware to check dynamic address on the fly, Pacman will have more accurate results than static alias analysis. Besides, Pacman does not rely on hardware and OS support for memory protection. Therefore, if the program uses fine-grained locks, Pacman would not cause any memory fragmentation. In fact, there is no need for compiler transformation or source code modification.

Unlike existing hardware data race detection schemes, Pacman has several advantages. First, Pacman does not need additional hardware to support rollback and re-execution. Second, although it is a signature based system, we have *not* found any false positive problem during our experiments. The first reason is that Pacman only records the read/write addresses inside the critical section and usually the size of the critical section is small. In addition, Pacman does not suffer from false-positive races due to false sharing. This is because Pacman encodes *word* addresses in signatures. Accesses to different words of the same line do not create collision. Finally, there is no need to modify the cache protocol or cache structure to support Pacman. We only need one simple cache modification to report the displacement events for the clean data on the network. The necessity of the modification is discussed in Chapter 3.4.

In the following, we are going to describe Pacman's operations in details under three stages: basic Pacman protocol, deadlock issues, and cache displacement. The actual implementation of Pacman is in Chapter 4.

3.2 Pacman Protocol

As shown in Figure 3.1, the key component of Pacman is a SigTable which is connected to the on-chip network. First, we introduce the Pacman protocol to detect asymmetric races, and then we explain an advanced protocol to face up more complicated situations.

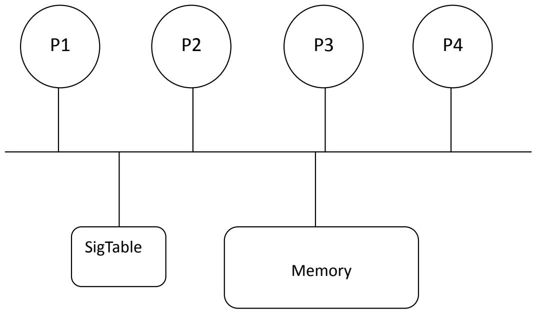


Figure 3.1: Overview architecture of Pacman

3.2.1 Basic Pacman Protocol

To detect asymmetric data races, the basic Pacman protocol needs a SigTable with several entries, each one of which is comprised of a *tid* and a *signature*.

- *tid* records the id of the thread that allocates the entry (we assume only one thread is running on one processor/core, tid is the same as processor/core id)
- *signature* records all read/write addresses inside critical section.

These two elements are enough to define the basic protocol, when a request arrives to SigTable:

1. For a lock acquire: Allocate a SigTable entry and set tid to the corresponding id.
2. For the read/write access inside the critical section: Insert the address in the corresponding signature
3. For a write access outside the critical section: If this access is conflicted with any signature in SigTable, it would be stalled by a NACK message sent by SigTable.

4. For a lock release: Deallocate the corresponding entry in SigTable.

When a program uses nested locks, Pacman manages it by flattening, which means that the *signature* will accumulate all the addresses inside the outmost lock and does not allocate a new entry to inner locks. To support this feature, Pacman needs a new element for each entry in SigTable - *counter*.

- *counter* records the current nested lock level.

With this element, the protocol is slightly modified in the release/acquire operations:

1. For a lock acquire: if there is no entry in SigTable for this thread, create a new entry in SigTable (the same as the previous protocol) and set the *counter* to 1. If Pacman has already created the entry for that thread, increases one to the *counter*.

4. For a lock release: decrement the *counter* in the corresponding entry, and if the *counter* is zero, deallocate the entry.

With this simple protocol, Pacman detects most of the situations of asymmetric races in programs.

3.2.2 Advanced Pacman Protocol

In previous software approaches [21, 20], deadlock may happen in some corner cases. To handle these deadlock situations, we define the advanced Pacman protocol.

In the advanced protocol, Pacman sometimes needs stall not only the threads outside the critical section, but also the threads inside the critical section, because data conflict between two concurrent critical sections. To manage this, Pacman needs to introduce a new element in the SigTable: the *stall_tid*.

- The *stall_tid* records the thread id which causes the current thread stalling, and it is set to -1 if the thread is not stalling.

