Kernel Support for Deterministic Redundant Execution of Shared Memory Workloads on Multiprocessor Systems

by

Jesse Pool

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

Abstract

Kernel Support for Deterministic Redundant Execution of Shared Memory Workloads on Multiprocessor Systems

Jesse Pool

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2007

Motivated by future processors that will contain an abundance of execution cores, we believe redundant execution will be a practical method for increasing system security and reliability. However, redundant execution relies on the premise that duplicating external inputs identically to a set of replicas will produce identical outputs. Unfortunately, multi-threaded applications exhibit non-determinism that breaks this premise, especially on the multiprocessors that will be widely available in the future.

This thesis presents a method for deterministically replicating the accesses to shared memory made by concurrent threads in a kernel level redundant execution system. Our approach relies on user space annotations, which define *sequential regions* in the application that the kernel will schedule deterministically. Further, we show that these annotations can be largely derived from the use of locks already present in the application and that replication can be achieved with only modest overhead.

Acknowledgements

I would first like to thank Professor David Lie for his tireless contributions to this thesis. Our many discussions and analysis sessions formed the foundation of my work. Thank you for pushing me to always be at my best.

I also acknowledge the contributions of Ian Sin, with whom I worked on many aspects of this thesis. Our debates and time at the whiteboard were of great value to me and I will always look back on those times with fond regard.

For their financial assistance, I thank the Natural Sciences and Engineering Research Council of Canada (NSERC), as well as the University of Toronto and The Edward S. Rogers Sr. Department of Electrical and Computer Engineering.

To those who gave helpful comments on early versions of this work, I kindly thank. Specifically, Lionel Litty, Tom Hart, Richard Tamin and Ashvin Goel for being consistently critical, as well as all of the members of the Security Reading Group (SRG) and the members of the System Software Reading Group (SSRG).

Finally, I thank my friends and family for their much needed support throughout my studies. I have achieved all that I have achieved thanks to you. I especially thank Marta Antoszkiewicz for her ever celebrating spirit. Thank you for helping me take joy in every accomplishment, big or small.

Contents

List of Tables

List of Figures

List of Acronyms

Acronym	Definition
API	Abstract Programming Interface
ASLR	Address Space Layout Randomization
CMP	Chip-level Multiprocessor
COW	Copy on Write
CPU	Central Processing Unit
FSA	Finite State Automaton
GTK	The GIMP Toolkit
IPI	Inter-processor Interrupt
MMU	Memory Management Unit
MPM	Multi-Process Module
NPTL	Native POSIX Thread Library
OS	Operating System
PCB	Process Control Block
PID	Process Identifier
POSIX	Portable Operating System Interface
SMP	Symmetric Multiprocessor
TLB	Translation Lookaside Buffer
VMM	Virtual Machine Monitor

Chapter 1

Introduction

The recent trend toward commodity multiprocessor hardware, in the form of multi-core chips, has encouraged the research community to search for methods to take advantage of the newly available processing power. We feel that, although multiprocessors will allow for improved performance on commodity hardware, they are also attractive for redundant execution applications. Redundant execution maps naturally to multiprocessors because it is inherently parallelizable. As a result, redundancy provides an immediate application for under-utilized execution cores.

Kernel level redundant execution is conceptually straightforward. A redundant execution system runs several *replicas* of an application simultaneously and provides each replica with identical inputs from the underlying operating system (OS). The redundant execution system, operating at the system call interface, compares the outputs of each replica. It relies on the premise that replica inputs are deterministic, so that any divergence in their outputs must indicate an error. For example, executing identical replicas has been used to detect and mitigate soft-errors from the kernel [4] and a similar approach has been shown to work from user space as well [3, 43]. There have also been several proposals to use kernel level redundant execution to execute slightly different replicas to detect security compromises [14] and private information leaks [68]. In the latter two

projects, injecting carefully chosen differences among replicas will cause them to produce divergent outputs only when some violation has occurred.

Although the growing prevalence of multiprocessors will encourage the use of multi-threaded programming, kernel level redundant execution systems to date have not been able to efficiently support multi-threaded programs on multiprocessor systems. This is because the relative rates of thread execution among processors are non-deterministic, making inter-thread communication difficult to duplicate precisely in all replicas, especially when the communication is made through shared memory. Allowing the order of shared memory communication to diverge among replicas can cause a *spurious divergence*, which is not the result of a failure or violation. The inability to efficiently deal with the non-determinism that exists when running multi-threaded programs on multiprocessors threatens the future feasibility of kernel level redundant execution systems.

Intuitively, when all accesses made to shared memory are done so deterministically in an application, the outputs derived from those accesses will also be deterministic. A kernel level redundant execution system must ensure that all replicas access shared memory deterministically, so that no spurious divergences can occur as a result of non-deterministic thread scheduling on a multiprocessor system. While hardware support for memory page protection may be used by the operating system kernel to synchronize access to shared memory, this would require that replicated threads be interrupted on all memory accesses and would be prohibitively expensive. Rather than examine and faithfully replicate every memory access, we rely on the application to inform the kernel when it is accessing shared memory and would like access ordering to be replicated. Annotation directed shared memory interposition by the OS kernel allows our system to achieve deterministic replication without the high overhead cost of thread interruption on every memory access. Our system also avoids the repeated cost of altering memory page protection bits, which is an expensive operation on multiprocessor systems. Annotations placed in the target application, either by manually editing the source code or through

automatic inference, call into the kernel before and after each explicit access to shared memory is made. The kernel may then force a thread to wait if an attempt is made to access a shared memory region out of turn. Annotated sections of code that access shared memory are referred to as *sequential regions*. The order in which threads execute these sections of code is deterministically duplicated for all replicas in the system.

We have implemented a kernel level redundant execution system, *Replicant*, to evaluate the feasibility of annotation directed deterministic redundant execution of shared memory workloads on multiprocessor systems. Preliminary work on Replicant can be found in [48] and a detailed description of its replication architecture can be found in [67]. Replicant is implemented as a modification to the Linux™ 2.6.16 kernel [5, 37], allowing us to evaluate our approach with several representative multi-threaded applications, including: Apache HTTP Server, Squid Web Proxy Cache and MySQL database server. This thesis will focus on the architecture, implementation and evaluation of Replicant's shared memory annotation mechanisms.

1.1 Contributions

This thesis makes three contributions to the research area of kernel level redundant execution systems:

1. The evaluation of several large scale, multi-threaded server applications and their shared memory non-determinism properties.

2. The development of an architecture for shared memory annotation that allows deterministic redundant execution, including a definition of guarantees that such a system must provide to ensure determinism.

3. The implementation, evaluation and analysis of a working system that is able to redundantly execute several representative large scale server applications.

1.2 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 will provide the reader with relevant domain background. Chapter 3 will present our problem description and briefly describe Replicant, our kernel level redundant execution engine. We describe a method for annotation directed deterministic redundant execution of shared memory workloads in Chapter 4 and follow with a detailed description of our 2-replica prototype implementation in Chapter 5. Chapter 6 will enumerate the multi-threaded applications that we have ported for execution on Replicant and Chapter 7 will evaluate their performance. A literature survey is provided in Chapter 8, which describes the many works on which this thesis derived motivation and inspiration. We highlight several areas of interesting future work and potential improvements to our prototype implementation in Chapter 9. Finally, we conclude in Chapter 10.

