DREM: ARCHITECTURAL SUPPORT FOR DETERMINISTIC REDUNDANT EXECUTION OF MULTITHREADED PROGRAMS

by

Stan Kvasov

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2009 by Stan Kvasov

Abstract

DREM: ARCHITECTURAL SUPPORT FOR DETERMINISTIC REDUNDANT EXECUTION OF MULTITHREADED PROGRAMS

Stan Kvasov

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2009

Recently there have been several proposals to use redundant execution of diverse replicas to defend against attempts to exploit memory corruption vulnerabilities. However, redundant execution relies on the premise that the replicas behave deterministically, so that if inputs are replicated to both replicas, any divergences in their outputs can only be the result of an attack. Unfortunately, this assumption does not hold for multithreaded programs, which are becoming increasingly prevalent – the non-deterministic interleaving of threads can also cause divergences in the replicas.

This thesis presents a method to eliminate concurrency related non-determinism between replicas. We introduce changes to the existing cache coherence hardware used in multicores to support deterministic redundant execution. We demonstrate that our solution requires moderate hardware changes and shows modest overhead in scientific applications.

Acknowledgements

First, I would like to thank Professor David Lie for his countless contributions, guidance, and support for this project. His encouragement and expertise helped formulate many of the ideas found in this work.

I would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) and the University of Toronto for providing continued funding for my degree.

I would also like to thank my fellow graduate students: Lionel Litty, Tom Hart, James Huang, and Lee Chew. I would also like to thank members of the Security Reading Group(SRG) and Professor Ashvin Goel for their comments and constructive criticisms of my work.

Finally, I want to thank my parents, Olga and Alexei, and my girlfriend, Annie Jekova, for their advice, support, and inspiration.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Despite growing awareness and concern about security compromises, software developers continue to produce programs with vulnerabilities, which Internet miscreants continue to exploit. The most severe vulnerabilities allow an attacker to arbitrarily overwrite and corrupt memory locations in a program's address space. The most prominent example of such an attack is the buffer overflow [2]. According to the Common Vulnerabilities and Exposures (CVE) database[1] there were over 600 buffer overflow vulnerabilities reported in 2008 and 71 have been reported in the first 3 months of 2009 alone. Each of these vulnerabilities can potentially allow an attacker to hijack a program and usurp privileges that she should not have.

There have been many attempts to eliminate the presence of memory corruption vulnerabilities or mitigate their effects [9, 13, 31]. One of the more promising attempts is address space layout randomization (ASLR) [6, 15]. Traditional memory corruption vulnerability exploits rely on the attacker knowing the addresses of sensitive objects in memory. ASLR makes these addresses random, thus providing vulnerable programs with probabilistic protection against attacks. ASLR does not require source code and poses essentially no run time overhead. Unfortunately, Shacham et al. [29] demonstrated

[1]http://cve.mitre.org/

that because of constraints on the placement of objects in a process' address space, there is not enough diversity to stop an attacker from exhaustively guessing the address space layout. To make ASLR protection immune to guessing, N-Variant [11] redundantly executes two replicas with different address space layouts and compares their behavior. However, N-Variant does not support multithreaded programs. Since N-Variant does not impose any constraints on the interleaving of threads in the replicas, the non-deterministic interleaving of threads can cause replicas to diverge even when no attack is occurring.

While there have been attempts to remove concurrency related non-determinism through software methods, this can result in performance slowdowns as high as 800%, which are too great to be usable in practice [14]. Similarly, lockstep execution of processors in hardware can provide deterministic redundant execution, but also carries a performance penalty as stalls in one processor force stalls to occur in the other processor [5]. Instead, we propose *Deterministic Redundant Execution on Multicores (DREM)*, which is able to remove concurrency related non-determinism with only 23% run time overhead. DREM allows cores in both replicas to execute independently and only constrains execution on inter-thread communication events. DREM records the order of inter-thread communication in one replica, called the *leader* and replays it in the other replica, called the *follower*. To record inter-thread communication efficiently, DREM leverages the cache coherence protocol, and thus only requires a small amount of new hardware to be added to a multicore processor.

1.1 Contributions

DREM is the first system to support redundant execution of diversified multithreaded applications on multicores by extending the baseline cache coherence protocol. All previous approaches do not support multithreaded workloads or rely on developer annotations to outline the sequential regions in applications. In addition, DREM shows additional use cases for existing work in multithreaded debugging. Previous work only focused on race-recording and offline replay. DREM combines the two techniques to record races in one application context and simultaneously replays them in another diversified context. Finally, we are the second researchers to publish the overheads of replaying races using the deterministic debugging techniques. Previous work focused on the overheads of race-recording and only one work evaluated the overheads of replay [20].

1.2 Thesis structure

The structure of this thesis is as follows. Chapter 2 provides the relevant background for the reader. Chapter 3 gives high-level overview of DREM and establishes the security guarantees that DREM can provide. Chapter 4 describes the race-recording and replay hardware and Chapter 5 outlines the implementation details of DREM in cycle-accurate MIPS simulator. Chapter 6 evaluates DREM using the SPLASH-2 [34] scientific application suite. Finally, related work is introduced in Chapter 7 and we conclude in Chapter 8.

Chapter 2

Background

This chapter gives an overview of the vulnerabilities that DREM gives protection against using a technique called redundant execution. Later in the chapter we describe memory coherence and consistency which are necessary to grasp DREM's redundant execution technique.

2.1 Memory Corruption Vulnerabilities

Memory corruption vulnerabilities constitute a general class of vulnerabilities that can allow an attacker to corrupt sensitive application data or give an attacker the ability to execute arbitrary code. DREM leverages address space diversification to provide protection against the latter type of attacks. In this section, we show how an attacker can use memory corruption vulnerabilities to execute arbitrary code.

Attackers exploit memory corruption vulnerabilities to make controlled modifications to locations in a program's address space. Two steps must be taken by an attacker to exploit an application. First, the attacker must find a vulnerability in an application that allows her to overwrite a code pointer. The ability to alter a code pointer allows the attacker to divert normal control flow of an application to achieve the attacker's goal. There are numerous programming flaws that allow code pointer overwrites. The most common type is the buffer overflow. Buffer overflow vulnerabilities occur when a program, written in a non-type safe language such as C, copies an input buffer from an untrusted source into a target buffer without checking that the target is large enough to hold all the contents of the input buffer. If the input buffer contains data that is indeed longer, the copy will ”overflow” the end of the target buffer and overwrite whatever data is located after the buffer. Another common way an attacker can overwrite an arbitrary location in memory is by using a format string vulnerability. These types of vulnerabilities arise when an application lets an attacker specify a format string used by string and I/O functions in C [10]. By cleverly specifying format tokens, an attacker can overwrite any location in memory with an arbitrary value.

Second, the attacker must provide malicious code that the overwritten code pointer in step one will point to. The attacker has two options - either find a buffer in an application that can store the malicious code or find existing code in the application which can serve the attacker’s purpose [7]. When the vulnerable application dereferences the changed code pointer, the control flow is transferred to the malicious code. Typically, it spawns a shell and gives the attacker control of a machine with the privileges of the vulnerable application.

2.2 Redundant execution applications

Redundant execution is a technique that involves running an application in multiple contexts by executing each instruction multiple times. If the inputs to the application are deterministic and constant across all contexts, then the outputs are guaranteed to be the same. Using this assumption, redundant execution can provide fault-tolerance against transient hardware faults in a processor. Transient faults occur when a cosmic ray strikes and alters voltage levels that represent data values [33]. A transient hardware fault that affects some, but not all, contexts is detected since it causes a divergence in the outputs. The divergence causes the hardware to roll-back and re-execute the violating instruction across all contexts.

Redundant execution can also be used to provide security to applications. Diversity can be introduced to the application contexts in such a way that an attacker cannot successfully exploit all contexts. The most recent work in this area is N-Variant [11] and Replicant [25]. These systems use address space diversification across application contexts to give protection against memory corruption attacks. For an attacker to exploit a memory corruption vulnerability she must be able to change the value of a pointer to an object or instruction of her choosing. These systems make this impossible by executing two or more replicas of the program with different address space layouts. This means that every corresponding instruction will be located at a different address in the replicas. Since the same malicious input is sent to all replicas, it is impossible for the attacker to change the pointer to point to the same corresponding instruction in all replicas – while the corresponding instructions will be located at different addresses, the pointer will be overwritten by the same value in both replicas. As a result, execution will be directed to different instructions in the replicas resulting in a divergence in instructions being executed. This divergence is detected and the application is shut down.

The previous approaches in redundant execution for security largely ignore multi-threaded software. Multithreaded applications pose a challenge for such systems because inter-thread communication is inherently non-deterministic. The non-determinism arises because threads execute at arbitrary rates and make racy shared memory accesses. We illustrate this form of non-determinism with an example. As shown in Figure 2.1, suppose we are executing two application threads redundantly. In one execution context, thread #0 acquires a lock first, increments a shared variable, and prints its thread id and the value of the shared variable. Thread #1 acquires the lock afterwards and performs the same actions. In a redundant execution context, since the threads can execute at arbitrary rates, thread #1 arrives at the lock first. The different order of arrivals in the

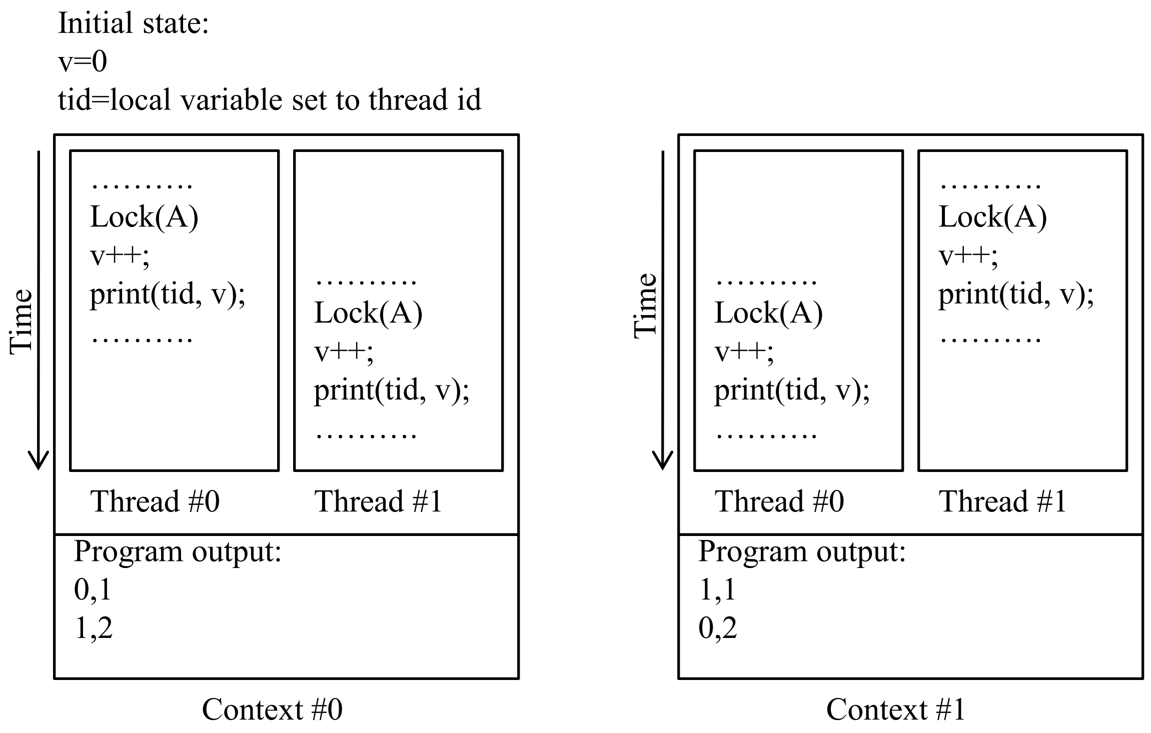


Figure 2.1: Redundant execution race example

two contexts causes a difference in the program output. Under DREM, this divergence gets labeled as an attack. To eliminate this form of non-determinism the outcomes of all racy shared memory accesses must resolve identically across all application contexts.