Now, we need to extend the basic Pacman protocol to advanced Pacman protocol as following:

1'. For a lock acquire: If T0 tries to acquire a lock L, compare the mutex variable address of L with all signatures in SigTable. If there is no collision, then T0 keeps running. If L conflicts with thread T1's signature and SigTable shows that thread T1 is not stalled by any other threads, keep T1 running. If thread T1 is stalled by thread T2, delete thread T2's entry in SigTable, send a message to the correct processor so that thread T1 can keep running on it. After L is grabbed, put L to T0 thread's signature in SigTable and increment the corresponding *counter*. If this thread's entry is not in SigTable, create a new entry in SigTable.

2'. For the read/write access inside the critical section: If a thread T0 inside a critical section accesses to non-mutex address A, compare A with any signature in SigTable, if there is no collision, put A into T0's signature. If A conflicts with thread T1's signature, detect whether there exists a cycle among the stalled threads. If we detect a cycle in SigTable and T1 is part of that cycle, delete the signature of T1 so that the deadlock does not happen. If there is no cycle, T0 would be stalled and we will put T1 to T0's stall_tid.

3'. For a write access outside the critical section: If this access conflicts with any signature in SigTable, SigTable will send a NACK message to the processor to stall.

4'. For a lock release: Decrement the *counter* in the corresponding entry, and if the *counter* is zero, deallocate the entry.

As we can see, there are four rules to describe the protocol. Rules 3' and 4' are quite simple while rules 1' and 2' are a bit more complicated. We will discuss rules 1' and 2' in the following sections.

3.3 Deadlock Issues

As we described before, Pacman uses cache protocol to send NACK messages to instruct a processor to stall. Thus, Pacman adds extra stalling requests due to the collision with signature which could generate new dead lock situation. In this section, we describe two such cases to better understand the protocol.

3.3.1 Nested Lock

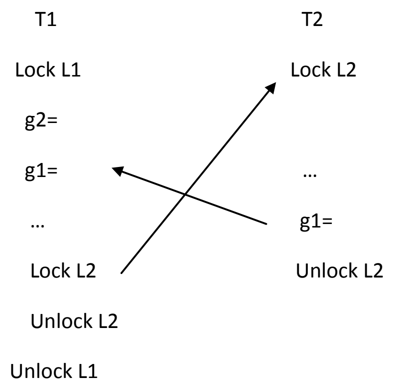


Figure 3.2: Nested lock example

The first example is about nested lock shown in Figure 3.2. In this example, at the first time, T1 grabs the lock L1 while T2 grabs the lock L2. After that, T1 issues write accesses to g2&g1 and put g2&g1's

addresses to its signature. When T2 tries to access g1, since it is conflicted with T1's signature, T2 should receive a NACK message. This behavior could generate a deadlock when T1 tries to grab lock L2 later. The problem is that T1 is stalled by T2 because T1 can not grab lock L2 and T2 is stalled by T1 due to signature confliction.

The simplest solution to avoid this deadlock is to delete one entry from SigTable when a conflict is detected between two critical sections. For example, in this case, when T2 detects the collision with T1, T1's entry will be deleted from SigTable to avoid the deadlock. But this solution has a disadvantage. Consider the following case in Figure 3.3:

T1	T2	T3
Lock (L1)	Lock (L2)	
g2=		
g1=	…	
…	g1=	
…	…	g2=
=g2		
Unlock (L1)	Unlock (L2)	

Figure 3.3: An example illustrating disadvantage of simple solution

Since T1's signature is deleted when T2 detects the conflict, even if there is no deadlock, the critical section of T1 looses the opportunity of protection.

A better solution for this example is to delete T1's entry when T1 tries to grab the lock L2 which is owned by T2. Compared to the simplest solution, the deletion of T1's entry is delayed. In general, the deadlock happens in the following way: there exists three threads Ti, Tj, Tk (Ti and Tk could be the same thread). Ti wants grab a lock which is owned by Tj and Tj is stalled by Tk. Basically, since Tj holds the lock, this solution wants to keep Tj running to avoid the deadlock. The solution is to delete Tk's entry from the SigTable so that Tj can keep running. This solution is used as rule1' in our protocol.

Plain variables in multiple critical sections

The second deadlock case could happen in multiple critical sections as shown in Figure 3.4.

In this example, T1 first puts g1's address in its signature while T2 puts g2's address in its signature. T2 will be stalled by T1 because g1 conflicts with T1's signature and T1 would be stalled by T2 due to g2.

This case is easier to solve than the previous deadlock case as in Chapter 3.3.1. The reason is that for nested locks, T2 must keep running since T2 has the lock which T1 wants. However, in this example, the

T1	T2
Lock L1	Lock L2
g1=	…
…	g2=
…	g1=
g2=	…
Unlock L1	Unlock L2

Figure 3.4: another example for deadlock

solution could delete either T1's or T2's entry to avoid deadlock.