Chapter 2

Background

In order to provide the reader with relevant domain background, this chapter will give an overview of redundant execution and its uses as well as describe shared memory work loads and multiprocessor systems in the context of this thesis.

2.1 Redundant Execution

Redundant execution has long been studied in an attempt to promote availability [20] and fault tolerance [34]. Availability can be achieved by maintaining several, potentially distributed, instances of an application and its data. In the event of a failure in the primary instance, there is at least one backup that can brought online in its place, limiting system downtime perceived by an observer. In the case of fault tolerance, a consensus of the majority can be taken before performing some critical action, ensuring that the effects of a faulty replica remain isolated. Availability and fault tolerance have been naturally achieved in user space and implemented as cooperation protocols between replicas [32, 33]. Redundant execution may also be supported by the operating system kernel [4] or hypervisor [6] to achieve the same goals. However, they are implemented beneath the application and are generally not distributed. Moreover, redundant execution systems that exist at a layer below the application may require little or no application

support for replication.

There has been a recent effort to use redundant execution, along with control flow independent application diversity, such as Address Space Layout Randomization (ASLR) [47], to achieve security guarantees that cannot be made by running a single application instance. These redundant execution systems exist in the operating system kernel. It is an important consideration for security mechanisms to be isolated from the application and a layer below approach is generally considered more secure (i.e. either in the kernel or in the hypervisor). When redundant execution protocols are implemented as part of the application, or as a module in the same address space, a subverted application would mean the redundant execution system is compromised.

Replicated instances of an application can be compared for consistency in the kernel, where a divergence in outputs or control flow can indicate a security compromise. Consider the N-Variant system [14]. n instances of the same binary are started and inputs from the operating system are replicated to each instance. The instances, called variants, contain some diversity such that a malicious input can only compromise one of the n instances. As a result, a malicious input will cause a divergence and the security compromise can be flagged. TightLip is another system that attempts to achieve security through redundant execution. It operates on the same principles as N-Variant, but aims to protect user privacy [68]. These applications are more closely examined in Chapter 8, but an important limitation of previous work in this area is a lack of support for shared memory workloads on multiprocessor systems.

The premise of redundant execution is that, given deterministic inputs, an application will reproduce deterministic output. Unfortunately, security applications rely on some element of non-determinism, such that an attack on one application instance may succeed, while the same attack fails on another instance. However, all non-malicious inputs must result in deterministic output. The goal of a redundant execution system, targeted at promoting security guarantees, is to remove any non-determinism in outputs, while

providing enough diversity to effectively prevent a compromise from malicious input. A widely used method for adding inexpensive, application agnostic diversity is through the use of ASLR [53]. Other forms of diversity include, but are not limited to, instruction set tagging [14], changing complier optimization levels and varying library versions.

Our work focuses on redundant execution implemented in the operating system kernel to support security guarantees. Our system is not distributed, but rather allows for several instances of an application to execute concurrently on the same machine. Each instance on its own may be vulnerable to attack, but the replicated system as a whole can detect some important classes of malicious input. Further, our system supports applications that communicate through shared memory and also supports this execution on multiprocessor systems [48, 67].

2.2 Programming Concurrency

As multiprocessor systems move to commodity hardware architectures, developers will attempt to take advantage of the additional parallelism by refactoring applications to operate in parallel units. Efficient applications are typically parallelized into several threads of execution, which share resources that are protected by a synchronization mechanism.

Thread programming is non-trivial because the developer may encounter synchronization issues that lead to race conditions, deadlocks, and starvation. Further, multi-threaded application development will often require care due to issues such as lock granularity, non-determinism and awkward debugging. While multi-threading with synchronized shared memory access is generally considered difficult to program, it is currently the most common method of achieving concurrency in application development.

Another efficient programming paradigm is event driven programming [45]. However, on its own, an event driven application cannot effectively take advantage of a multiprocessor system due to a lack of parallelism. Alternatively, the transactional memory [28, 55]

paradigm may address some of the short comings of synchronized communication between threads, such as deadlocks. However, transactional memory is not currently being widely used in practice and so our work focuses primarily on multi-threaded applications with synchronized access to shared memory.

The following sections describe multi-threaded programming on Linux and provide some insight into how the internal concurrency constructs are implemented.

2.2.1 Concurrency, Shared Memory and Locking

There is a long history of using several concurrent execution contexts with shared resources for achieving parallelism. The Dining Philosophers Problem [15], which was introduced in the early 1970s, continues to be a part of computer science curricula today.

The difference between threads and other execution contexts, such a processes, is that threads typically share more of their resources. This includes, but is not limited to, address space, the file descriptor table and signal handlers. There are several advantages to allowing resource sharing, including performance benefits. However, when several threads have access to shared resources, the developer must ensure that these accesses are synchronized. Synchronization refers to protections placed around shared resources such that concurrent accesses do not corrupt data or place the system in an inconsistent state. For example, one shared resource may require mutually exclusive access, while another will allow any number of concurrent readers. The synchronization mechanism protecting a shared resource will depend on the type of resource and the semantics of the application in its use of the resource. Performing synchronized access to a shared resource is often the primary cause for error and complexity in multi-threaded applications. A by-product of synchronizing access to shared resources is that successive runs of the same application may not be deterministic. As a result, debugging multi-threaded applications can also be complex and tedious.

Threaded applications running on UNIX and UNIX-like platforms will most often

require support for POSIX Threads (IEEE Std 1003.1c) from the operating system and shared libraries [26]. POSIX Threads, or Pthreads, is a standard Abstract Programming Interface (API) for creating and manipulating threads of execution. In order to allow synchronized access to shared resources, the Pthreads standard also provides several constructs that allow a developer to implement portable resource sharing. This includes several variations of mutual exclusion. Figure 2.1 depicts the C language definitions for several representative Pthread functions. [26] provides a complete list of function prototypes in the Pthreads standard.

2.2.2 Native POSIX Thread Library

The Native POSIX Thread Library (NPTL) should not be confused with Pthreads in general, discussed in the previous section. NPTL is a specific implementation of the Pthreads standard that has been targeted to take advantage of specific features in the Linux kernel.

Early Pthreads support on the Linux platform was implemented in the form of LinuxThreads as part of the GNU C Library, `glibc`. This was an entirely user space implementation which included a manager thread to perform tasks such as handling fatal signals and managing some aspects of memory. Although individual threads did share an address space through the `clone` system call, LinuxThreads was inefficient and lacked complete POSIX compliance [56, 16]. Inefficiencies were largely related to synchronization primitives that were implemented with signals as well as having a thread creation bottleneck at the manager thread. Compliance with the POSIX standard was incomplete due to incorrect signal behaviour, incorrect process identification as well as a limit on the total number of threads – 8192, minus one for the manager [16].

The goals for NPTL were to improve upon the LinuxThreads implementation with complete POSIX compliance, effective use of SMP, low startup costs and scalability. NPTL does not include a manager thread, which has its functionality moved into the

kernel along with POSIX compliant signal support. As a result, startup cost is reduced because thread creation is no longer serialized. By moving functionality into the kernel, POSIX compliant synchronization is also possible, which will be discussed in the following section. Applications can link in the `libpthread` library on Linux to use the the NPTL Pthreads implementation. `libpthread` is compiled as part of the `glibc` source code, which contains many individual libraries compiled along side the Standard C Library. We will refer to `libpthread` simply as part of `glibc` for the remainder of this thesis.