DREM's contribution is a hardware extension to a processor's cache coherency hardware that records the outcome of racy shared memory accesses in one execution context and replays the outcome of races in the redundant execution context.

2.3 Cache coherence

Current multicores are connected by a shared bus and utilize multilevel memory hierarchies with private and shared caches to increase locality and reduce their memory bandwidth requirements [17]. Private caches introduce a problem of coherence since they allow shared data to be replicated. If a single core decides to change shared data that is loaded in multiple caches, then a mechanism must exist for other caches to receive the newer updated value. Otherwise, the other cores will observe the old value in their cache.

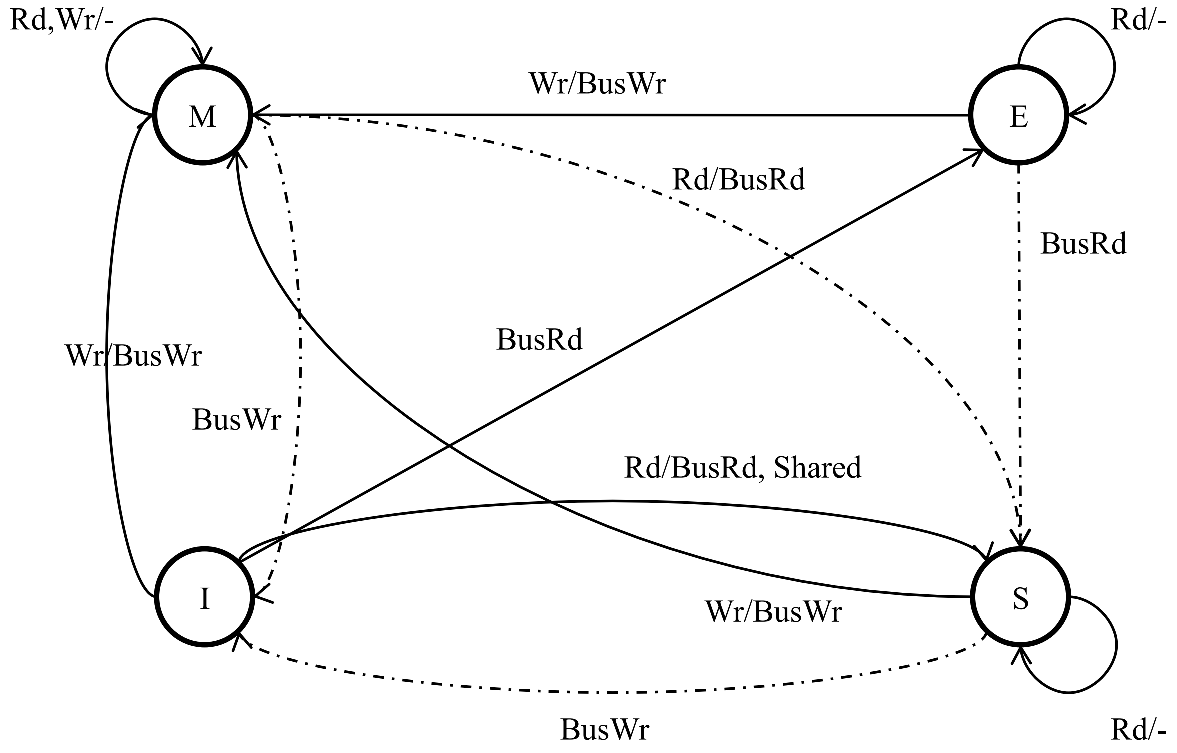


Figure 2.2: MESI state machine

To address the coherence problem outlined above, multiprocessors and multicores use hardware cache coherence. There are two main classes of coherence protocols: directory and snooping. Directory coherence uses a global directory to keep track of which processors are caching data and the state of that data. Directory coherence is generally used in large scale multiprocessors and not inside multicores. Moving to directory coherence has disadvantages since it requires more hardware and increases the chip's complexity. Snooping coherence utilizes a shared bus between cores that is used to broadcast updates to global shared data. Each core snoops on the shared bus to see whether it has cached a shared block and whether it needs to perform any action to ensure coherence. Today's multicores rely on bus-based snooping-coherence. Since DREM is aimed towards today's multicores, we assume the same type of coherence.

A common snooping protocol used in multicores is a four state (MESI) write-back invalidation protocol. DREM uses the messages sent by MESI coherence to record memory dependencies between cores. Hence, to understand DREM's design, we outline the MESI protocol. MESI uses cache line granularity to tag each cache line with a state. The four possible MESI states signify whether the cache line is shared by one or more cores (S), whether a line is dirty (M), whether it belongs exclusively to one core (E), or whether the line is invalid (I). The state transitions are initiated by bus messages, read and write, and each core's memory reads and writes. A simplified state machine for MESI is shown in Figure 2.2. The signals *Rd* and *Wr* are local reads and writes performed by a core. The signals *BusRd* and *BusWr* are broadcast on the shared bus. The filled line represents state transitions that are performed by a core initiating a memory access. The dashed lines represent state transitions in response to broadcast bus messages.

Note that MESI minimizes memory accesses by performing cache-to-cache transfers whenever possible. Since caches are built from faster SRAM memory, and memory from DRAM, the caches can supply data more quickly [12]. DREM utilizes cache-to-cache transfers to propagate some dependency information as will be shown later.

2.4 Memory consistency

A memory consistency model formally specifies how shared memory appears to a programmer [1]. For example, memory consistency places restrictions on what values read returns with respect to writes. There are numerous models ranging from the restrictive sequential consistency to more permissive release consistency. These models differ in how much pipelining and buffering of shared memory reads and writes they allow [16]. For example, sequentially consistent execution appears to all cores as some interleaving of execution of processes in program order. This model is restrictive but it is convenient for use with race recorders because it guarantees that the observed order of memory accesses during record and replay is identical. As a result, DREM's race recorder assumes sequential consistency. Other models, like release consistency, relax the ordering between reads and writes to different memory locations. This memory model requires special instructions such as acquire and release to synchronize between cores [1]. We note that DREM has a limitation since it does not support current multicores which use weaker memory models such Total Store Order. DREM can be extended to support weaker memory models by recording all memory re-ordering or by moving to a memory model which appears sequentially consistent but executes at release consistency speeds [8].

Chapter 3

Deterministic Redundant Execution for Security

The previous chapter described the challenges involved in redundantly executing multithreaded applications. Here, we give a high level overview of DREM and describe how it eliminates non-determinism related to multithreading applications. Later in the chapter we describe what type of attacks this form of redundant execution can prevent.

3.1 Deterministic execution

Multithreaded applications running on multicores introduce non-determinism since each thread makes shared memory accesses and creates data dependencies with other running threads. The non-deterministic data dependencies are called memory races because they are not constrained by program execution to occur in a predefined order. To enable redundant execution on multicores, we must ensure that all races resolve identically across all execution contexts.

DREM supports redundant deterministic execution by recording all cross-node data dependencies between cores in one execution context and replays these dependencies on a diversified execution context. DREM chooses one application context as the leader and

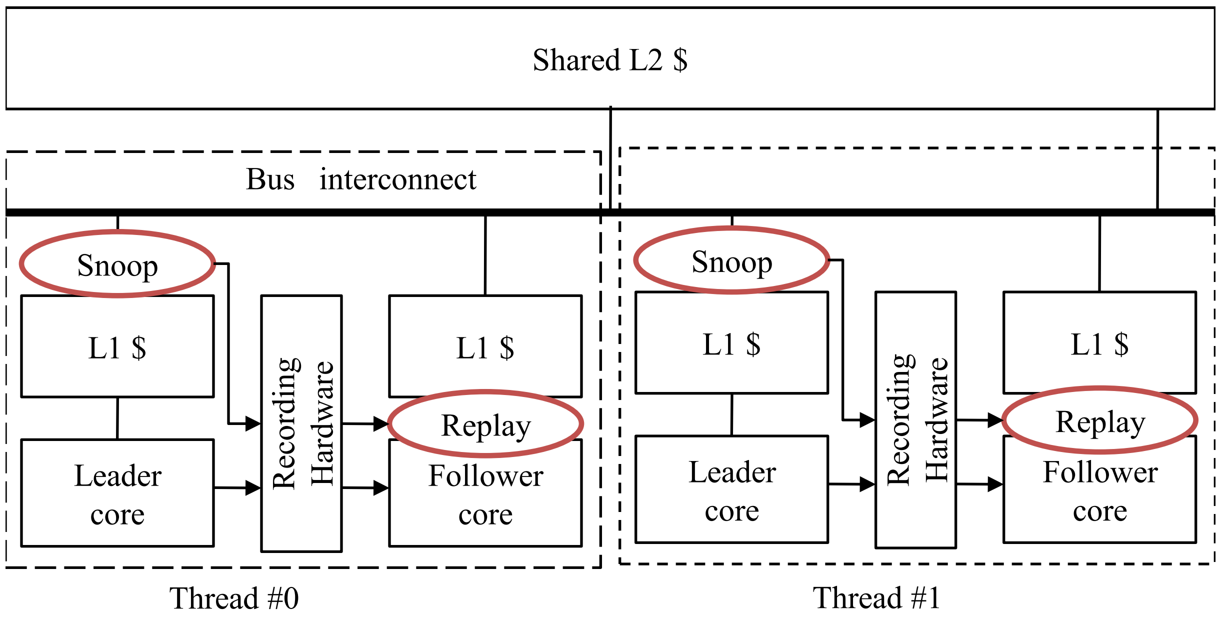
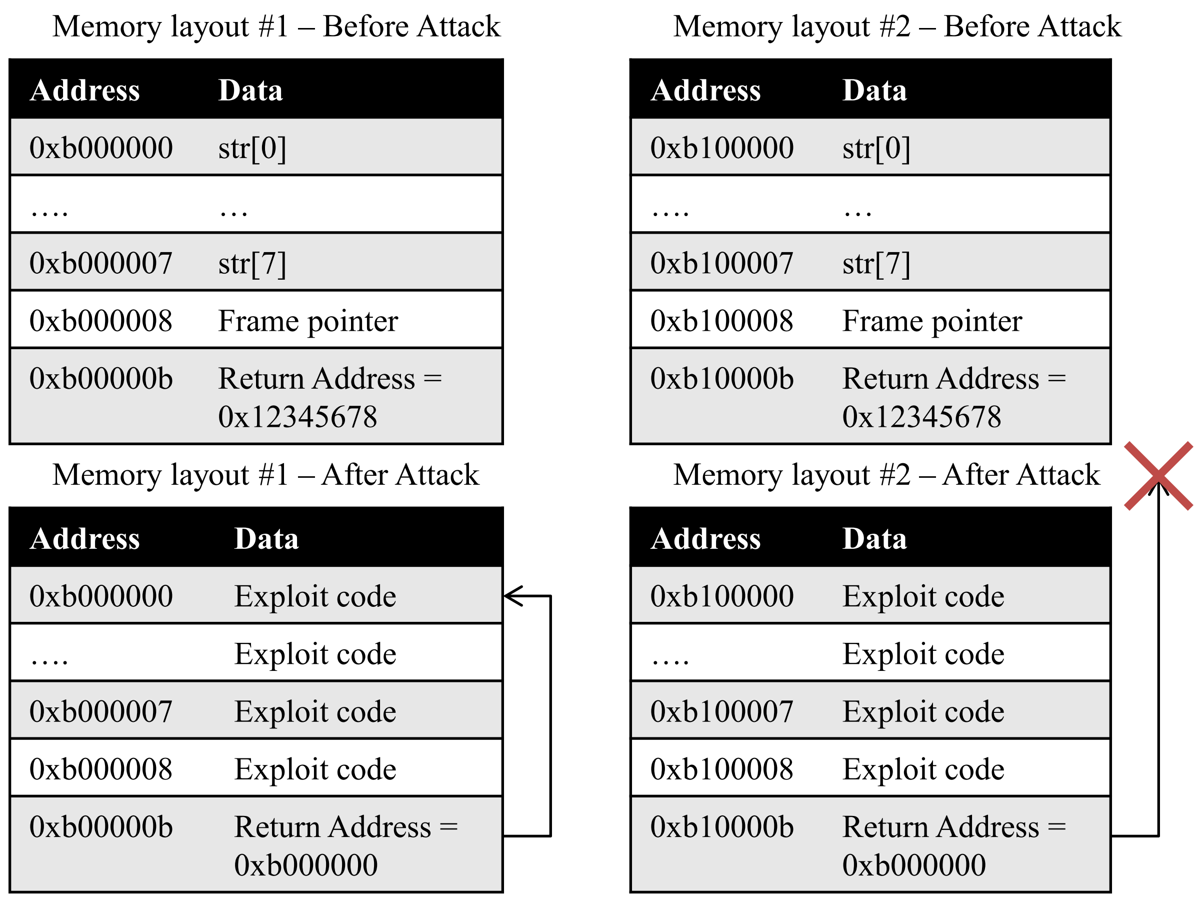