As discussed in Chapter 3.3.1, the simplest solution (delete one signature in case of conflict) can also be applied to solve the deadlock problem while the chance to protect more critical sections is lost. A better solution is to try to detect the potential stalling cycle in SigTable. The example of Figure 3.4 shows that T1 is stalled by T2 and T2 is stalled by T1. If there is a cycle in SigTable, according to rule 2', we can delete one entry in SigTable.

3.3.2 Optimizations for the Protocol

A solution of the problems in Chapter 3.3.1 and Chapter 3.3.1 is to delay the time to delete the entry from SigTable. However, this solution cannot protect the critical section anymore after the deletion of the entry.

T1	T2	T3
Lock L1	Lock L2	
g1=	g3=	
…	g2=	
…	g1=	
g2=	…	
…	…	g3=
…	=g3	
Unlock L1	Unlock L2	

Figure 3.5: Disadvantage of the protocol

In the Figure 3.5, according to the advanced protocol, T2 is stalled by T1 because of g1's confliction. When T1 issues write access to g2, Pacman detects the stalling cycle between T1 and T2 and then deletes the entry of T2 in SigTable. When T3 wants to write g3, it should be an asymmetric race with T2 but since T2's signature has been deleted in the table, Pacman can not detect this data race.

To address this problem, instead of deleting the entry in SigTable, we keep it in SigTable. However, in order to avoid deadlock, Pacman does not stall the processor if detects the conflict with signature and there exists a stalling cycle. With this new approach, for Figure 3.5, when T1 issues the write operation to g2, it detects the conflict with T2's signature and the stalling cycle between T1 and T2. Now, Pacman does not delete the entry of T2, keeps T2's entry in SigTable but allows T1 writes to g2 for just once. After that, since T2's signature is in SigTable, when T3 issues write to g3, it would receive a NACK message.

The same idea can be applied to Chapter 3.3.1. In Figure 3.2, rather than remove T1's entry we allow T2's access of g1 and keep T1's entry in SigTable.

Since all the dead lock examples in Chapter 3.3 are buggy code with data race, we believe in real applications, it happens very infrequently. Depending on the performance or reliability, we can choose the advance protocol or optimization approach to tolerate asymmetric races.

3.4 Cache Displacement

To minimize the hardware modification to support Pacman, SigTable simply attaches to the on-chip network without modification of processor, cache coherent protocol, etc.

But in respect that Pacman can only monitor the events on the network, some read/write operations which are not be reported on the network may affect Pacman's correctness. Usually there are two kinds of read/write which cannot be detected in the network: write to a dirty data in the owner cache and read of a clean data from the owner cache. We would like to discuss the following four cases where Pacman cannot monitor the events.

1. A processor P1 writes to a dirty variable in its own cache outside the critical section. Since P1 will not put this write access to the bus, Pacman cannot capture this event to compare with the entries in SigTable. Fortunately, Pacman does not need to handle this case since only P1 owns this variable and the bus access does not conflict with any other signature.

2. P1 reads a clean variable in its own cache outside the critical section. Pacman only stalls the write accesses from unsafe thread, so that it allows any read accesses.

3. P1 writes to a dirty variable in its own cache inside the critical section. In this case, the signature must record the address of the variable, however the cache does not give the address to the bus at the moment. Instead, Pacman can insert this address into the signature later when other thread wants to access this dirty variable. For example, when P2 tries to write the dirty address in P1, Pacman can detect P2's write request first, and then P1 needs to write back the dirty data to the memory. Now, Pacman is able to detect this write back event through bus, put this address into the signature and send NACK to P2. If during P1's critical section, no other threads require this dirty variable, Pacman will still behave correctly without this address in the signature.