There are several aspects of the NPTL implementation that are relevant to a redundant execution system implementation based on the Linux 2.6 kernel. (1) There is a 1-on-1 user thread to kernel thread relationship. (2) Synchronization is implemented through shared memory primitives that rely on the kernel to arbitrate only on contention [16].

A 1-on-1 user thread to kernel thread implementation means that for every user space thread there is a corresponding kernel space representation (a task in the case of Linux). This is in contrast with an m-on-n approach where several user space threads are scheduled on top of one kernel thread, or process. Had NPTL used a m-on-n design, a kernel level redundant execution system would be unaware of some of the user space threads, potentially leading to added complexity. Because each user space thread has a corresponding kernel space representation, our system can maintain per-thread state that may have been complicated otherwise.

In NPTL, synchronization primitives are designed so that traps to the kernel are only incurred when there is contention. As a result, the kernel may or may not be able to see the result of a lock acquisition. Without being informed of which thread has acquired a lock in one replica, other replicas cannot be deterministically replicated and may diverge.

2.2.3 Fast Userspace Mutex

The Fast Userspace Mutex, or futex, is an alternative to the more classical, heavy weight mechanisms provided by UNIX-like operating systems to support process synchroniza-

tion. Traditional synchronization is provided through System V semaphores, file locks and message queues, which all require system calls on each lock access [22]. With a futex, the kernel is only called on a contended operation, which should reduce overhead on a lightly contended workload, or when locks are sufficiently fine grained.

In Linux, a futex is an integer value in user space, shared among a number of threads or processes. A corresponding system call, called `sys_futex`, operates on this integer to arbitrate contended access. The futex system call is wrapped by a library, NPTL in `glibc`, with atomic operations that perform abstractions to the futex interface for portability. The futex construct is used for many of the Pthreads operations on Linux. These include implementations for locking and unlocking a mutually exclusive region with the `pthread_mutex_*` family of functions, as well as to implement conditional variable operations such as `pthread_cond_wait`, `pthread_cond_signal` and `pthread_cond_broadcast`. Internally the kernel maintains a single, system wide, static hashtable that is indexed by several properties of the futex, such as its offset into a page. Waiting threads enqueue themselves on a waitqueue and are appended to a linked list. When the futex is woken, threads are dequeued in the same order that they were enqueued. Tasks which are waiting on a futex are set to an interruptible state, which allows them to be dequeued if a signal is pending. In this case, the futex system call will return a status code indicating that it was interrupted (EINTR) and the NPTL library code will automatically re-execute the system call. Note that this will give the opportunity for any pending signal to execute its handler.

The futex mechanism presents a challenge to redundant execution systems in general. Since futex library calls operate non-deterministically on shared memory, some futex calls may never require kernel involvement, such as when a lock is uncontended. However, a redundant execution system may require determinism in futex operations, as these operations may be directly involved in functions that will produce replica divergence. As a result, any kernel level redundant execution system will need to enforce at least some

determinism in futex operations, which are a special case of shared memory access.

2.3 Multiprocessor Systems

Multiprocessor systems are computing platforms that contain two or more central processing units (CPU). There are many varieties of multiprocessing platforms, but we are primarily concerned with symmetric multiprocessors (SMP) and chip-level multiprocessors (CMP). These are the types of multiprocessors that are typically available on commodity hardware found in large scale enterprise servers and even modern personal computers. In an SMP system, several CPUs are available to the operating system that are essentially identical. Although there may be some efficiency advantages to being selective, the operating system is free to schedule tasks on these CPUs interchangeably and concurrently. CMPs are SMPs where several processing cores are packaged onto a single chip. There may be cache size, communication channel and other differences, but these differences are not relevant for this thesis.

Multiprocessing is an important element in the design of a kernel level redundant execution system for several reasons: (1) it is inherently more difficult to enforce determinism on a multiprocessor and (2) redundant execution can benefit from the added parallelism of a multiprocessor system by making use of unused processing cycles. These two opposing topics will be discussed throughout the remainder of this thesis.

```
1  /* Thread Creation/Destruction */
2  int pthread_create(pthread_t *restrict, const pthread_attr_t
3          *restrict, void *(*)(void *), void *restrict);
4  void pthread_exit(void *);
5  int pthread_join(pthread_t, void **);
6
7  /* Mutual Exclution */
8  int pthread_mutex_destroy(pthread_mutex_t *);
9  int pthread_mutex_init(pthread_mutex_t *restrict, const
10         pthread_mutexattr_t *restrict);
11 int pthread_mutex_lock(pthread_mutex_t *);
12 int pthread_mutex_trylock(pthread_mutex_t *);
13 int pthread_mutex_unlock(pthread_mutex_t *);
14
15 /* Read-write Locks */
16 int pthread_rwlock_destroy(pthread_rwlock_t *);
17 int pthread_rwlock_init(pthread_rwlock_t *restrict, const
18         pthread_rwlockattr_t *restrict);
19 int pthread_rwlock_rdlock(pthread_rwlock_t *);
20 int pthread_rwlock_wrlock(pthread_rwlock_t *);
21 int pthread_rwlock_unlock(pthread_rwlock_t *);
22
23 /* Conditional Variables */
24 int pthread_cond_destroy(pthread_cond_t *);
25 int pthread_cond_broadcast(pthread_cond_t *);
26 int pthread_cond_init(pthread_cond_t *restrict, const
27         pthread_condattr_t *restrict);
28 int pthread_cond_signal(pthread_cond_t *);
29 int pthread_cond_wait(pthread_cond_t *restrict, pthread_mutex_t
30         *restrict);
```

Figure 2.1: Pthreads Function Prototypes. Several representative functions from the POSIX Threads API. While these are among the most often used functions, the POSIX Threads standard interface contains many more less frequently used operations.

Chapter 3

Overview

This chapter highlights the difficulties inherent in redundant execution of shared memory workloads, and describes the platform we have implemented to evaluate the design of a working solution.

3.1 Problem Description

Redundant execution systems rely on the presumption that if inputs are copied faithfully to all replicas, any divergence in outputs among replicas must be due to undesirable behaviour, such as a transient error or a malicious attack. On such systems, the replication of inputs and comparison of outputs are done in the OS kernel, which can easily interpose between an application and the external world, such as the user or another application on the system. However, since inter-thread communication through shared memory is invisible to the kernel, and relative thread execution rates on different processors are non-deterministic, events among concurrent threads in a program cannot be replicated precisely and efficiently, leading to spurious divergences.

To illustrate, consider the scenario described in Figure 3.2. Three threads each add their thread ID to a shared variable, `counter`, make a local copy of the variable in `local`, and then print out the local copy. However, the threads may update and print the counter

in a non-deterministic order between the two replicas. Assume we have a 2-replica system where in Replica 1 threads execute the locked section in the order (1, 3, 2) by thread ID. Then the printed output would be the values "1", "4" and "6". If the threads in Replica 2 execute the locked section in the order (2, 3, 1), then its printed output would contain the values "2", "5" and "6". This example demonstrates that multi-threaded applications may generate output values based on the ordered access to shared memory regions.

To avoid these spurious divergences, the redundant execution system must ensure that the ordering of updates to the counter is the same between the two replicas. If the redundant execution system ensures that threads enter the locked region in the same order in both replicas, then both replicas will produce the same outputs, though possibly in different orders. If the system further forces the replicas to also execute `printf` in the same order, then both the values and order of the outputs will be identical.