Figure 3.1: DREM 2-leader, 2-follower example

allows it to run ahead of a trailing context. The leader context is run on a set of cores called the leaders and the trailing context is run on a set of follower cores. Corresponding threads from each context are scheduled by the operating system on adjacent leader and follower cores.

By letting one context run first, DREM is able to monitor the bus interconnect for MESI coherence messages that expose almost all cross-node data dependencies. The dependencies not exposed by MESI are captured using transitive reductions and a slight modification to MESI as described in the next chapter. These data dependencies are stored in small hardware buffers that are fed to the follower cores. As the followers execute, they closely trail the leader cores in the instruction streams and schedule all memory accesses to comply with the recorded data dependencies. This scheme forces all memory races to resolve identically in both execution contexts eliminating all the multithreading related non-determinism. The high level diagram of redundant execution of two threads on two leaders and followers is shown in Figure 3.1.


Memory layout #1 – Before Attack
Memory layout #2 – Before Attack
Memory layout #1 – After Attack
Memory layout #2 – After Attack

Memory layout #1 – Before Attack

Address	Data
0xb000000	str[0]
….	…
0xb000007	str[7]
0xb000008	Frame pointer
0xb00000b	Return Address = 0x12345678

Memory layout #2 – Before Attack

Address	Data
0xb100000	str[0]
….	…
0xb100007	str[7]
0xb100008	Frame pointer
0xb10000b	Return Address = 0x12345678

Memory layout #1 – After Attack

Address	Data
0xb000000	Exploit code
….	Exploit code
0xb000007	Exploit code
0xb000008	Exploit code
0xb00000b	Return Address = 0xb000000

Memory layout #2 – After Attack

Address	Data
0xb100000	Exploit code
….	Exploit code
0xb100007	Exploit code
0xb100008	Exploit code
0xb10000b	Return Address = 0xb000000

Figure 3.2: Attack example

3.2 Security guarantees

For an attacker to exploit a memory corruption vulnerability she must be able to change the value of a pointer to an object or instruction of her choosing. DREM makes this impossible by executing two replicas of the program with different address space layouts. The address spaces are diversified by shifting the locations of stacks, heap, globals, text, etc. As the leaders and followers execute, each follower checks to ensure that the instruction addresses of the executed instructions maintain the fixed text offset which is specified at program initialization.

The address space diversification protects against memory corruption attacks because every corresponding instruction will be located at a different address in the replicas. Since the same malicious input is sent to both replicas, it is impossible for the attacker to change the pointer to point to the same corresponding instruction in both contexts– while the corresponding instructions will be located at different addresses, the pointer will be overwritten by the same value in both replicas. As a result, execution will be directed to different instructions in the replicas resulting in a divergence in instructions being executed.

An example of a buffer attack detection is show in Figure 3.2. In this scenario, two contexts are initialized with a fixed offset between the stacks of 0x100000. The addresses and values of the relevant stack regions are shown at the top of the figure. A malicious user injects malicious code into the stack region and overwrites the return address to point to the start of the exploit code in one of the layouts. When that function returns, it dereferences the return address and jumps to that location. Since in both contexts, the return address was overwritten by the same value, it breaks the fixed offset specified for the text regions. When the offset change is detected, the exploited application is shut down.

A weakness in this detection mechanism arises because the offset must maintain alignment restrictions set by the operating system and architecture of the machine. For example, an offset applied to a cache aligned address must result in another cache aligned address. These restrictions give the attacker room to change the lower order bits of an address not protected by the offset without detection. For example, if a code pointer in two contexts contains values 0xAAA000 and 0xBBB000 with an offset 0x111000, then any of the 12 low order bits are not protected by the offset. The attacker can overwrite these values in both contexts without detection. This is a known problem of redundant execution [11] and has not been addressed in this thesis.

Chapter 4

Architecture

The following section describes how DREM enables redundant execution with diversified address spaces on multicore processors with snoopy MESI coherence. First, we describe how a pair of cores work together to achieve redundant execution of a single application thread. Then, we demonstrate how these multiple core pairs integrate together to provide deterministic redundant execution of multiple threads on idealized hardware with infinite caches. Finally, we remove the infinite cache assumption and show how DREM behaves on realistic hardware. Table 4.1 summarizes the hardware that DREM adds to the processor and Table 6.1 gives the sizes of these hardware structures.

4.1 Single-threaded applications

To enable DREM, an operating system initializes two address space diversified application instances. The instances differ in the locations of shared libraries, stack, heap, global, and text segments. After initialization, the operating system simultaneously schedules the two instances on a pair of cores called the leader and the follower. The operating system supports redundant execution by buffering the results of system calls in the leader threads and feeds the identical results to the follower threads [11, 25]. This removes divergences that would normally result from the replicas executing system calls that return time-

Hardware Structures	**Section**
I-FIFO	4.1
IC Register	4.2
IC cache tags, M-FIFO, previously-modified cache bit	4.2.1
LIC, RIC	4.2.2
Read Bloom Filter, Write Bloom Filter	4.3
Rendezvous IC register	4.3.1

Table 4.1: Summary of hardware structures required to support deterministic redundant execution

dependent or random results such as `gettimeofday` or reading from `/dev/rand`.

To support redundant execution, the leader and follower cores are coupled together with an instruction FIFO (I-FIFO) as shown in Figure 4.1. The leader runs ahead of the follower and enqueues the address of every committed instruction onto the I-FIFO. The follower's text segment location differs from the leader's by a fixed offset, which is set and written into a special register by the OS prior to execution. To detect a divergence, the follower core dequeues an instruction address from the I-FIFO during commit, applies the offset to that address, and compares it against its own committed instruction address. If the addresses do not match, the instruction stream in the two replicas has diverged and an exception is raised.

Because of events such as cache misses and bus arbitration delays, corresponding threads in the leader and follower may execute at different rates. To tolerate these differences, DREM's I-FIFO has 512 entries allowing the follower to lag by up to 512 instructions.

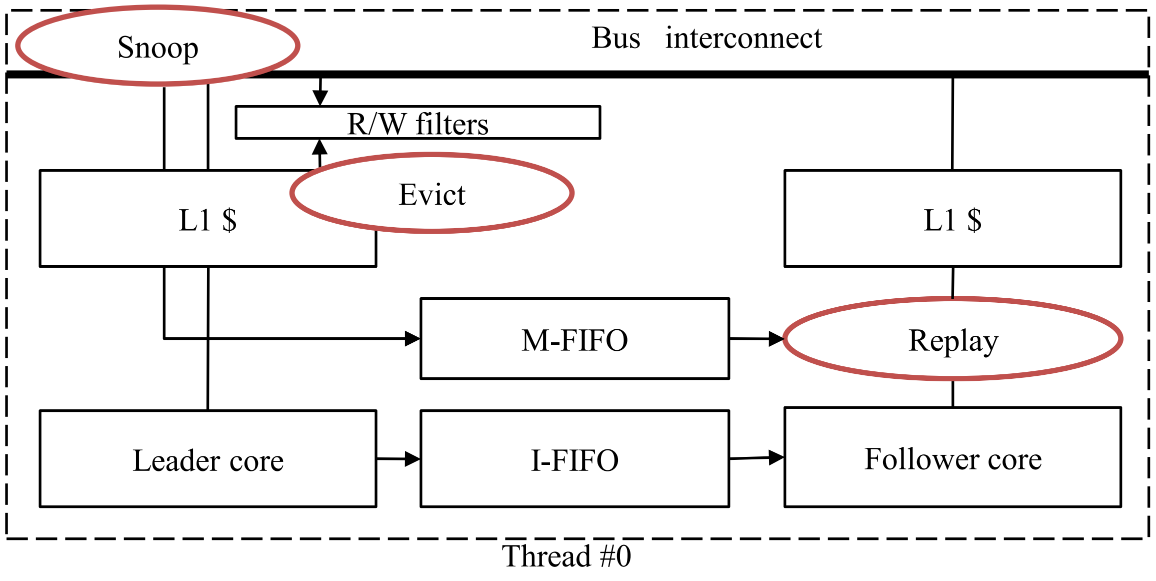


Figure 4.1: DREM hardware

4.2 Multithreaded applications with infinite caches

Multithreaded programs contain races, which if allowed to resolve differently on leader and follower, may lead to concurrency related divergences in the replicas. To support redundant execution, DREM records shared memory dependencies between the leader cores and replays them on the follower cores. We define a dependency as the closest pair of memory accesses executed on two different cores that access the same location in memory where at least one memory access is a write.

Our race recorder assumes a sequentially consistent memory model, so that all memory accesses performed by threads create a total order and each thread's memory accesses match its program order. Since each thread's accesses match its program order, we use an upward counting memory instruction counter (IC) in each core as a Lamport timestamp [19] to create a partial ordering between dependent memory accesses. The IC is implemented as a 32-bit register that is incremented every time a memory request is issued.

4.2.1 Recording dependencies

Dependencies are recorded as a pair of tuples, where each tuple contains a IC value and a core ID. Together, these allow a tuple to identify a particular dynamic memory instruction on a particular core. The pair of memory accesses that constitute a dependency are called the *source* and *destination*. The source precedes the destination in the total order.