4. P1 reads a clean variable in its own cache inside the critical section. Basically, Pacman cannot detect this behavior by the bus. But when other thread requests this variable, if this variable is still in the cache, Pacman is able to put this address to the signature and send NACK to requester. The only problem is that this clean variable has been displaced inside the critical section and other thread requests this data. As shown in Figure 3.6, at the beginning, P1 reads clean data A1 so that we cannot detect this behavior by bus. After that, P1 reads A2 which need replace A1 in the cache. Now, since A1 is neither in the P1's cache nor in P1's signature, P2 can write to A1. To address this problem, we would require cache to report clean data displacement to the bus, so that we can put these "clean" addresses to the signature during the critical section. Obviously, not all these "clean" addresses are read during the critical section, we could have the false positive problem. But our experiment results show that cache displacement happens very rare inside the critical section, so this should not be a big issue.

```
  P1                                   P2
Lock L1
  = A1 (A1 is clean in the cache)
  …
  = A2(replace A1 in the cache)
  …                                    A1=
  …
Unlock L1
```

Figure 3.6: Example of cache displacement

Chapter 4

Implementation Issues

Pacman implementation requires a new hardware module connected to the memory bus and two new instructions. However, Pacman does not modify cache tags or data arrays in the cache hierarchy and does not need to change the software nor libraries.

4.1 Pacman Module in Bus

The Pacman module is a simple hardware module placed in the bus that snoops the cache coherence messages. As shown in Figure 4.1, Pacman module comprises a signature table and a controller.

In our current implementation, Pacman works with one thread per core on a 4 core chip-multiprocessor, and we use 2 bits *tid*, 1kbits *signature*, 2bits *stall_tid* and 10 bits *counter*. Note that the *counter* may overflow if the critical section has a lot of nested locks. To solve this problem, when the *counter* reachs its maximun value, the following acquire requests do not increment the *counter*. This situations may induce that the thread entry was deallocated before the critical section finishes, and not the whole critical section is protected from asymmetric races. Usually there is hardly deep nested locks in real applications, and this situation is very rare. In our current implement, we use 10 bits to support a maximun of 1024 levels of nested lock.

The Pacman controller manages all the incoming coherence messages on the bus and takes the appropriate decisions. When a incoming request arrives to Pacman module, the controller tests for conflicts in the signatures of the table entry which *tid* is different from the request's *tid*, and if a conflict exists, the Pacman controller resolves the race in the conflict resolution module (deleting the signature or sending a NACK to the conflicting core, see Chapter 3.1). Also, if the *tid* of the incoming message exists in the SigTable, the Pacman module inserts the address in the corresponding signature. As well, the controller uses the signature functional unit to perform basic operations in signatures (test, insert and clear operations).

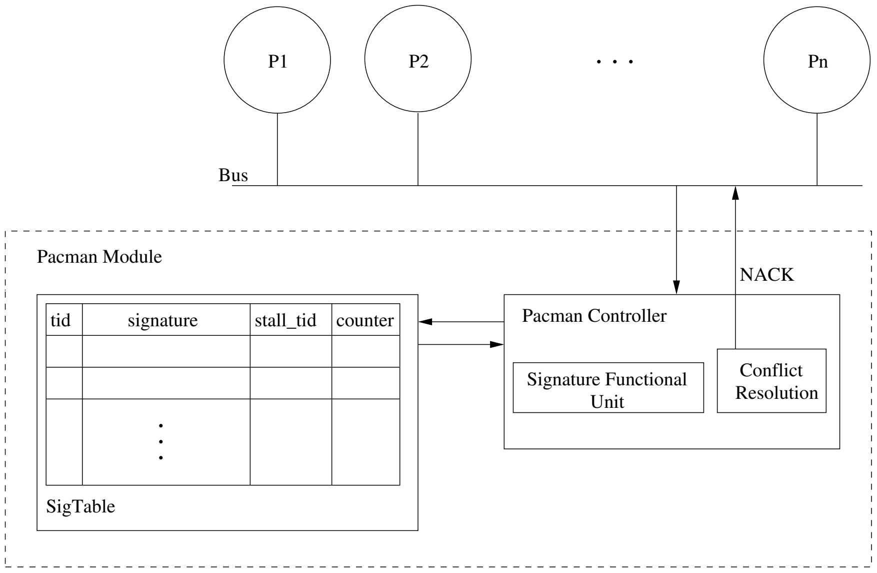


Figure 4.1: Pacman Module

4.2 New Instructions

Pacman adds two new instructions to the system to enable and disable the Pacman module (Table 4.1) at runtime.

Pacman Instruction	Description
pacman_on	Enables Pacman module
pacman_off	Disables Pacman module

Table 4.1: Instructions to manage Pacman module.

When any thread executes *pacman_on*, the cache controller send a message to Pacman module, and from that moment the Pacman module begin to collect addresses in critical sections from that core, and detect asymmetric races from other accesses. When *pacman_off* is executed in a thread, the cache controller sends a message to Pacman module, and from that moment, the Pacman module will not tolerate asymmetric races in that thread.