A simple solution might be to make accesses to shared memory visible to the OS kernel, by configuring the hardware processor's memory management unit (MMU) to trap on every access to a shared memory region. For example, since `counter` is a shared variable, we would configure the MMU to trap on every access to the page where `counter` is located. However, trapping on every shared memory access would be very detrimental to performance, and the coarse granularity of a hardware page would cause unnecessary traps when unrelated variables stored on the same page as `counter` are accessed.

A more sophisticated method is to replicate the delivery of timer interrupts to make scheduling identical on all replicas. While communication through memory is still invisible to the kernel, duplicating the scheduling among replicas means that their respective threads will access the counter variable in the same order, thus resulting in the exact same outputs. Replicating the timing of interrupts is what allows systems like ReVirt [17] and Flashback [60] to deterministically replay multi-threaded workloads. Unfortunately, as the authors of those systems point out, this mechanism only works when all threads are

scheduled on a single physical processor and does not enable replay on a multiprocessor system. This is because threads execute at arbitrary rates relative to each other on a multiprocessor and as a result, there is no way to guarantee that all threads will be in the same state when an event recorded in one replica is replayed on another.

Finally, a heavy-handed solution might be to implement hardware support that enforces instruction-level lock-stepping of threads across all processors. Unfortunately, this goes against one of the primary motivations for having multiple cores, which is to reduce the amount of global on-chip communication. In addition, it reduces the opportunities for concurrency among cores, resulting in an unacceptably high cost to performance. To illustrate, a stall due to a cache miss or a branch misprediction on one core will also stall all the other cores in a replica.

In summary, to support multi-threaded applications on a multiprocessor architecture, the redundant execution system must be able to handle outputs produced in non-deterministically different orders among replicas. The redundant execution system must also be able to deal with the non-deterministic ordering of communication among replicas, which may result in divergent replica output values. In both cases, the system must either enforce the necessary determinism at the cost of some lost concurrency, or it must find ways to tolerate the non-determinism without mistaking it for a violation.

3.2 Argument for Annotations

We have outlined the non-determinism issues inherent with redundant execution of shared memory workloads in the previous section. Non-deterministic outputs may be produced when interleaving threads of control access shared memory in different orders. Therefore, the ordering of shared memory accesses must be deterministically reproduced in all replicas. Unfortunately, kernel level redundant execution systems have limited visibility into accesses made to shared memory by application level threads.

Application	Threads	Mappings		Pages	
		Writable	Total	Writable	Total
Apache 2.2.3	64	88	176	169376	170057
Squid 2.3.STABLE9	16	35	71	1999	2620
MySQL 5.0.25	8	30	61	23258	24810

Table 3.1: Writable Application Pages. The number of writable shared pages mapped into several representative applications. Application versions and configurations are equivalent to those described in Section 7.1.2.

Ideally, hardware support should be used to allow the operating system to interpose on scheduled access to shared memory. As described in the previous section, hardware page protection may be used by the operating system to trap on access to shared memory. The results of each instruction can be buffered and then replicated. Unfortunately, this solution is prohibitively expensive, because the application would need to be interrupted on every access to writable memory.

A default configuration of the Apache HTTP Server 2.2.3, with the worker Multi-Process Module (MPM) enabled and 64 threads, contains 169376 writable memory pages in 88 mappings after start up. Any instruction that accesses memory on these pages would require a trap to the kernel for interposition. Table 3.1 lists the number of writable pages in several representative applications taken immediately after start up. Although most of these pages will never be shared among threads, the kernel cannot reason about which pages should be protected without resorting to heuristics or annotations. In either case, the kernel would need to have some information about how memory is shared in the application. Further, manipulating hardware page protection is an expensive operation on x86 multiprocessors. Each time the protection is modified by one processor, an Inter-processor Interrupt (IPI) is triggered so that any cached reference to the modified page on other processors may be flushed. This includes flushing the Translation Lookaside Buffer (TLB).

Hardware support has been used in the past to facilitate deterministic replaying of shared memory applications [6, 17]. In this work hardware branch counters, along with the program counter, were recorded every time an interrupt was delivered. During replay, the recorded counters were referenced to determine the exact instruction at which an interrupt should be delivered. As a result, thread access to shared memory is scheduled deterministically. However, this mechanism cannot be applied for replay on multiprocessors. As we noted in the previous section, threads execute at arbitrary rates relative to each other on a multiprocessor and the counters used in replay systems cannot be used to reason about interrupt delivery.

At present, no hardware solution exists that will allow the deterministic replication of shared memory access on a multiprocessor with acceptable overhead. The result is a dependence on the user space application to inform the kernel of accesses to shared memory. This is a disadvantage, because it requires developer involvement in the form of annotations to the application. However, there is also the benefit that annotations may only be added when needed in order to enforce determinism rather than at every access to shared memory – allowing for acceptable performance overhead. We will also show that development effort can be minimized by automatically inferring the placement of annotations.

3.3 Replicant

In order to explore the challenges of redundant execution of shared memory workloads on multiprocessor systems, we have implemented a novel redundant execution engine as an extension to the Linux 2.6.16 kernel. Our system, called Replicant, implements an input replicating and output matching architecture that is tolerant to the reordering of events, and uses determinism annotations to enforce ordering on events that can cause divergence in replica output values. While the focus of this thesis is on deterministic shared memory

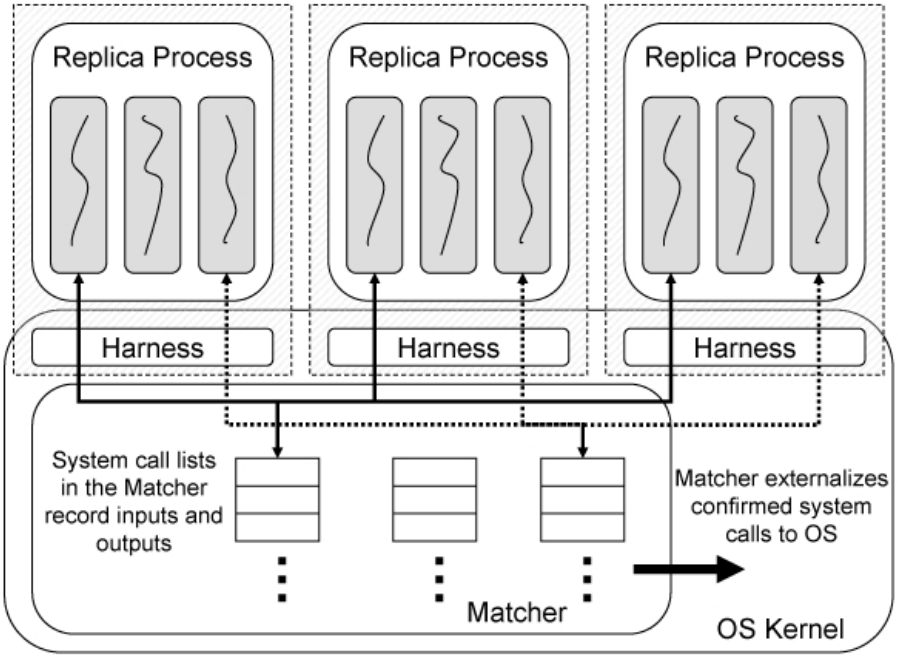

Figure 3.1: The Replicant Architecture. Replicant allows each replica to execute independently with the support of a harness. Outputs are only externalized by the matcher.

workloads, this section will give a high level overview of the larger Replicant system. A detailed description of Replicant's implementation can be found in [48] and [67].