To record memory dependencies, the MESI cache coherence hardware needs to be augmented with three new components. First, an IC tag is added to each cache line in the private caches of each core. The IC tag records the memory instruction count of the last read or write to the line. Under our infinite cache assumption, no cache lines are evicted due to conflict and capacity misses. This relieves us from having to worry about losing IC information about the last access to a cache line. Since no cache lines are evicted, all memory dependencies are revealed by cache coherency messages. When a core makes a memory request that depends on the value stored in another cache, a cache coherency exchange is made to synchronize the cache line between the pair of cores. The IC and core ID from the source are piggy-backed to the destination on this exchange. The core making the memory request (the destination) records the IC of its own memory request and associates it with the core ID and IC of the source, which it gets from the piggy-backed message.

When a core assembles an IC pair and a core ID of a dependency, it records the dependency on the second piece of hardware DREM adds – a FIFO buffer called the M-FIFO. The M-FIFO connects each leader-follower pair in the same fashion as the I-FIFO.

Since DREM requires at least one operation to be write, DREM only records read-after-write (RAW), write-after-write (WAW) and write-after-read (WAR) dependencies.

We illustrate an example how DREM records a RAW dependency in Figure 4.2. In this scenario, core #0 issues a write with IC 50 to line X denoted by W(X,IC=50). Later, core #2 issues a read R(X,IC=25). It places a read for line X onto the snooping bus

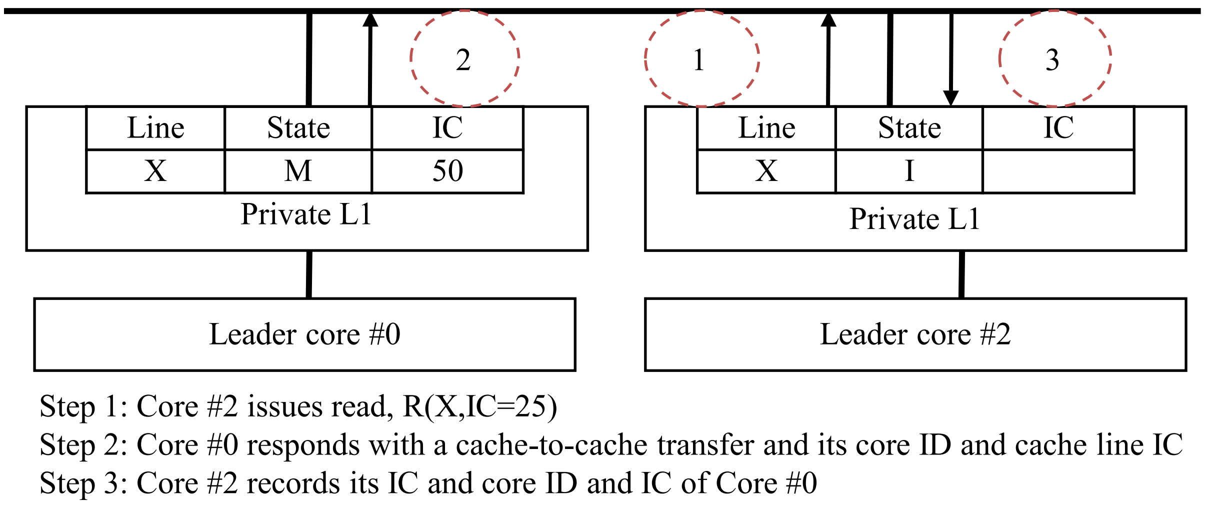

2
1
3
Line
State
IC
X
M
50
Private L1
Line
State
IC
X
I
Private L1
Leader core #0
Leader core #2
Step 1: Core #2 issues read, R(X,IC=25)
Step 2: Core #0 responds with a cache-to-cache transfer and its core ID and cache line IC
Step 3: Core #2 records its IC and core ID and IC of Core #0

Figure 4.2: RAW recording

and waits for an acknowledgment. Core #0 snoops in its cache, sees that it has the line and sends an acknowledgment to core #2 along with its IC(50) and core ID(0). Core #2 then records its own IC(25) and the response core ID(0) and IC(50).

The DREM requires a third piece of hardware to be added to the processor – a per-cache line "previously-modified" bit. To see why this bit is needed, consider what happens in the previous example when a third core #4 attempts to read line X. To correctly make a RAW dependency from core #0 to core #4, core #0 must "remember" that it has previously modified the line since the cache coherence state of its line is now shared. To record that core #0 had modified the line at one time, DREM sets the previously-modified bit whenever a line transitions from M to S state, and clears it when the line is invalidated. When a read request on the snooping bus matches a line in a core's private cache, the core will form a dependency with the reader if the previously-modified bit is set, even if the current state of the line is shared.

When the first core in a WAW dependency performs its store, it sets the cache line state to modified (M). In an infinite cache, there can be no evictions due to conflict and capacity misses, the cache line maintains its M state until another core writes to an address backed by this line. When that occurs, the cache with the modified line will perform a cache-to-cache transfer to the writer and piggy-back the line's IC and its core ID.

The starting scenario for a WAR dependency is either one core with a cache line in exclusive (E) or modified (M) state or multiple cores with the line in shared (S) state. When the write occurs, it has to record dependencies to all the cores that cache the line. All the cores that observe the write signal initiated by the writer respond with their core IDs and respective cache line ICs.

We note that our recording algorithm can lose precision but still record a sufficient set of dependencies. For example, as shown in Figure 4.3, suppose a core performs a write and updates the cache line IC with the value 5. If it later reads from that line, it will update the IC of the cache line with the value of the read and leave it in M state. When another core makes a RAW dependency, the source IC of the dependency will be equal to the IC of the read and not the write. This type of dependency, which we call a *loose dependency*, still guarantees deterministic replay because R(X,IC=10) will always occur after W(X,IC=5) due to the sequential consistency property.

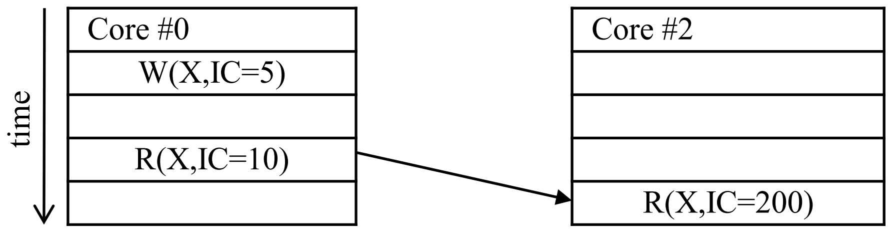


Figure 4.3: Loose dependency

4.2.2 Replaying dependencies

Each follower core also maintains an IC that starts at the same initial value as the leader's counter. Before issuing a memory request, each follower core checks the top of the M-FIFO to see if the upcoming memory access has any dependencies. If the follower request IC matches an IC in the M-FIFO then a dependency exists and the follower must check whether the dependency has been met. If it has not, the core must stall until the dependency has been met (i.e. the source core has passed the IC in the dependency). Note that if a memory operation has multiple dependencies, the follower has to ensure that all the dependencies have been satisfied.

To enable this, two additional hardware structures are added to each follower core to replay dependencies in DREM, a vector of last observed ICs (LIC) and a vector of response ICs (RIC). Each vector contains an entry for each follower core. All the cores snoop on the bus for IC updates and update their respective LIC. This happens whenever a core places its IC on a bus, for example when piggy backing ICs on coherence messages, or when broadcasting IC requests and callbacks, which we discuss below. Because the entries in the LIC are only updated opportunistically, it is possible for them to be inconsistent with the IC value of the actual core whose LIC they are caching.

Suppose a follower core K dequeues a dependency from its M-FIFO that indicates that the current instruction has a dependency with a follower core P at IC_P. In the simple case, if $LIC[P] \geq IC_P$, then the dependency has already been satisfied and the follower can proceed.

If $LIC[P] < IC_P$, then K does not know if P has passed IC_P or not. K stalls and sends a request to P for a callback when $IC[P] \geq IC_P$. When P receives such a request, it checks to see if $IC[P] \geq IC_P$. That condition could be true because K has potentially outdated information stored in its LIC. If $IC[P] \geq IC_P$, P responds immediately and wakes up K. We call this type of stall a *soft-stall* because the dependency was satisfied but the core had to momentarily wait because it had outdated IC information. If $IC[P] <$

IC_P, then P has not passed the source of the dependency yet. P records IC_P in a response IC vector (RIC), $RIC[K]$. As P continues to execute, it scans the RIC to see whether it can issue the callback. When P eventually executes IC_P (i.e. $RIC[K] < IC$), it resets $RIC[K]$ and broadcast its IC on the shared bus. K receives this broadcast and continues execution. This type of stall is called a *hard-stall* because the dependency was not satisfied initially and K had to wait until P caught up and executed IC_P.

Follower cores broadcast their updated IC whenever they request a callback and when they issue callbacks. Cores broadcast their ICs for callback requests to eliminate deadlocks that can occur when two cores have mutual dependencies and both stall waiting for each other because they have outdated LIC information. Sending an IC on callbacks notifies the stalled core that its dependency has been fulfilled.

4.3 Supporting finite caches

In this section, we extend DREM to support finite caches. Finite caches pose a difficulty to the previously described recording algorithm because the caches lose the last access information when cache blocks are evicted. This can cause the recorder to miss some dependencies. For example, suppose core #0 writes to a cache line, which is then evicted due to a capacity miss. If core #2 later reads from the same address, core #0 will not respond with the IC of its write since the IC has been evicted with the cache line. As a result, no dependency between core #0 and core #2 will be recorded.

4.3.1 Recording dependencies with evictions

To capture the dependencies that are lost due to evictions, we add a read and write bloom filter for each core. The filters are indexed by the address of the cache line. If a cache line is evicted in M state or in previously-modified state, we add it to the write bloom filter. All other evictions are added to the read bloom filter. We use two separate bloom filters to ignore read-after-read (RAR) dependencies.

In addition to snooping in its L1 for cache coherence, each core checks its read and write bloom filter for dependencies. If a core observes a read, it only searches its write bloom filter. For writes, the cores search in both read and write bloom filters. If there is a hit, then a potential dependency exists between the core performing the request and the core with the bloom filter hit. Since bloom filters can have false positives the dependency created could be unnecessary. Such dependencies impose extra constraints during replay, reducing the amount of concurrency. However, we note that this only hurts performance, and does not affect correctness.

Rather than record dependencies on evicted cache lines pairwise as we did in the previous section, dependencies with evicted lines are recorded using a *rendezvous point* between all cores. A rendezvous point is recorded by the leader cores by noting their current position in the instruction stream and replayed like a barrier on the follower cores thus ensuring that the evicted memory access happens before the rendezvous and the new request after the rendezvous. The rendezvous also transitively reduces all dependencies that start prior to the rendezvous and end after it. The transitive reduction allows us to flush the bloom filters of all cores and ignore all future dependencies that originate prior to the flush.

To record a rendezvous, every core records their current last committed IC in the M-FIFO. This IC will be used to re-synchronize all the cores during replay. In addition, each core will store the last committed IC in a special rendezvous IC register. The saved IC is used to transitively reduce dependencies that cross the rendezvous. If a core receives a request for a cache line with an IC less than the IC of the last rendezvous, no dependency will be recorded.