These two instructions are used in our approach to exclude serial regions of programs and shared libraries to be recollected by the Pacman module. They can reduce the overheads and signature pollution when Pacman is not necesary.

4.3 Other Issues

Current systems usually support two common issues that we do not mention until now: multithreading and OS thread migration. Multithreading is when a core supports multiple threads at the same time, and thread migration is when the OS migrates one thread from one core to another.

The previous discussion of Pacman assumes in single thread core without virtualization support. In this section, we would discuss how Pacman supports these two issues in a simple way.

4.3.1 Thread Migration Support

Although thread migration is uncommon when a thread is inside a critical section, Pacman is able to handle this situation.

The simplest solution to support thread migration in Pacman is to delete the entry of the migrated thread in the signature table and, from that moment, Pacman does not tolerate asymmetric races inside that critical section. However, we want that migrated threads keep safe. We propose that, when the thread is preempted, the OS marks that thread in the SigTable as preempted, and it flashes the cache to insert dirty lines in the signature. Note that, even when the thread is preempted, Pacman continues protecting it from asymmetric races because the signature is not deleted. When the thread is running again, the OS changes the old *tid* for the new *tid*, and Pacman continues running as usual. To support this feature, a new software interface is necessary in the OS to access the Pacman module.

4.3.2 Multithreading Support

Since our approach is based on snoop cache coherence requests, Pacman can not distinguish two requests from different threads in the same core. Pacman protects all threads in a multithreaded core under the same signature. This means that if two threads in their critical section are executed concurrently in the same core, Pacman uses the same signature for both, and it increments the *counter* for each acquire and decrements it for each release (in a similar way as nested lock, see Chapter 3.2). To support asymmetric race tolerance in this context, safe threads and unsafe threads should be scheduled in different multithreaded cores, to minimize the probability of asymmetric races among threads in the same core. This need extra OS support to schedule them in different cores.

Chapter 5

Evaluation

5.1 Experimental Setup

To evaluate the potential and performance of Pacman, we model Pacman by using Pin [8], a software framework for dynamic binary instrumentation, and run it on a real 4-processor shared-memory machine.

The Pacman model behaves as connected to a coherence bus to observe the memory access information. The default bit width of each signature is 1,024 bits and the default size of SigTable is 8 entries. Pacman simulator by Pin detects a pair of pthread_mutex_lock and pthread_mutex_unlock as a critical section and records the addresses of read or write accesses inside the critical section into a signature. To evaluate the overhead of the additional bus traffic due to clean cache-line displacements, we also model a 32-Kbyte L1 caches and their coherence bus with an MESI cache coherence protocol [16] for 4-core chip multiprocessor. When a snoop request from another processor displaces a cache line in the clean status, Pacman simulator records the displaced cache line address into a signature. In addition, Pacman simulator dynamically counts the number of simultaneous threads that start at pthread_create and end at pthread_join to simulate new Pacman instructions, *pacman_on* and *pacman_off*. If the number of simultaneous threads is one, Pacman simulator does not enable SigTable because Pacman only targets the critical sections in parallel region. Pacman simulator also ignores the critical sections in shared libraries.

Table 5.1 shows the default parameters used in Pacman simulator.

SigTable entries	4	L1 cache size	32 Kbytes
Signature size	1024 bits	L1 cache line size	64 bytes
tid	2 bits	L1 associativity	4
stall_tid	2 bits	Coh. Protcol	MESI
counter	10 bits		

Table 5.1: Default parameters in evaluation

We evaluate Pacman with all applications in SPLASH-2 [28] and PARSEC [2] benchmarks. To run SPLASH-2 as multi-threads applications, we use the modified SPLASH-2 [1] that implements the MACROS in SPLASH-2 with the pthreads library and defines a critical section as a pair of pthread_mutex_lock and pthread_mutex_unlock. We use all fourteen benchmarks in SPLASH-2 and their default inputs.

On PARSEC benchmarks, we use *pthreads* configuration to run them as multi-thread applications. The critical sections are also defined by a pair of pthread_mutex_lock and pthread_mutex_unlock in the pthreads library. We use all twelve benchmarks supported in the *pthreads* configuration in PARSEC and their input data are *simmedium*.