The architecture of Replicant is described in Figure 3.1. A unique aspect of Replicant is that it permits replicas to execute independently of each other and diverge in their behaviour. However, only outputs that a majority of replicas have *confirmed* (i.e. independently reproduced) are externalized outside of the redundant execution system. To allow replicas to execute independently, Replicant places each replica in an OS sandbox, called a *harness*, where the replica can make changes to the OS state that are only visible to the replica itself. The harness is kept up-to-date by applying the outputs and effects of all system calls to a copy-on-write (COW) file system provided to each replica, as well as private copies of other OS state. With these facilities, the Replicant harness emulates the underlying OS with enough fidelity that the replica is not aware that its outputs are being buffered.

Replicant also adds a *matcher* component to the OS kernel for each set of replicas.

The purpose of the matcher is to fetch and replicate inputs from the external world into the harness, and determine when outputs from the harness should be made externally visible. The matcher is implemented as a set of system call lists that are used to buffer the arguments and results of system calls made by the replicas. Each thread in a replica is associated with exactly one thread in every other replica, and this group of threads forms a *peer group* across all replicas. Threads in a peer group are all created by the same thread creation event and share a system call list in the matcher. In the example given in Figure 3.2, threads with the same thread ID form a peer group across the replicas. A new thread is not allocated a system call list and is not permitted to run until a majority of threads in its parent's peer group have also created a new thread. At this point, the thread creation event is confirmed, a new peer group is formed, a new system call list is allocated, and the new group will be permitted to execute and confirm system calls.

Replicant allows each replica to diverge in its execution and will only commit outputs that are independently reproduced by the majority of replicas. However, because some non-deterministic events can cause spurious divergences that cannot be matched, Replicant requires some user space annotations to produce correct results. One such annotation is the *sequential region*. A sequential region ensures that thread execution of specified code regions is performed deterministically across all replica instances. As a result Replicant is able to support multi-threaded workloads that use shared memory as a communication mechanism between threads. This is an important class of application, that includes many server workloads. Further, Replicant is also able to support these workloads on multiprocessor architectures, which present a unique set of problems as described in the previous section. The remainder of this thesis will focus on the design and implementation of sequential regions. The implementation details of Replicant's redundant execution mechanisms and architecture are outside the scope of this thesis, but are described in detail in [48] and [67].

```
1  /* A global counter that is incremented by all threads. */
2  int counter = 0;
3
4  /* Functions that are used as a big lock to grab all memory. */
5  extern global_lock(void);
6  extern global_unlock(void);
7
8  /* Each thread acquires a global lock and adds its identifier to
9     a global counter. A local copy is printed. */
10 void thread_start (void *arg)
11 {
12         int local;
13
14         global_lock();
15         counter = counter + thread_id();
16         local = counter;
17         global_unlock();
18
19         printf(``%d\n'', local);
20 }
21
22 /* Create three threads that execute the thread_start routine. */
23 void  main (void)
24 {
25         thread_create(thread_start); /* thread_id = 1 */
26         thread_create(thread_start); /* thread_id = 2 */
27         thread_create(thread_start); /* thread_id = 3 */
28 }
```

Figure 3.2: Non-deterministic Threaded Application. Pseudo code example illustrating non-determinism in a multi-threaded program. Two independent executions of this application will often result in divergent outputs.

Chapter 4

Architecture

When inputs are duplicated to all replicas in a single threaded application, deterministic execution will result. Similarly, multi-threaded applications which do not communicate can be replicated using traditional methods. However, in order to replicate shared memory workloads, where non-determinism can result in diverging executions, alternative mechanisms are needed.

Section 3.1 described why redundant execution of shared memory workloads is a difficult problem, and also explained why this problem is exacerbated when shared memory workloads are executed on a multiprocessor architecture. We have implemented a kernel level redundant execution system that can replicate shared memory workloads by enforcing order on memory operations. This chapter outlines our method for enforcing deterministic execution of shared memory workloads.

4.1 Design Considerations

In order to implement a mechanism that both allows redundant execution of shared memory workloads and is practical in modern systems, we have defined several goals which our implementation must meet.

Multiprocessor support. The future of the commodity processor architecture will be multi-core or many-core [29]. As such, it is important that our system be generalizable to any number of physical or logical processors. Further, executing redundant instances of an application will typically result in some measurable loss in performance. Multi-core systems will be advantageous for Replicant, because the likelihood of idle cycles is increased. Redundant execution systems in general should be able to make use of available processing power offered by running on under-used cores, limiting performance overheads. The runtime overhead of our system is evaluated on a multiprocessor platform in Section 7.1.

Intuitive annotation. From the discussion in Sections 3.1 and 3.2, we have argued that user space annotations will provide the best method for enforcing ordered access to shared memory. Since we will be calling on a developer to add annotations to his or her application, we want to make the process of adding annotations intuitive. As discussed in Section 2.2, POSIX Threads provides the most common API for dealing with shared memory on Linux, and UNIX systems in general. We would like our annotations to intuitively map to locking constructs available in the Pthreads library. This should allow our system to integrate with existing server applications targeted to UNIX platforms more easily. The ability to port existing applications to our system with proper annotations is evaluated in Chapter 6.

Generally applicable. There are several recent redundant execution systems, targeting security, that are unable to support multi-threaded shared memory workloads on multiprocessors [14, 68]. Our implementation should be equally well applicable to these systems. Achieving this goal will demonstrate a clear contribution to the research community. We evaluate the porting of our shared memory determinism mechanism to another redundant execution system in Section 7.2.

```
1  /* A global counter that is incremented by all threads. */
2  int counter = 0;
3
4  /* Each thread acquires a global lock and adds its identifier to
5     a global counter. A local copy is printed. */
6  void thread_start (void *arg)
7  {
8          int local;
9
10         BEGIN_SEQ_REGION;
11         global_lock();
12         counter = counter + thread_id();
13         local = counter;
14         global_unlock();
15         END_SEQ_REGION;
16
17         printf(``%d\n'', local);
18 }
```

Figure 4.1: Annotated Threaded Application. Pseudo code example illustrating a how Figure 3.2 may be manually annotated.

4.2 Source Code Annotations

In order to satisfy our design goals, outlined in the previous section, we have implemented a user space annotation mechanism, which provides information to the operating system kernel allowing it to duplicate shared memory access ordering across replicas. The application developer can explicitly identify accesses to shared memory whose ordering must be deterministically replicated by annotating the corresponding code with a *sequential region*. A sequential region ensures that thread interleaving through the specified code segment is performed deterministically across all replica instances.

In the example given in Figure 3.2, the developer places a sequential region around the critical section bounded by the `lock` and `unlock` operations at `line 14` and `line 17`, respectively. This ensures that corresponding threads in each replica pass through this region in the same order. Thus, each individual thread will produce the same output value.

In order for an application to be completely annotated, a sequential region must be placed around any shared memory operation that has a direct or indirect effect on the externally visible output of the application. Although it is admittedly difficult to generalize a minimal annotation scheme, a replicated application will execute correctly if sequential regions are used more often than needed. Initially, applications should have all shared memory accesses annotated, which will ensure correctness. Annotations can be removed on a case-by-case basis to improve performance.