We demonstrate how a rendezvous allows us to record dependencies between evicted cache blocks and how it transitively reduces dependencies by an example shown in Figure 4.4. Suppose core #0 performs a write to addresses X and Y. Eventually, the cache

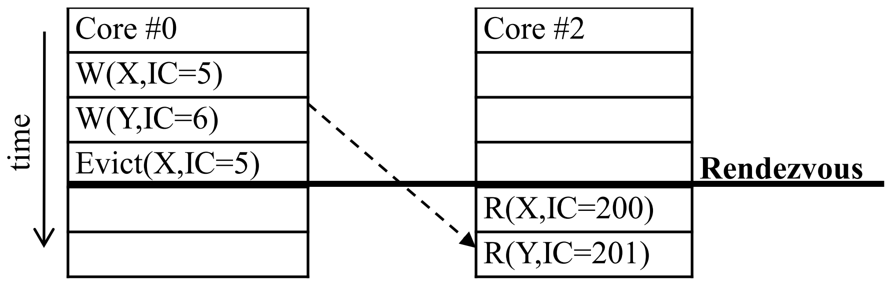


Figure 4.4: Rendezvous example

line backing X is evicted. Later, core #2 issues a read on address X and places the request on the bus. Core #0 observes the read and searches its write bloom filter. It has a hit because it previously evicted X and recorded this eviction. Core #0 notifies #2 of the hit and both cores create a rendezvous by recording their last committed IC. Then, both cores flush the filters and record the IC of the rendezvous. When core #2 issues a read for Y, even though there is a cache coherency exchange between the two cores, we do not record this dependency because it is transitively reduced by the rendezvous.

Note that applications typically make dependencies between data that is stored in the private caches. Data that has been evicted rarely causes dependencies to be created. As a result, most bloom filter hits are caused by false positives and not actual dependencies. This observation motivates our bloom filter clear policy. Instead of tolerating multiple bloom filter hits, we choose to create a rendezvous point on the first hit and clear the filters. An alternate strategy is to tolerate several bloom filter hits and then create the rendezvous point. We studied this policy and noted that tolerating more hits in the bloom filters creates excessive false positives which removes any advantage in reducing the number rendezvous points. The false positives arise since tolerating more hits increases the occupancy of each bloom filter thereby increasing each filter's false positve rate. The extra dependencies also arise due to WAW and RAW dependencies becoming ambiguous. Suppose that that a core performs a write and hits in the write bloom filters of two cores. There is no information about the ordering of the two evicted writes - it is ambiguous which write was performed first. To be conservative the new write has to create dependencies with both cores.

4.3.2 Replaying dependencies

Follower cores look at the head of the M-FIFO to see if their current IC matches the IC of a rendezvous. If it does, the follower core does not issue the current memory request. Instead, it broadcasts that it reached a rendezvous point to all the other cores. All cores keep track of the number of such broadcasts and wait until the number of such broadcasts equals the number of follower cores. When the last follower core arrives at the rendezvous, it broadcasts its arrival, and all the cores simultaneously wake up and resume execution.

Chapter 5

Implementation

The following chapter discusses how DREM was simulated using an academic cycle-accurate multicore simulator SESC [27]. First, we discuss how SESC was extended to support simulation of two identical processes. Second, we outline the I-FIFO implementation and redundant execution of single-threaded applications. Third, we describe how SESC was modified to support simulation of pairs of redundant execution threads. Finally, we describe the memory subsystem changes required for the race-recorder and replayer.

5.1 Redundant processes

SESC is designed to simulate one application instance at a time. Since DREM requires two diversified instances of an application running in tandem, SESC has to be extended to provide this support. These modifications are akin to the changes that need to be made to an operating system for redundant execution. First, SESC has to correctly initialize redundant contexts. Second, all the process and thread management system calls that trap into SESC have to correctly handle redundant contexts.

5.1.1 Address space initialization

SESC's initialization routine creates one virtual and one physical address space for the main program context. To support redundant execution, SESC needs to create another set of virtual and physical address spaces.

All the regions of the virtual address space(heap, stack, text, etc) for the follower context are initialized with the same values as the leader context. Note that the virtual address space of the redundant context has to be diversified for security. Since the goal of this thesis is to study the performance implications of redundant execution, the virtual address space of the redundant context was not diversified. The performance results of redundant execution are not affected by whether the contexts are diversified or not. However, it does simplify the implementation since it allows us to use load value queues to aid in simulation. The details of these queues are described later in this chapter. Once the virtual address space for the follower context is created, it is initialized to point to the redundant context's physical memory.

The physical address space is allocated inside SESC. After the main application context is initialized, the follower context is allocated with identical contents. For the purposes of performance simulation, the virtual address spaces were not shifted, thus allowing us to copy the physical contents of the initial context verbatim. If the text segments were shifted, the absolute references in the executables have to be modified respectively.

5.1.2 Process and thread management

All the SESC system calls that are exposed to applications that deal with thread management have to be aware of redundant contexts. The systems calls include operations such as process and thread creation and synchronization via *wait*.

Process creation has to take care of setting up identical initial state for both program contexts. When the redundant process is created, environment variables, program parameters, and registers are initialized identically across the two contexts.

The thread creation system calls are modified to create follower threads when a leader thread is created. Once the two threads are created, they are linked by reference and pinned on adjacent leader and follower cores.

Finally, the semantics of *wait* in SESC have to be changed for redundant execution. The original semantics of wait in SESC specify that the wait system calls suspends the waiting thread if any of the child threads have not exited. If all threads have exited before the wait system call is invoked, the thread is not suspended. A problem arises if the leader and follower contexts behave differently at the wait system call. If only one of the contexts is suspended, a divergence in the instruction stream arises because the sleep call executes extra instructions. To eliminate this spurious divergence, SESC was modified to force both leader and follower contexts to suspend if any of the leader and follower child threads have not exited when the leader wait is called. If all threads have exited when the leader calls wait, then the leader and follower parent threads are guaranteed to not wait. Note that this change only pertains to how SESC accounts for executed instructions and it would not be necessary when making OS related changes to support DREM.

5.2 Single threaded redundant execution

To support redundant execution of single-threaded applications, only an I-FIFO is required. The I-FIFO structure is implemented using a simple STL queue. It connects the frontend and backend of the leader core and to the frontend and backend of the follower core.

```
if (isLeader()) {
        instFIFO->enqueue(retireVAddr);
}
if (isFollower()) {
        assert (instFIFO->dequeue() == retireVAddr);
}
```

Figure 5.1: Pseudo-code of the retirement checks made in the ROB

5.2.1 Backend

At the retirement stage of the leader core, all the non-speculative instructions are committed in the re-order buffer(ROB) and enqueued onto the pipelined I-FIFO. Since the leader core executes several hundred instructions ahead of the follower core, we assume that the latency of enqueuing and dequeueing the same element onto the I-FIFO will be overlapped by the time it takes the follower core to catch up. As the follower core commits its instructions, it dequeues an address from the I-FIFO and compares it against its own committed address. The cost of dequeuing was not accounted in the simulations because the I-FIFO is only empty upon initialization. The initialization period of the I-FIFO is short with respect to the entire simulation run. When the I-FIFO is non-empty, we assume that there is enough time to propagate the instruction address. In case of a divergence, the simulation is halted. The pseudo-code of the checks in the ROB are shown in Figure 5.1.

5.2.2 Frontend

The frontend of the leader core has to be modified to peek into the I-FIFO to check to ensure that it has enough room to enqueue additional instructions. If the I-FIFO is full, the front-end of the leader core stalls and does not issue any additional instructions.

Similarly, the front-end of the follower core performs a check to ensure that the current I-FIFO size is greater than the number of instructions in the pipeline. If that is the case, the instruction is issued. Otherwise, the follower core stalls. These changes were added to *Processor::advanceClock* which is responsible for driving each stage of the SESC pipeline.

5.3 Simulating multiple redundant threads

Before implementing the race recorder in SESC, the simulator itself needs to be changed to support redundant execution of multithreaded applications. When running a SESC simulation, the simulator functionally executes instructions as it fetches them. Functional execution during fetch allows SESC to know apriori whether the branch predictor is correct, what address the instruction will potentially access, etc. After functionally executing an instruction, SESC creates a dynamic instruction object and specifies all the parameters in the object that it needs for simulation. An example of that is whether the dynamic instruction is speculative or not. The dynamic instruction is used for the timing simulation and hence it is passed through the simulator pipeline.

The race-recorder described in the next section modifies the timing simulation of SESC and leaves the functional simulation intact. As a result, the functional simulation itself can create spurious divergences. If the race-recorder enforces a certain order of memory accesses in the follower cores, it does not preclude the functional simulator of SESC to execute instructions in a different order when creating the dynamic instruction objects.

To accurately model the timing aspect of the race-recorder and eliminate the divergences resulting in the functional simulation, we used load value queues(LVQ) inside SESC to pass load values from leader threads to the follower threads and suppress all stores in the followers during functional simulation. The functional simulation of stores did not change the process memory in SESC but were still modeled accurately in the timing phase of the simulation. The LVQs eliminate all divergences in the follower threads since they guarantee that the view of memory is identical for the functional simulation while still allowing arbitrary re-ordering during the timing simulation.

Divergences can also occur due to system calls returning different results to a leader and a follower thread. We used similar load value queues to save the results of system calls that are written to registers from the leader threads to pass to the follower threads. The results that are written to memory are automatically forwarded using the LVQs for load instructions. Similarly, all the system calls that change system state do not perform any action for the follower threads. An example of modification to the *read* system call is shown in Figure 5.2.

5.4 Race-recording and replay

The DREM race-recorder requires addition of instruction counters, bloom filters, and changes to both the processor and the bus side interface of the caches, and cache tags. The following subsections detail the implementation of DREM in SESC.

5.4.1 Instruction counters

Each processor maintains a monotonically increasing memory IC. The counter was added to the *Processor* class. Whenever an instruction is decoded and verified to be a memory instruction, the IC is incremented. The processor maintains a committed memory instruction counter that is incremented whenever a memory instruction is committed.

In addition to the two counters, each dynamic instance of a memory instruction has the current value of the processor IC counter assigned to it. This dynamic IC value is propagated with the instruction until the memory operation is issued. Prior to scheduling the memory access, the dynamic IC is checked against the corresponding entry in the LIC vector.

```
#define LVQ(VAR,EXPR) \
 if (pthread->leader) { \
   VAR = EXPR; \
   lvq[thePid].push(VAR);\
   lvq[thePid].push(errno); \
 } \
 else { \
   VAR = lvq[thePid-1].front(); \
   lvq[thePid-1].pop(); \
   errno = lvq[thePid-1].front(); \
   lvq[thePid-1].pop(); \
 }

OP(mint_read)
{
     Pid_t thePid=pthread->getPid();
     int32_t r4, r5, r6;
     int32_t err;

     r4 = pthread->getIntArg1();
     VAddr buf = pthread->getIntArg2();
     r6 = pthread->getIntArg3();

     LVQ(err, read(r4, (void *)(pthread->virt2real(buf)), r6));
     pthread->setRetVal(err);
     if (err == -1)
         pthread->setperrno(errno);

   // Return from the call (and check for context switch)
   I(pthread->getPid()==thePid);
   return pthread->getRetIcode();
}
```

Figure 5.2: Example of LVQ usage in the read system call

```
Line *l = cache->writeLine(addr);

if (l && l->canBeWritten()) {
  l->instrCount = mreq->getDInst()->instrCount;
  l->modified = true;

  writeHit.inc();
  ...
  return;
}
...
```

Figure 5.3: Example of updating the IC and the previously-modified bit

5.4.2 Cache tags

Caches require two additional tags. One tag is necessary to store the IC value of the last memory operation to access the cache line and another tag to store the previously-modified bit. The updates to the cache tag IC are done in *SMPCache::doWrite* and *SMPCache::doRead*. An example of the update during writes is shown in Figure 5.3. We assume that the updates to the tags can be overlapped with the actual write operation and do not increase the critical path of the cache access.

```
PAddr rpl_tag = calcTag(rpl_addr);
// only fill up the bloom filters for the leaders
if (IS_LEADER(l->cpuId)) {
  if(l->isDirty() || l->modified) {
    evict_dirty.insert(rpl_tag);
  } else {
    evict_clean.insert(rpl_tag);
  }
}
```

Figure 5.4: Example of updating the bloom filters

5.4.3 Cache evictions

Whenever a cache line is evicted from the private L1 caches, the cache address needs to be inserted into a bloom filter to ensure that no dependencies escape the race-recorder. When a new cache line is allocated in *SMPCache::allocateLine* an older cache line gets evicted. We insert the addresses of the evicted dirty and modified lines into the write bloom filter. All other evictions are inserted into the read bloom filter. The addition shown in Figure 5.4 was added to the cache line allocation procedure in SESC.