5.2 Characterization

Category	Application	#CS	% insts in CS	#insts /CS	Max insts in CS	#reads /CS	#writes /CS	#disps /CS	#sig addrs /CS	Max sig addrs in CS	Max d. nested locks
SPLASH-2 Kernels	cholesky	13,898	0.048%	18.6	168	5.9	2.3	0.000	3.9	12	1
	fft	36	0.268%	30.8	47	10.6	9.3	0.083	7.0	10	1
	lu/contiguous	276	0.003%	35.7	47	12.5	10.5	0.000	7.6	10	1
	lu/non_cont.	84	0.047%	33.8	47	11.8	9.9	0.048	8.4	11	1
	radix	78	0.003%	25.2	47	9.0	8.0	0.000	6.7	9	1
SPLASH-2 Apps	barnes	68,951	0.388%	118.1	1,892	40.1	29.3	0.000	15.5	89	1
	fmm	44,614	0.221%	142.1	252	54.7	27.9	0.000	21.2	36	1
	ocean/contiguous	5,264	0.028%	27.7	45	10.1	8.0	0.007	6.5	10	1
	ocean/non_cont.	5,144	0.028%	27.0	45	10.1	8.0	0.005	6.5	10	1
	radiosity	541,746	1.106%	11.2	1,233	4.4	2.5	0.000	3.2	163	5
	raytrace	189,918	0.316%	16.1	6,010	3.8	2.7	0.000	3.4	595	1
	volrend	72,592	0.019%	12.1	50	5.0	3.0	0.000	7.0	12	1
	water-nsquared	6,368	0.046%	49.8	51	34.0	12.2	0.000	19.9	22	1
	water-spatial	234	0.001%	18.2	49	6.8	4.7	0.021	5.4	11	1
PARSEC Kernels	canneal	8	0.003%	6,561.0	13,120	1,020.3	653.8	0.375	45.8	88	2
	dedup	35,862	0.101%	165.1	809	62.7	35.7	0.055	13.1	63	2
	streamcluster	104,208	0.002%	9.9	39	3.1	2.8	0.000	3.6	7	2
PARSEC Apps	blackscholes	0	0.000%	0.0	0	0.0	0.0	0.000	0.0	0	0
	bodytrack	16,341	0.008%	22.5	1,235	9.3	7.1	0.001	6.3	55	2
	facesim	8,754	0.001%	39.5	154	19.4	10.7	0.001	7.3	17	2
	ferret	141,615	0.750%	314.6	431,271	90.1	64.7	0.010	16.0	957	3
	fluidanimate	4,227,740	1.049%	12.0	39	6.6	3.5	0.000	6.1	12	2
	raytrace	60	0.000%	8.0	31	2.7	2.3	0.033	3.2	7	1
	swaptions	0	0.000%	0.0	0	0.0	0.0	0.000	0.0	0	0
	vips	14,010	0.006%	49.3	7,197	18.5	11.7	0.014	10.0	149	23
	x264	4,071	0.001%	18.9	43	9.0	4.8	0.024	5.9	9	1

Table 5.2: Characteristics

Table 5.2 gives the characteristics of the critical sections in all 26 applications. Column 3 lists the number of critical sections except serial region and shared libraries. Column 4 shows the percentage of the dynamic instructions in critical sections to the total executed instructions. Most benchmarks execute less than one percent instructions as critical sections of total executed instructions. Radiosity and Fluidanimate execute more than one percent instructions in critical sections since they repeatedly acquire and release locks inside a loop. Blackscholes and Swaptions do not have critical section in user routine. Their threads independently run without any synchronization. We see that the real application programs only have short periods that require to protect critical section from asymmetric race.

Column 5 and 6 show the average number of executed instructions per critical section and the maximum number of instructions in a critical section respectively. Most benchmarks execute less than one hundred instructions in critical section in average, however some benchmarks execute more than one thousand instructions in critical section. Canneal and Ferret especially shows distinctive characteristics. Some critical sections in both benchmarks have huge loops that last to release the lock. The size of critical section affects performance of Pacman, since activated Pacman may stall other threads having doubtful accesses until the end of the critical section.

Columns 7-9 list the numbers of read, write and clean cache displacements to shared memory per critical section respectively. Pacman observes these accesses on a bus and registers the addresses into SigTable. We can see that the number of clean cache displacements that cause extra traffics on a bus is less than one per critical section.

Column 10 shows the number of addresses recorded in SigTable per critical section and the maximum number of recorded addresses in a critical section are shown in column 11. Since SigTable keeps independent addresses on a bus to their signature bits by hash functions, the number of recorded addresses in the signature are less than the total number of accesses. These number of recorded addresses affects the possibility to cause an unnecessary false positive conflict on a signature when other threads access to shared memory and the address is converted to same signature bit by the hash function.