Definition 4.2.1. *A sequential region is a section of code, delimited by annotations, where the operating system kernel will guarantee deterministic thread execution ordering across all replicas.*

Sequential regions are identified to the kernel by annotating the source code of an application. The annotations are translated into system calls that are placed in the code and executed at runtime. Figure 3.2 illustrates a key insight that makes it easier to place sequential region in source code – accesses to shared variables are typically protected by critical sections defined by `lock` and `unlock` pairs. Further, lock structures, such as the Pthreads standard `pthread_mutex_t` structure, are shared variables themselves. Based on this observation, placing sequential regions around critical sections already defined in the application source code provides a logical mapping. Conceptually, the developer uses the annotations by placing a `BEGIN_SEQ_REGION;` whenever a critical section begins, such as before the `lock` statement in Figure 3.2, and an `END_SEQ_REGION;` whenever the critical section ends, such as right after the corresponding `unlock`. Figure 4.1 demonstrates how the previous example can be manually annotated so that Replicant will ensure deterministic access to shared memory. When executed on the Replicant kernel, the annotated application will produce deterministic replica executions.

Replicant allows individual replicas to diverge in execution, provided that their externally visible outputs are consistent (e.g. writes made to a file must be deterministic). As such, sequential regions need only be inserted if the ordering of events can affect

externally visible outputs. For example, if the program in Figure 4.1 did not print the intermediate values of the shared `counter` variable on `line 17`, but instead only printed the final value after all threads had completed, then no sequential regions would be needed. This is because the thread ordering no longer has any effect on the application output. Since sequential regions enforce ordering of threads within a replica, they can reduce opportunities for concurrency, and should be used only when necessary. There are many events whose ordering will not have an effect on the values of outputs, such as calls to `malloc`, the heap allocator. Although these non-determinisms will result in inconsistencies across replicas, they will not effect Replicant's ability to match output values, and as such do not generally require annotations.

In annotating an application, the developer may need to annotate several critical sections with sequential regions. To avoid adding unnecessary dependencies between critical sections protected by unrelated locks, Replicant allows the developer to define an arbitrary number of sequential region *domains*, also referred to simply as domains. Replicant enforces the order in which threads cross sequential regions that belong to the same domain, but does not enforce any order on sequential regions in different domains. As a result, there is a one-to-one mapping between locks in an application and sequential region domains, and each critical section that is protected by a certain lock maps to a sequential region in the corresponding domain.

Definition 4.2.2. *A domain allows a partial order to be placed on sequential regions. Ordering guarantees are made on sequential regions of the same domain, but no ordering is enforced on sequential regions of differing domains.*

Consider a producer/consumer application where a producer enqueues items on a list which are dequeued by several consumers. Both the enqueue and dequeue operation require a sequential region in order to support redundant execution. Further, these two sequential regions will also belong to the same domain, because they access the same shared memory addresses (i.e. the work queue). If the consumer threads also happen to

share another global structure that is protected by a different lock, then access to that structure could be safely placed in a separate domain to avoid the performance penalty of serialized access to a single domain. Well written, optimized applications will generally have a very fine grained locking scheme, rather than one big lock. This is beneficial to our system, because unrelated sequential regions will not affect one another, allowing for increased parallelism.

4.3 Guarantees

Sequential region implementations may depend on the target operating system or on scalability requirements. However, regardless of implementation, the following guarantees must hold in order to remove non-determinism:

1. The order in which threads enter a sequential region of a particular domain is reproduced deterministically for all replicas.

2. Sequential regions are mutually exclusive within a replica, but are not necessarily mutually exclusive across multiple replicas.

Guarantee (1) is intuitive and forms the basis of our approach. As threads in one replica enter a sequential region the order is observed and recorded for the specific domain. Threads in a following replica must be made to access the sequential region in the same order. The definition of which thread sets the ordering and which threads are made to follow is not fundamental to the concept. An implementation may have this relationship established at startup or it may dynamically reevaluate the relationship during operation. Our implementation, which will be described in detail in Chapter 5, allows for flexibility in thread relationships.

Guarantee (2) is less obvious. Once a thread is allowed to enter a sequential region, another thread may not enter until the first has exited. This is important, because the

kernel cannot enforce relative thread execution rates and allowing more than one thread to execute a sequential region at a time prevents the kernel from making guarantees on the order in which threads execute each instruction in the bounded region. Since the order of shared memory accesses must be controlled, and because relative thread execution rates in a sequential region cannot be enforced, sequential regions are made mutually exclusive. Although the mutually exclusive property of sequential regions may seem to pose an overhead cost, they are most often aligned with critical sections that are already mutually exclusive. As a result, mutual exclusivity adds little or no overhead in the common case.

4.4 Discussion

While one can infer most of the inter-thread communication in an application from its use of locks, developers frequently find application-specific opportunities to increase performance by avoiding the use locks when accessing shared variables. As a result, when porting applications, we have found that while using information gleamed from the locks to automatically add sequential regions saves a great deal of time, some amount of manual analysis is usually required to discover the communication that does not occur in a critical section, but can still affect external output values.

Consider the example in Figure 4.2. This is a slightly simplified code section copied from Apache 2.2.3. In this example, `queue_info` is a global variable shared by a producer thread, which receives connections from the network, and n consumer threads, which handle each connection. This function is called by the producer thread to wait for an idle consumer thread to handle the next work item. Upon completing an item of work, a consumer will increment the `queue_info->idlers` count. When the producer finds an idle consumer, the same counter will be decremented, as shown on `line 39` in the example. However, incrementing and decrementing this counter is not performed inside

a lock and atomic operations are used instead. Further, because the producer is the only thread that decrements the counter, it is able to safely perform the boolean expression on `line 12`. Here the producer checks if the number of idle consumers is zero. If so, the `queue_info` structure is locked and the value of the `idlers` property is checked again on `line 18` before the producer waits on a conditional variable on `line 19`. The recheck prevents a race condition where a consumer may have become idle between the time that the check was made, at `line 12`, and the lock was acquired, at `line 13`. By not locking the structure initially, the producer thread is able avoid locking the `queue_info` structure in the common case (i.e. there will almost always be at least one idle consumer) and gain an increase in performance.

While Apache is able to gain a performance improvement by avoiding a lock acquisition, the resulting implementation may introduce non-determinism that cannot be replicated by a kernel level redundant execution system. In this example a sequential region can be added to the function to ensure that a deterministic execution is enforced and all replicas will arrive at a deterministic result for the boolean expression at `line 12` (note that all accesses to this shared variable must be annotated). However, what is more difficult is determining if in fact this particular section of code must be made deterministic for successful redundant execution. The difficulty is in determining if a non-deterministic path will cause a divergence that the redundant execution system cannot reconcile. Perhaps surprisingly, the global variable `queue_info` does not require annotation for Apache to correctly operate on Replicant. This is because the calling thread will not cause divergent outputs by waiting.

Narayanasamy et al. term inter-thread communication that occurs outside of a critical section as a *benign data race* [41]. Further, they have been able to automatically identify these race conditions from uninstrumented applications. Manual analysis is also used to classify these races into five categories which correspond to the situations in which a programmer was able to increase performance by avoiding locks. The categories outlined

in their work map precisely to the cases that we have observed when running applications on Replicant. The situation described above, and depicted in Figure 4.2, is categorized as a *Double Check* in the Narayanasamy et al. classification. Although a check is made on a global variable without synchronization, the value is checked again after a lock is acquired. Other categories include, *Both Values are Valid*, *Redundant Writes* and *Disjoint Bit Manipulation*.