5.4.4 Processor-cache interface

The processor-cache interface on the follower cores has to perform two checks. First, the core must check whether the upcoming memory operation has satisfied all the necessary memory dependencies. Second, the core must check whether it has a pending acknowledgment that it must service.

Checking dependencies requires a lookup into the M-FIFO to check whether the IC of the upcoming memory operation matches the IC of a dependency in the FIFO. If the source of the dependency is less than the $LIC[P]$, the core sends a broadcast and is put to sleep. A dependency on the M-FIFO could also signify rendezvous point. In that case, the core allows the memory operation to complete that marks the rendezvous point and stalls on the next memory operation.

In our implementation, we assume the M-FIFO has a similar structure to the I-FIFO - it's pipelined and the head of the FIFO is ready to be read when the memory operation is about to be scheduled. However, we do model the latency of bus accesses and bus contention when sending these broadcasts. Bus accesses in SESC are modeled with slots. If the upcoming bus slot has been filled by another request, a newer bus access will have to wait for the next available slot.

Checking for pending acknowledgments requires a parallel lookup in the RIC instruction vector and a comparison to an IC. The RIC has an entry for all the follower cores. Hence, for a 32 core processor, each follower core will have 15 RIC entries. We assume that the lookup in this structure can be overlapped with the lookup into the M-FIFO. If an entry in the RIC is less than the currently scheduled memory operation IC, the core sends a broadcast to wake up any sleeping cores and resets the RIC vector.

5.4.5 Snooping-cache interface

The snooping interface on the leader cores is responsible for replying to MESI broadcast messages. In addition to that, DREM extends the logic at that interface to perform two additional tasks: replying with IC and core ID to incoming coherence requests and lookups in bloom filters to find dependencies with evicted cache blocks.

When an incoming coherence request arrives at the snooping interface of a leader cache, the cache performs a tag-lookup to find whether a matching address is loaded in the private cache. If there is a hit and a reply is due, the IC that is stored in the cache tag and the core ID are piggybacked in the coherence reply. Sending additional information in the reply is modeled by utilizing an additional bus slot. Note that not all incoming coherence messages generate replies with piggy-backed IC information. For example, if the tag IC is less than the IC of the last bloom filter clear, then the dependency has been transitively reduced and no reply is necessary.

In addition to the tag lookup, DREM requires a parallel lookup in a 4096-bit bloom filter. We assumed that the bloom filter lookup can be overlapped with the tag lookup(which is 8-way associative). If there is a hit, the bloom filters are set to be cleared and a broadcast is made to all cores to record the synchronization point.

A simplified version of what each core does on a read miss is shown in Figure 5.5. Similar actions are performed on invalidate and write misses.

```
void MESIProtocol::readMissHandler(SMPMemRequest *sreq)
{
  PAddr addr = sreq->getPAddr();
  Line *l = pCache->getLine(addr);
  PAddr tag = pCache->calcTag(addr);
  int readFilterHit = false;
  int writeFilterHit = false;
  GProcessor* attachedCPU =
    osSim->id2GProcessor(pCache->getCpuId());
  const MemRequest* depRequest = sreq->getOriginalRequest();

  SMPCache* receiver = pCache;
  SMPCache* sender = (SMPCache*) sreq->getRequestor();

  if (l && !l->isLocked())  {
    if (l->getState() == MESI_MODIFIED || l->modified) {
      if (attachedCPU->isLeader() &&
            l->instrCount > pCache->getLastClear()) {
        sreq->addSourceDep(l->cpuId, l->instrCount,
        l->memop == MemRead);
      }
    }
    combineResponses(sreq, (MESIState_t) l->getState());
    changeState(l, MESI_SHARED);
    sendReadMissAck(sreq);
    return;
  }

  if (IS_LEADER(sender->getCpuId())
         && IS_LEADER(receiver->getCpuId())) {
    writeFilterHit = pCache->evict_dirty.search(tag);
    readFilterHit = pCache->evict_clean.passiveSearch(tag);
  }

  if (pCache->evict_dirty.atCapacity()) {
    sreq->setFilterFull();
    pCache->evict_clean.clear();
    pCache->evict_dirty.clear();
  } else if (readFilterHit) {
    sreq->setState(MESI_SHARED);
  }

  sendReadMissAck(sreq);
}
```

Figure 5.5: Example of updating the bloom filters

Chapter 6

Evaluation

6.1 Experimental setup

To evaluate DREM we used SESC, a cycle-accurate multicore simulator for the MIPS architecture. We modified SESC to model an in-order execution DREM multicore processor with up to 32 cores with 16 KB private L1 caches and a 512 KB shared L2 cache. MESI snooping coherence is used on a 50 GB/s inter-processor bus that connects the L1 caches and the shared L2. The second L2 port is connected to a 10 GB/s memory bus. All the relevant architectural parameters are listed in Table 6.1.

The L1 cache is sized to accommodate the most important working sets in almost all SPLASH-2 applications. Selecting a larger L1 cache size would absorb a larger portion of the working set and keep the miss rates low. Since the rendezvous point frequency is sensitive to the miss rate, the chosen L1 size allows us to study some applications that exhibit higher miss rates. Similarly, the shared L2 cache is sized to be smaller than the total working set size of the SPLASH-2 applications. A larger L2 cache size would encapsulate most of the working set and not stress the memory bandwidth sufficiently.

The I-FIFO and M-FIFO sizes were sized conservatively. We varied the size of both structure to determine at which point increasing the size of the structure had no significant overhead on redundant execution. Smaller I-FIFO and M-FIFO sizes resulted in more frequent stalls when the FIFOs became full.

The bandwidth of the interprocessor-bus and the memory bus is chosen to match the bandwidth provided by today's multicores.

To support the race recording algorithm, we need support for the sequentially consistent memory model. SESC is modified to support sequential consistency by enabling in-order execution, disabling load-forwarding, and disabling load-bypassing. To evaluate the performance impact of DREM, we ran benchmarks from the SPLASH-2 suite. These applications were chosen since they produce many shared memory accesses thus allowing us to stress test the recording and replay of DREM. The parameters used for the benchmarks are listed in Table 6.2.

DREM is able to detect and prevent memory corruption attacks through redundant execution of replicas with address diversity. DREM provides the same level of security as other redundant execution systems, such as N-Variant, whose ability to detect attacks has already been evaluated in the literature [11]. As a result, we focus on evaluating the performance penalty DREM imposes. DREM's performance overhead has two components: overhead due to redundant execution and overhead from race recording and replay. Redundant execution overhead is caused by additional resource requirements due to running a redundant replica, which increases pressure on the processor interconnect, shared cache, and memory bandwidth. In addition, the I-FIFO and M-FIFO can stall leader and followers when they are empty or full, respectively. Race recording and replay overhead is a result of stalls and additional bus messages required to record memory dependencies, and stalls imposed during replay to satisfy those dependencies.

By evaluating the cost of redundant execution independently from the cost of record and replay we are able to separately measure the cost that any redundant execution such as N-Variant [11] or Replicant [25] system must pay and the additional overhead DREM imposes to support redundant execution of multithreaded workloads.

Frequency	5 GHz
Issue	4-way
DL1	16 kb/4-way/32b block
Hit/miss latency	2/2 cycles
Read bloom filter size	4096 bits
Write bloom filter size	4096 bits
I-FIFO size	512 (2 kb)
M-FIFO size	128 (768 bytes)
LIC/RIC size	16 entries (64 bytes)
IL1	32 kb/2-way/32b block
Hit/miss latency	1/1 cycles
Shared L2	512 kb/8-way/32b block
Hit/miss latency	10/14 cycles
Coherence	MESI
Shared bus latency	10 cycles
Shared bus bandwidth	50 GB/s
Memory bus latency	15 cycles
Memory bus bandwidth	10 GB/s
Memory latency	470 cycles

Table 6.1: Simulation parameters

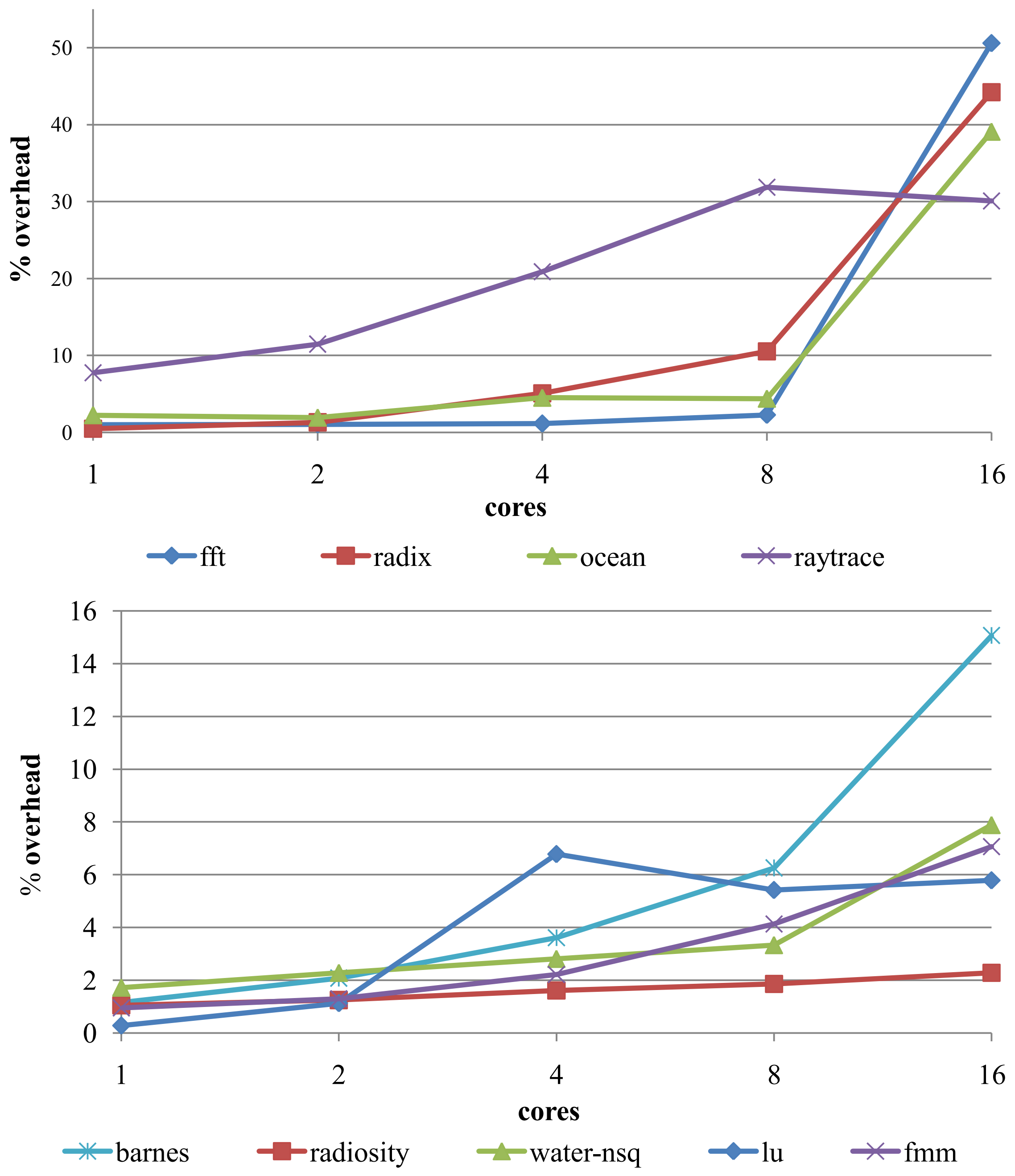


Figure 6.1: Redundant execution overhead

Barnes	16K particles
FFT	64K points
FMM	16K particles
LU	512x512 matrix, 16x16 blocks
Ocean	258x258 ocean
Radiosity	Room, -ae 5000.0 -en 0.050 -bf -.10
Radix	1M integers, radix 1024
Raytrace	Car
Water-nsq	512 molecules