The last column shows the maximum depths of nested locks except shared libraries. The value more than one means that the benchmark has nested locks. The value one indicates that the benchmark has no nested lock and the value zero means no critical section in the benchmark. We can see that the maximum depth of nested locks is less than or equal to three except Radiosity and Vips. The data structure of Radiosity and Vips consists of a tree list and a sequential list whose node operation is executed recursively. Since Radiosity and Vips acquire and release locks during each node operation, they recursively create nested locks depend on the depth of the tree list and the length of the sequential list respectively. More nested locks than Vips might overlook asymmetric races even on Pacman, since Pacman does not acquire new signature for the critical section when *counter* for the nested locks is overflowed as discussed in section 4.1. However, we can see that Pacman hardly overlook asymmetric races even in such deep nested locks, because the periods in critical section is quite short shown in column 4.

5.3 Overheads

Table 5.3 shows the overheads of Pacman on SPLASH-2 and PARSEC benchmarks. Column 3 lists the number of executed Pacman instructions (*pacman_on* and *pacman_off*) that are inserted to exclude parallel sections and shared libraries from the bus observation of Pacman. We can see that the number of dynamic instructions are absolutely smaller than the total executed instructions shown in Table 5.2 and induces negiligible overheads.

Column 4 shows the bus traffic overhead connecting Pacman module. When processors in Pacman system

Category	Application	# Pacman instructions	%increase in traffic	%stall in worst case
SPLASH-2 Kernels	cholesky	4	0.000%	0.144%
	fft	4	0.050%	0.803%
	lu/contiguous	4	0.002%	0.008%
	lu/non_cont.	4	0.024%	0.142%
	radix	4	0.009%	0.010%
SPLASH-2 Apps	barnes	4	0.000%	1.165%
	fmm	4	0.000%	0.663%
	ocean/contiguous	4	0.003%	0.083%
	ocean/non_cont.	4	0.002%	0.085%
	radiosity	12	0.000%	3.317%
	raytrace	4	0.000%	0.949%
	volrend	4	0.000%	0.057%
	water-nsquared	4	0.000%	0.137%
	water-spatial	4	0.019%	0.002%
PARSEC Kernels	canneal	10	0.004%	0.008%
	dedup	13	0.004%	0.302%
	streamcluster	10	0.001%	0.007%
PARSEC Apps	blackscholes	5	0.000%	0.000%
	bodytrack	161	0.002%	0.023%
	facesim	75	0.001%	0.003%
	ferret	52	0.001%	2.251%
	fluidanimate	5	0.000%	3.148%
	raytrace	166	0.089%	0.000%
	swaptions	5	0.000%	0.000%
	vips	58	0.004%	0.018%
	x264	65	0.017%	0.004%

Table 5.3: Overheads

acquire and release locks, they issue *acquire* and *release* requests to SigTable attached on a bus. In addition, Pacman displaces cache line in clean status by a snoop request from another processor. We assume that the traffic size of *acquire*, *release* and *clean_displacement* is one byte, one byte and five byte respectively. We can see that the overheads of bus traffic are less than 0.1%. Since the number of dynamic instructions in critical section are small, Pacman limits the overheads of additional bus traffics to a minimum.

Finally, we estimate the stall cycle in the worst case shown in the last column. When an access from a thread conflicts on a signature that may cause an asymmetric race, Pacman stalls the processor until the signature for a critical section is released. If signature conflicts are occurred in all critical sections, the total stall cycle becomes the worst case. We can see that the stall cycles are less than one percent except Barnes, Radiosity, Ferret and Fluidanimate even in the worst case.

5.4 Handling Bugs

5.4.1 Existing Bug

Since SPLASH-2 and PARSEC are widely used benchmarks, usually they are well synchronized and Pacman cannot find asymmetric races in most applications. However, in PARSEC, we do find an asymmetric race from bodytrack application in Figure 5.1.

```
T1                                    T2

Lock                                  if (slack > 0){
...
while(nWakeupTickets == 0) {
...                                    nWakeupTickets++;
}
nWakeupTickets--;                     }
Unlock
```

Figure 5.1: An asymmetric race in PARSEC

In this case, T1 accesses nWakeupTickets before and after a while loop inside a critical section and T2 accesses nWakeupTickets outside the critical section. We suggest it is not necessary to put nWakeupTickets inside the critical section.