Lastly, correct redundant execution is guaranteed if *all* accesses to shared memory are annotated with sequential regions. Because it is difficult to determine if a particular shared variable requires annotation, finding and annotating all shared memory accesses is the preferred method for porting large multi-threaded applications to execute on Replicant. Annotations can be removed if they are discovered to be unnecessary. Automatically inferring the placement of sequential regions will be discussed in depth in Section 5.4.

```
1  struct queue_info_t {
2      apr_uint32_t idlers;
3      apr_thread_mutex_t *idlers_mutex;
4      apr_thread_cond_t *idlers_cond;
5  };
6
7  int queue_info_wait_for_idler(queue_info_t *queue_info)
8  {
9      int rv;
10
11     /* Block if the count of idle workers is zero */
12     if (queue_info->idlers == 0) {
13         rv = pthread_mutex_lock(queue_info->idlers_mutex);
14         if (rv != 0) {
15             return rv;
16         }
17
18         if (queue_info->idlers == 0) {
19             rv = pthread_cond_wait(queue_info->idlers_cond,
20                                    queue_info->idlers_mutex);
21             if (rv != 0) {
22                 int rv2;
23                 rv2 = pthread_mutex_unlock(
24                                 queue_info->idlers_mutex);
25                 if (rv2 != 0) {
26                     return rv2;
27                 }
28                 return rv;
29             }
30         }
31
32         rv = pthread_mutex_unlock(queue_info->idlers_mutex);
33         if (rv != 0) {
34             return rv;
35         }
36     }
37
38     /* Atomically decrement the idle worker count */
39     atomic_dec(&(queue_info->idlers));
40
41     return rv;
42 }
```

Figure 4.2: Non-locking Shared Memory Access. A thread that accesses shared memory without locking. Simplified snippet modified from Apache 2.2.3 in `fdqueue.c`.

Chapter 5

Prototype Implementation

Our determinism infrastructure is built as a component to a complete redundant execution system called Replicant. This chapter will describe the details of the in kernel implementation that was required to enforce deterministic redundant execution of shared memory workloads, and will outline our support for automatically annotating user space applications. The system calls that have been added to the Linux kernel to support shared memory workloads are briefly outlined in Table 5.1 and an annotated sample application is depicted in Appendix A.

5.1 Domains

Replicant guarantees that replicas enter and exit sequential regions in the same order. However, an application typically only requires a partial order to be enforced among its sequential regions. As illustrated in Chapter 4, sequential regions heuristically map onto critical sections protected by locks. Just as threads can execute critical sections protected by different locks concurrently, no ordering is required for sequential regions that map to different locks. However, an order for sequential regions mapping to the same lock must be enforced. To define partial orders for sequential regions, Replicant allows the developer to create an arbitrary number of sequential region *domains*. Replicant enforces the order

System Call	Description
`init_dom(`*`label`*`)`	Initializes a domain identified by *`label`*.
`destroy_dom(`*`label`*`)`	Deallocates domain identified by *`label`*.
`alias_dom(`*`label, alias`*`)`	Associates an alternate domain label *`alias`* to *`label`*.
`begin_seq(`*`label`*`)`	Enters sequential region in domain *`label`*.
`end_seq(`*`label`*`)`	Exits sequential region in domain *`label`*.

Table 5.1: System Call Annotations. Annotations added to an application, either by directly modifying the source code, or by using an automated method, will translate into the system calls described here.

Structure	Name	Description
Domain	`domain_t`	Domain specific state (linked list node).
Sub Damain	`domain_sub_t`	Replica specific per-domain state.
Task Order	`domain_tsk_t`	Per-domain thread ordering (linked list node).

Table 5.2: Data Structures Summary. Brief summary of the data structure described in Figure 5.4.

that threads pass through sequential regions that belong to the same domain, but does not enforce any order between sequential regions associated with different domains. As described in the previous chapter, there is a one-to-one mapping between locks in an application and sequential region domains in the kernel and each critical section that is protected by a certain lock maps to a sequential region in the corresponding domain.

Figure 5.1 depicts the organization of the kernel level data structures added in support of our sequential regions implementation, where domains are represented as large white boxes. The domain structure holds data specific to each domain, such as the thread ordering that is to be enforced (shown in the figure as the `tasks` list). Each domain will also have one *sub-domain* for each replica. Sub-domains are shown as small hashed boxes in Figure 5.1 and are responsible for maintaining per-replica data. In a 2-replica system, there are always 2 sub-domains for each domain. Replica specific data includes

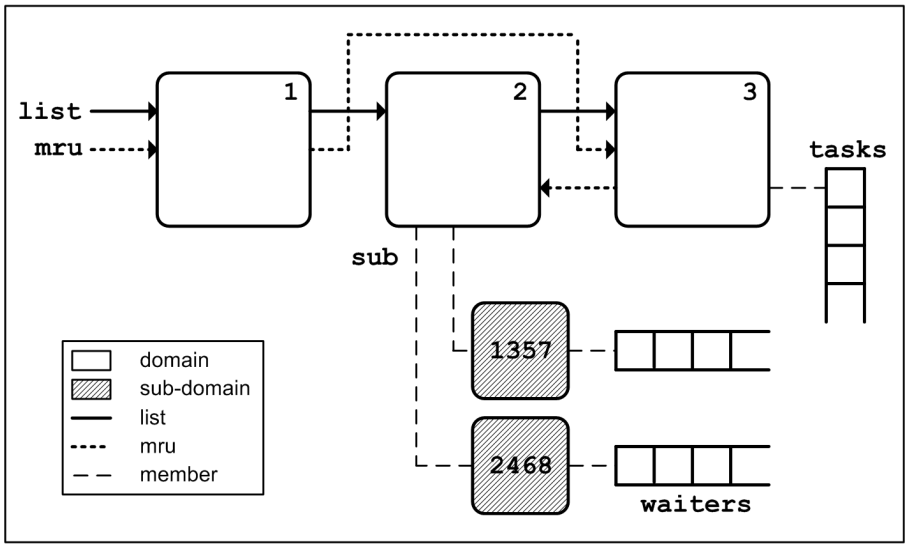

Figure 5.1: Sequential Region Domain Design. The structures that implement our sequential region domains infrastructure.

a queue for threads waiting to access the domain, a *busy* flag that also indicates the PID of the task currently executing in a sequential region belonging to the domain and a *nesting* count to support recursive locks. Most importantly, each sub-domain maintains a per-replica *label* that is used by the application annotations to identify a specific domain. Figure 5.1 shows the sub-domains corresponding to domain #2 and their label values in the middle of the hashed box. Corresponding sub-domains do not require equal labels values. However, domain labels must not be duplicated within each replica in order to have uniquely identifiable domains. A summary of the data structures added to the Linux kernel in support of our sequential regions implementation is shown in Table 5.2. Each structure is also written out explicitly in Figure 5.4.

To create a domain, the application executes the `init_dom` system call, and passes a domain identifier of type `long` as the only argument, which will be used as the domain label. Resources allocated in the kernel to track a specific domain are released by a corresponding `destroy_dom` system call, which also takes the domain label as its only argument. When a domain is created a `domain_t` structure is allocated in the kernel and added to a linked list in the Process Control Block (PCB), which is shared across

all replicas. In Linux, the PCB is implemented as the `task_struct` structure and linked lists are often implemented with the `list_head` structure and associated macros [5]. The list of domains must be shared between any threads that share writable memory pages. This includes threads that share their entire address space, as well as processes that share memory pages through the System V API call to `shmget`.