Table 6.2: Benchmark inputs

6.2 Redundant execution overhead

To study the overheads for the SPLASH-2 applications without a race recorder, we modified the functional simulation of SESC to forward load values from the leader to the follower threads. This effectively removes all races without any overhead while accurately simulating the timing of the follower cores. We then compare the runtime of the SPLASH-2 benchmarks run redundantly on this modified processor against a single instance running on a vanilla processor. Figure 6.1 shows the overhead as we scale the number of cores. The top graph shows the overhead for benchmarks with higher memory bandwidth requirements and benchmarks with lower memory requirements are shown in the bottom graph. As we can see, memory bandwidth is the limiting resource for redundant execution as benchmarks with higher bandwidth demands also suffer higher performance overhead. Across these applications the average observed overhead is 22%.

To understand the effect of memory bandwidth pressure on the performance, refer to Table 6.3, which gives the miss rates for L1 and L2 under vanilla and redundant execution and the measured memory bandwidth utilization of vanilla. The memory increment

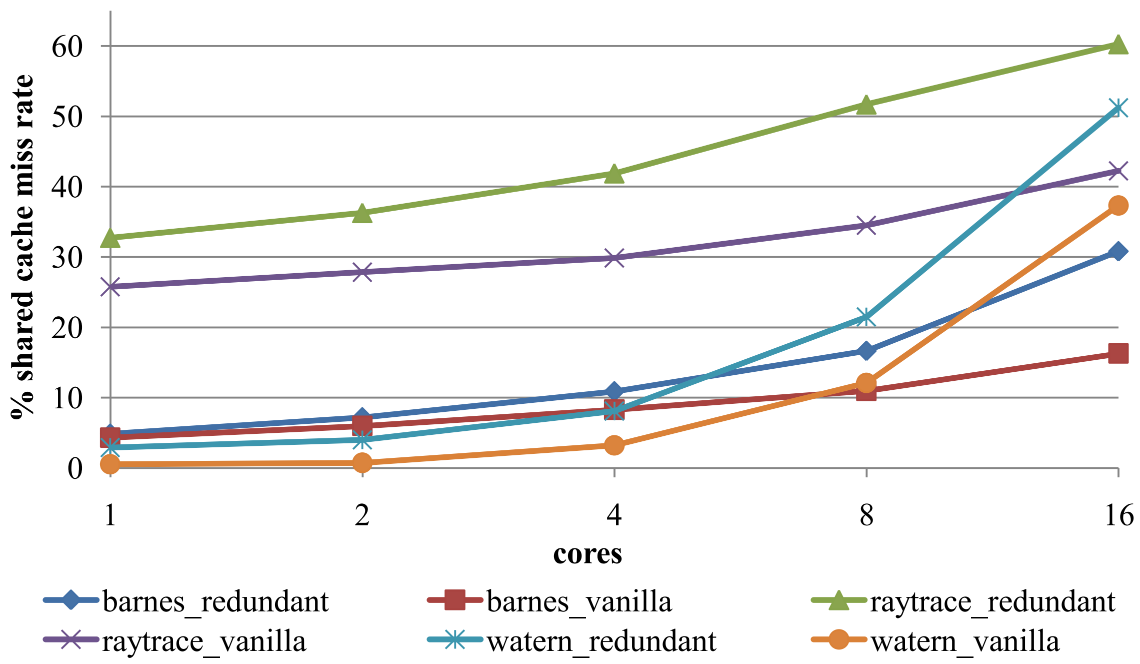


Figure 6.2: Shared cache miss rates

App	Both	Vanilla		Redundant		Mem
	L1	L2	Mem	L2	Mem_P	inc.
Fft	4.2	98	6397	99	13434	2.1
Ocean	10.6	93	6092	96	13402	2.2
Radix	7.2	64	4078	78	11418	2.8
Raytrace	3.9	40	2392	68	8133	3.4
Barnes	2.2	16	786	30	3065	3.9
Fmm	0.4	57	834	78	2252	2.7
Lu	0.3	78	808	93	1858	2.3
Water-nsq	1.5	38	578	52	1792	3.1
Radiosity	0.6	48	456	55	1048	2.3

Table 6.3: Miss rates and memory bandwidth for 16 leader cores

(Mem inc.) column gives the predicted increase in memory bandwidth requirement which is given by $2 \times L2MissRate_{redundant}/L2MissRate_{vanilla}$. This ratio accounts for two effects of the doubled number of cores that redundant execution uses on memory bandwidth consumption: more capacity misses due to higher cache pressure and doubled number of requests going to the shared cache. Note that L1 miss rates stay the same because each added follower core in redundant execution also has its own private L1 cache. The Mem$_P$ column gives the *predicted* memory increase given by multiplying the vanilla memory bandwidth with $Meminc$ (hence some of the columns are larger than 10 GB/s). As we can see, the predicted memory bandwidth gives a fairly good indication of what the performance impact will be – for example it predicts the larger performance overhead of Barnes despite having a modest memory bandwidth requirement under vanilla.

Both scaling the number of threads in the application and using redundant execution utilizes more cores, thus increasing the shared L2 miss rate. Increasing the number of cores used by the application causes a larger portion of the working set to be loaded into the L2 cache. Similarly, with redundant execution, the working set size is doubled because a copy is made for the leaders and another for the followers. This increases the competition for the shared cache and increases the miss rates. Figure 6.2 shows that the difference in miss rates between vanilla and redundant execution increases as more cores are added.

The effects of increased shared bus pressure and I-FIFO stalls are minor. We measured an overall 31% increase in bus arbitration time for the SPLASH-2 suite and the overhead of the I-FIFO was less than 2%. This is consistent with the overhead observed by other systems, such as SRT [26], that couple leader and follower cores with similar FIFOs. Both the I-FIFO and the M-FIFO were sized to provide the best trade off between performance and area.

6.3 Race recording and replay overhead

DREM introduces a very modest overhead to record and replay multithreaded races. The cost of recording dependencies are extra processor interconnect messages and the cost of replay comes from rendezvous points and stalls to satisfy un-met dependencies.

The number of recorded dependencies as a fraction of all shared memory accesses is very small. In fact, most dependencies are not recorded because they are transitively reduced by periodically occurring rendezvous points. Since dependencies cannot cross rendezvous, the recorded dependencies only arise between pairs of memory accesses that occur close to each other in time.

Rendezvous points introduce overhead during recording because the leader cores must perform a barrier-like synchronization. Rendezvous increase in frequency with higher L1 miss rates since this usually is also accompanied by an increased number of evictions into the bloom filters. In the workloads we tested, important working sets fit in the L1 cache, which keeps the number of rendezvous points low.

During replay, dependent pairs of memory accesses can randomly occur in the opposite order than was recorded in the leaders. Since this introduces a race, the destination of a dependency hard-stalls until the dependent memory accesses completes. Hard-stalls do not occur for all dependencies, only those that can be re-ordered during replay.

We note that two types of memory accesses generally result in dependency related stalls – those used to implement synchronization primitives such as locks and barriers, and those used to access shared data. Synchronization dependencies naturally occur in sections of code that are likely to be executed concurrently. The order in which cores acquire locks or pass through barriers is highly non-deterministic. As a result, as shown in Table 6.4, they usually cause a larger percentage of stalls over dependencies recorded when compared to the stalls due to shared data. In fact, over all SPLASH-2 applications, 66% percent of lock and 65% of barrier dependencies require stalls during replay and only 19% of data dependencies require stalls.

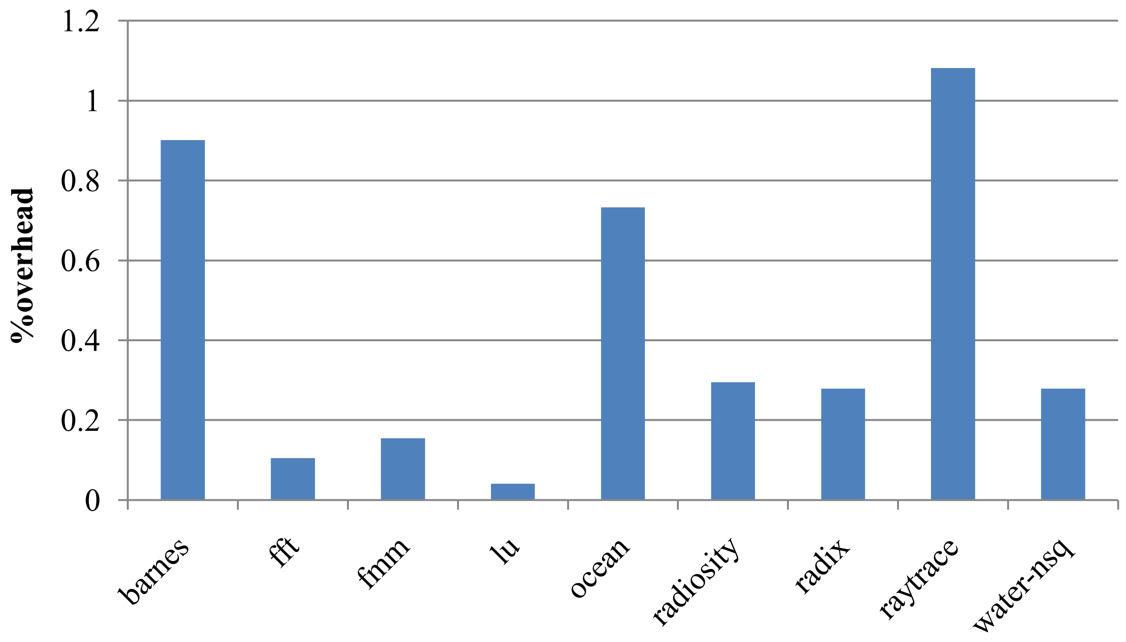


Figure 6.3: Relative overhead

In terms of average cost per stall, barriers have a higher cost associated than locks. In SPLASH-2, barriers make dependencies with all the cores and individual lock are typically shared by a subset of the cores. During barrier replay, a single core lagging behind causes all the remaining cores to wait and pay the stall penalty. Replay of locks that are shared by a subset of the cores causes fewer cores to be linked by dependencies and results in a lower probability of a core being 'late'.