5.4.2 Inserted Bugs

We also modified some SPLASH-2 benchmarks to have intentional asymmetric race collisions on their shared variables. We concurrently create a thread that continuously write random values to shared variables without any locks. These accesses cause asymmetric race problems during the execution. In the tests on Pacman simulation, we see that Pacman detects and toleratess all asymmetric race cases.

Chapter 6

Related Work

There have been two significant proposals focused on detecting and tolerating asymmetric data races in lock-based concurrent programs.

One of them is ToleRace [21], the first proposal on deal with this type of races. In a critical section, ToleRace replicates the shared data on the fly, so that the thread that is in the critical section has an exclusive copy, and it continues reading and writing to this copy until it releases the lock. When the critical section finishes, ToleRace compares the data versions and determine if a race has occurred, and in that case, it may tolerate or reported it. This approach may produce sequentially inconsistent states in some situations.

The other proposal is ISOLATOR [20], that is similar to ToleRace, but detects and tolerates asymmetric races using data replication and page protection. When a thread is in its critical section, ISOLATOR replicates data in a local copy and protects the pages containing shared variables. If an unsafe thread tries to access to protected pages, a page protection fault is raised and the asymmetric race is detected. ISOLATOR solves the inconsistency problems of the previous proposal, but at the same time introduces new deadlock situations.

Pacman differ from the previous proposals in that is the first hardware approach that detects asymmetric races, it is not based on data replication to maintain correctness in critical sections, and introduce a low overhead hardware solution based on signatures. Pacman also solve the inconsistency problems of Tolerace and the deadlocks of ISOLATOR.

Chapter 7

Conclusions

This thesis proposed Pacman, the first hardware approach to not only detect but also tolerate asymmetric races. To address this problem, Pacman does not change the processor and cache coherence protocol messages. It uses hardware signature model to detect asymmetric races and use cache coherence to tolerate the races. Moreover, Pacman can solve the limitations which previous works have.

We presented the architecture of Pacman, an implementation, and its software interface. Application code remains unmodified. Our experiment showed that Pacman can run efficiency and achieve low overhead.

References

[1] The Modified SPLASH-2 Benchmarks Suite. `http://www.capsl.udel.edu/splash/`.

[2] C. Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton Univ., 2008.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[4] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *International Symposium on Computer Architecture*, 2006.

[5] J.-D. Choi et al. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[6] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, 2003.

[7] Intel Corporation. Intel Thread Checker. `http://www.intel.com/support/performancetools/threadchecker`.

[8] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[9] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *ASPLOS*, 1991.

[10] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. Sigrace: signature-based data race detection. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 337–348, New York, NY, USA, 2009. ACM.

[11] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, New York, NY, USA, 2006. ACM.

[12] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Programming Language Design and Implementation*, June 2007.

[13] R. Netzer and B. Miller. Detecting data races in parallel program executions. In *Workshop on Advances in Languages and Compilers for Parallel Computing*, 1990.

[14] R. Netzer and B. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.

[15] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symp. on Princ. and Practice of Par. Prog.*, 2003.

[16] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 284–290, New York, NY, USA, 1998. ACM.

[17] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331, New York, NY, USA, 2006. ACM.

[18] M. Prvulovic. CORD: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *ISCA*, 2006.

[19] M. Prvulovic and J. Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *International Symposium on Computer architecture*, 2003.

[20] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Isolator: dynamically ensuring isolation in comcurrent programs. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 181–192, New York, NY, USA, 2009. ACM.

[21] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 173–184, New York, NY, USA, 2009. ACM.

[22] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM TOCS*, 1999.

[23] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *International Symposium on Microarchitecture*, 2007.

[24] S. Savage et al. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[25] Sun Microsystems. Sun Studio Thread Analyzer. `http://docs.sun.com/app/docs/doc/820-0619`.

[26] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. Softsig: software-exposed hardware signatures for code analysis and optimization. *SIGARCH Comput. Archit. News*, 36(1):145–156, 2008.

[27] C. von Praun and T. R. Gross. Object race detection. In *Conf. on Obj. Oriented Prog., Sys., Lang., and App.*, 2001.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22th Int'l Symp. on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.

[29] L. Yen et al. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA*, 2007.

[30] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Symp. on Operating Systems Principles*, 2005.

[31] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *HPCA*, 2007.