The rest of the shared data communication has stalls during replay because of soft-stalls, loosely made dependencies, false sharing. In this suite, 53% of stalls are due to soft stalls, 26% are caused by loose dependencies and the rest are caused by false shared dependencies.

Rendezvous points are the most expensive to replay because they are dependencies between all cores unlike the previous types of dependencies which are made between pairs of cores.

The overall cost of race record-replay compared to redundant execution is negligible as shown in Figure 6.3. Raytrace exhibits the highest overhead due to its frequent rendezvous points and a high number of locks. Radix has the most frequent and costly rendezvous points. The average overhead over the tested SPLASH-2 applications is 0.43%.

App	Locks				Barriers				Shared data			Rendezvous	
	#	Deps /1M	Stalls /1M	Delay	#	Deps /1M	Stalls /1M	Delay	Deps /1M	Stalls /1M	Delay	# /1M	Delay
Barnes	69095	1357	933	301	17	2	1	1893	88	16	641	33	667
Fft	16	191	137	662	7	115	68	649	168	72	302	34	860
Fmm	46010	47	37	82	34	4	2	1770	66	24	134	6	910
Lu	16	2	1	595	67	10	6	205	259	40	64	1	832
Ocean	3328	435	310	370	900	1088	710	1390	8159	1216	237	117	1024
Radiosity	230052	1968	952	175	10	2	1	306	5164	1414	628	6	467
Radix	198	37	25	518	11	25	18	858	280	24	319	120	2303
Raytrace	366394	3243	2416	266	1	0	0	30	3656	630	128	78	1272
Watern	18704	288	184	571	20	14	9	1170	1673	342	142	2	1850

Table 6.4: Dependency Replay Costs for 16 Leader Cores

Chapter 7

Related work

DREM builds on a large body of work from the areas of fault tolerance, race recorders for multithreaded debugging, and software redundant execution. In this chapter, we outline which aspects of this related work influenced the design and implementation of DREM. In addition, we cover another popular hardware technique to protect against memory corruption vulnerabilities.

7.1 Transient fault detection

Fault tolerance work provides protection against soft errors by running two identical copies of a program with replicated inputs on pairs of cores [3, 30] or threads on SMT architectures [26]. Faults are detected when the two programs diverge.

SRT [26] proposes using simultaneously and redundantly threaded processors to detect soft errors. In SRT architecture, one hardware thread runs ahead and saves the results of all load operations in a load value queue(LVQ). The trailing thread does not perform any memory accesses itself since it reads values from the LVQ. The LVQ serves a two-fold purpose in SRT. First, it removes any multithreaded related non-determinism in the trailing thread. If the trailing thread performed the load values itself, it could get a different view of memory. Second, the LVQ keeps the memory bandwidth requirements unchanged.

DREM differs from SRT and similar work which use queues for input replication because the the leader and follower have a different view of memory. The differences arise because pointers, which are also stored in memory, are changed by the address space diversification. As a result, DREM has to perform every memory operation again in the follower threads. This introduces multithreaded related non-determinism which DREM solves with its race-recording and replay hardware.

DREM borrows on the ideas of result-queues from this work to propagate information from leader to follower core and verifying the results of computation at the follower core. DREM feeds the follower cores the addresses of committed instructions which the follower cores use to verify against their own committed addresses.

7.2 Race recorders

DREM builds on techniques found in race recorders [18, 20, 22, 36, 37]. DREM uses instruction counts and cache line tagging used in FDR [36] to label cache lines with last access information. DREM records data races for realtime replay on the followers unlike FDR which records races on persistent storage to replay them later to diagnose multithreading bugs. DREM does not use the Netzer's [23] transitive reductions for point-to-point dependencies implemented in FDR since transitively reducible dependencies do not affect replay speed. Also, DREM works with snooping coherence whereas FDR requires directory coherence. Directory coherence simplifies race recording since it stores last access information for evicted cache lines. However, current multicore processors use snooping coherence.

Strata [22] is another race recorder that uses rendezvous points to record RAW and WAW dependencies and offline post processing to reconstruct WAR dependencies. Rendezvous points minimize the recorded race log size because they transitively reduce any dependencies that cross these points. Since rendezvous points are expensive to replay, DREM minimizes their use in favor of faster replay and relies on point to point dependencies. Both DREM and Strata used bloom filters to keep track of evicted cache lines.

DeLorean [20] uses techniques in transactional memory and speculative multithreading that provide sequential consistency at release consistency speeds. DeLorean relaxes sequential consistency assumptions of previous race recorders by grouping blocks of instructions into chunks that commit atomically. This allows DeLorean to record and replay execution at speeds close to release consistency. In later work, Capo [21] uses DeLorean hardware to build a prototype of a full system recorder and replayer using Linux.

7.3 Software redundant execution

There are numerous recent projects that attempt to provide similar security guarantees to DREM with software-only solutions. They do however have limitations when it comes to supporting multithreaded applications.

DieHard [4] runs several replicas each with a randomized memory manager. The memory manager randomizes the position of heap objects in each replica. The randomized allocation provides probabilistic protection against memory corruption attacks since buffer overflows are likely to overwrite different areas in each replica. To detect an attack, DieHard compares the output from each replica to ensure that all replicas agree on their output. In case of a disagreement, an attack is signaled. DREM differs from DieHard since the amount of diversification in DREM is limited to address space diversification because DREM uses instructions for comparing replicas rather than program outputs. DREM however supports deterministic execution of multithreaded workloads.

N-Variant [11] provides a secretless framework for security through diversity by running multiple redundant replicas with diversifications that include address space layout and instruction sets. To detect memory corruption vulnerabilities, N-Variant runs two variants with disjoint address spaces. If an attack overwrites a code pointer with an absolute address, that address cannot be dereferenced in both replicas since the address spaces are disjoint. N-Variant suffers from the same drawback as DREM because it cannot detect partial pointer overwrites which allow a valid pointer to be constructed in both variants. The drawback of N-Variant is the lack of any support for multithreaded applications.

Orchestra [28] uses redundant execution in a multi-variant execution environment. Orchestra distinguishes itself from the previous work by requiring no kernel modifications. Under orchestra, all the variants are driven by a user-space process using the Linux kernel's debugging facilities. Orchestra detects stack-based overflow attacks by running two variants - one which grows the stack up and another which grows it down. Orchestra has the same limitation as N-Variant - it does not support multithreaded applications.

Replicant [25] is the only solution known to us that supports some form of redundant execution of multithreaded workloads. Replicant provides support for threading by requiring sequential annotations around areas of code that introduce thread-related non-determinism. The annotations guarantee that replicas enter and exit sequential regions in the exact same order. The annotation that Replicant requires are a major problem to its adoption - they are difficult to insert since they require a deep understanding of the application's sharing patterns. DREM solves the annotation problem since our race-recorder can replay all shared memory accesses without developer hints.

7.4 Hardware information flow tracking

Other hardware techniques exist to provide security against memory corruption attacks using pointer tainting [13, 32, 35]. Pointer tainting taints unsafe inputs such as data received on a network socket and propagates it through the system by tagging every byte of memory and registers with taint bits. If tainted data is ever used as an instruction, a pointer, or a jump address then a memory corruption attack is detected.

These approaches have much smaller runtime costs than DREM but suffer from false positives. A problem with pointer tainting arises when applications use input-derived data safely. Hardware tainting recognizes such uses and attempts to detect whenever tainted data is used safely by looking at instructions that do bounds checking on tainted data. When bounds checking is applied to tainted data, the pointer tainting policy clears the taint bits. Problems arise when tainted data is used safely without a bounds check. Detecting such uses at runtime is impossible and requires compiler annotations to tag binaries with bounds information. Removing such false positives is still an ongoing research problem.

7.5 Deterministic multithreading

An alternate way of achieving determinism in multithreaded applications is to enforce determinism in the interleaving of shared memory accesses. Kendo [24] enforces deterministic acquisition of locks in multithreaded applications. If all application communication is protected by locks then Kendo guarantees that every run of an application will produce deterministic outputs. If, however, an application performs racy shared memory access then Kendo will produce non-deterministic results. DREM has the advantage that it records and replays all shared memory communication regardless of whether it is protected by locks and enforces determinism in all cases. Also, Kendo's deterministic interleaving creates higher overheads since all threads must acquire locks in a predefined order. Under DREM, the recording phase does not constrain threads to enter locks in any predefined order.

Chapter 8

Conclusion

We find that by adding a modest amount of hardware and leveraging state tracked by the cache coherence protocol DREM can efficiently record and replay races, enabling redundant execution for security to be applied to multithreaded workloads. DREM imposes almost no additional performance penalty over a baseline redundant execution. This is due to two factors. First, out of all the memory accesses that an application performs, every few result in recorded dependencies. The majority of shared memory accesses have their dependencies transitively reduced by rendezvous points. Second, even when a dependency results in a stall, the stalls are relatively short and last several hundred cycles on average. In addition, most of the dependencies that actually result in stalls occur at synchronization points. These factors allow DREM to impose less than 1% execution overhead over a base overhead of 22% for executing two replicas redundantly.

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[2] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), 1996.

[3] Todd M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 196–207, November 1999.

[4] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. *SIGPLAN Not.*, 41(6):158–168, 2006.

[5] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. NonStop advanced architecture. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 12–21, June 2005.

[6] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286, August 2005.

[7] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *CCS '08:*

Proceedings of the 15th ACM conference on Computer and communications security, pages 27–38, New York, NY, USA, 2008. ACM.

[8] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: bulk enforcement of sequential consistency. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, pages 278–289, June 2007.

[9] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, January 1998.

[10] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. Formatguard: automatic protection from printf format string vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 15–15, Berkeley, CA, USA, 2001. USENIX Association.

[11] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-Variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*, pages 105–120, August 2006.

[12] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, August 1998.

[13] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, pages 482–493, June 2007.

[14] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the*

4th International Conference on Virtual Execution Environments (VEE), pages 121–130, March 2007.

[15] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse computer systems. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, pages 67–72, May 1997.

[16] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. *SIGOPS Oper. Syst. Rev.*, 25(Special Issue):245–257, 1991.

[17] John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2003.

[18] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 265–276, June 2008.

[19] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[20] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 289–300, June 2008.

[21] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 73–84, March 2009.

[22] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 229–240, December 2006.

[23] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, pages 1–11. ACM, 1993.

[24] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 97–108. ACM, 2009.

[25] Jesse Pool, Ian Sin Kwok Wong, and David Lie. Relaxed determinism: making redundant execution on multiprocessors practical. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, pages 25–30, May 2007.

[26] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, pages 25–36, June 2000.

[27] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[28] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *EuroSys '09: Proceedings of the fourth ACM european conference on Computer systems*, pages 33–46, New York, NY, USA, 2009. ACM.

[29] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, October 2004.

[30] Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proceedings of the 39th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 223–234, December 2006.

[31] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, October 2004.

[32] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture (HPCA 2008)*, pages 173–184, February 2008.

[33] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, pages 87–98. IEEE Computer Society, 2002.

[34] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture (ISCA)*, pages 24–36, 1995.

[35] Jun Xu and Nithin Nakka. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 378–387. IEEE Computer Society, June 2005.

[36] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, pages 122–135, June 2003.

[37] Min Xu, Mark D. Hill, and Rastislav Bodik. A regulated transitive reduction (rtr) for longer memory race recording. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 49–60, 2006.