To promote the use of our system with legacy applications, one of the design goals described in Section 4.1 was to allow a convenient mapping of lock structures in user space to our kernel implementation. In order to promote executing unmodified applications, which will be discussed in Section 5.4, our system is able to use the address of a lock data structure in user space, typically the `pthread_mutex_t` data structure, as the label for a domain. Since Replicant is targeted toward security applications, the base heap address will normally differ across replicas, using ASLR. Since locks are usually designed as a member of a structure allocated in heap space, the address of a particular lock in user space will often differ between replicas. To easily map locks present in user space to domains in the kernel we must support domains that are referenced with different labels, depending on the replica. This means that domains initialized by each replica cannot be associated based on the label identifier. Instead, our design relies on initialization *order* to match up domains created by each replica. Unfortunately, this means that the order of lock initialization must also be enforced between threads in an application. To fulfil this requirement, a domain of *label 0* (zero) has special significance. This label cannot be initialized by the application and is instead preinitialized by the kernel for each process. It solves the chicken-and-egg problem where domains created with randomly different labels across replicas must themselves be ordered deterministically. In this case, domain initializations are placed inside a sequential region belonging to the domain label 0. The same result can be achieved if the application first creates a domain during a non-threaded section of execution and uses this domain for further domain creation in parallel sections. The domain label 0 is simply added for porting convenience.

In order to support applications with a very large number of domains, our system maintains a most recently used (MRU) list in parallel with the default list that remains allocation ordered, as shown in Figure 5.1. Linked list semantics require linear search, so the `mru` list is used to exploit temporal locality by moving any accessed domain to the front of the list. The performance benefit of the `mru` list relies on the assumption that recently accessed domains are likely to be referenced again in the near future. Most of the applications we experimented with had little overhead in searching a small list of domains. However, as will be discussed further in Chapter 6, MySQL allocates over 40,000 domains, most of which remain unused, on startup. This can lead to excessively long search times and thus motivated the use of an MRU list. In actuality, both `list` and `mru` data structures, depicted in Figure 5.1, form two circular, doubly-linked lists. The motivation for a linked list implementation was primarily based on development efficiency and to promote experimentation. There is no fundamental reason for using a list data structure. Section 9.1 will discuss improvements to this design, including removing the linear search aspect, and Section 9.2 will describe how these data structures can best be generalized to n-replica.

5.2 Sequential Regions

Sequential regions cover a section of code, and so one system call is required to inform Replicant when an application enters the region and another is used to inform Replicant when it exits the region. This is accomplished by the `begin_seq` and `end_seq` system calls that we have added to the Linux kernel. The developer also passes the domain identifier, or label, discussed in the previous section, to each `begin_seq` and `end_seq` system call. A sequence of `begin_seq` calls will define the order in which threads cross the sequential regions in a domain and Replicant will enforce that threads in all replicas cross these sequential regions in the same order.

As described in the previous section, the kernel maintains a list of domains, and each domain itself contains it own list that records the order in which threads pass through the sequential regions in the domain. To make the order in which threads traverse the sequential regions well defined, Replicant ensures that multiple sequential regions in the same domain cannot execute concurrently. If a thread in a replica calls `begin_seq` while another thread in the same replica is in a sequential region from the same domain, it is stalled until the first thread calls `end_seq`. While this may appear to reduce opportunities for concurrency, it does not in practice because sequential regions are usually aligned with critical sections that are already mutually exclusive. `begin_seq` calls on unrelated domains are unaffected and may proceed.

We demonstrate how sequential regions affect thread executions with an example illustrated in Figure 5.2. In our example, there are three threads executing code across four sequential regions, all within the same domain. As shown in the figure, the first replica to have a thread call `begin_seq` in a domain is designated the *master replica* of the domain, and the other replica is designated the *follower replica* for that domain. Each domain may have different master and follower replicas. The master replica defines the order that sequential regions must be traversed, as illustrated by the solid arrows between the grey boxes and numbers in the grey boxes. The follower replica must also pass through the sequential regions in the order set by the master. Each grey box is preceded by a `begin_seq`, which marks the beginning of a sequential region in the application code. The `end_seq` system calls are not shown in the figure, but are executed right after each grey box to mark the end of each sequential region.

When a thread in the master replica enters a sequential region, a lookup is made to find the domain's order list, and an identifier is enqueued onto the list. When a thread in the follower enters a sequential region, Replicant checks the domain's order list to see if the next thread to enter the sequential region in the master was its peer thread. If it was, then the thread proceeds into the sequential region. Otherwise, the thread must block

until all threads before it in the order list have passed through their sequential regions. In the example, T1 executes its second sequential region before T2 does, and is forced to block until T2 passes through sequential region #2. When T2 executes `end_seq`, it finds that the next thread in the order T1, is currently blocked, so it wakes up T1. Similarly, T3 executes `begin_seq` before T1's second sequential region, and must also wait for its turn. When there are no more outstanding sequential regions, the next replica to execute a `begin_seq` becomes the master replica and will define the ordering of sequential regions in the domain.

The domain's order list is shown in Figure 5.4 as the member `tasks`. It represents a linked list of `domain_tsk_t` structures which define the order of allowed threads in the domain. When a *following* task attempts to enter a sequential region of a particular domain, it will only be permitted if it is the next task on the domain's `tasks` list. We use a separate structure, rather than another `list_head` in the `task_struct`, because any task may be next on an arbitrary number of domains. In fact, the same task may be referenced in the `tasks` list an arbitrary number of times, depending on how far ahead the master replica is executing relative to the follower replica.

Threads that attempt to enter a sequential region may be required to wait, as the example above described. The `waiters` list in each sub-domain will have a different meaning depending on whether a replica acting as the master or follower replica. A task belonging to the master replica will wait on the `waiters` queue if another task in the same replica is currently executing in a sequential region belonging to the same domain. If this is the case, the `busy` flag will be set in the `domain_sub_t` structure. When the task that is currently holding the domain exits its sequential region, any waiting task may be dequeued from the `waiters` queue. A task belonging to the following replica will also be enqueued on the `waiters` queue of its sub-domain if the current domain is flagged as busy. However, a following task may also wait when an attempt is made to enter a sequential region out of turn. As a result, dequeuing a task from the followers

`waiters` queue is more complex then that of the master. First, tasks are either dequeued in the correct order, as defined by the `tasks` list in the domain, or not at all. Dequeuing a task out of order will only cause it to wait again and introduces unnecessary context switching. Secondly, the followers `waiters` queue must be checked for waiting tasks by the master replica on exit of a sequential region. This is because a following task may be enqueued as a waiter without having another task in the same replica occupy the domain.

Figure 5.1 depicts both the `tasks` list and the `waiters` lists and their relationship to sequential region domains in the kernel. The `tasks` list is associated with the `domain_t` because one order list is needed for each domain. There is one `waiters` lists for each sub-domain, defined by the `domain_sub_t` structure, because replicas operate on an individual domain independently. That is, if a thread has entered a sequential region of a specific domain in one replica and marked it as busy, a thread in another replica may still enter a sequential region of the same domain provided that another thread in the same replica has not marked the domain busy.

Because threads in the follower replica must pass through all the sequential regions in the same order as threads in the master replica, the developer must ensure that applications do not invoke any sequential regions that occur in one replica but not the other. If this does happen, then the follower replica will not be able to enter any sequential regions after the missing region and will not be able to make forward progress. The developer can prevent spurious sequential regions by ensuring that any variables that can determine whether a sequential region should be executed are themselves protected by sequential regions.

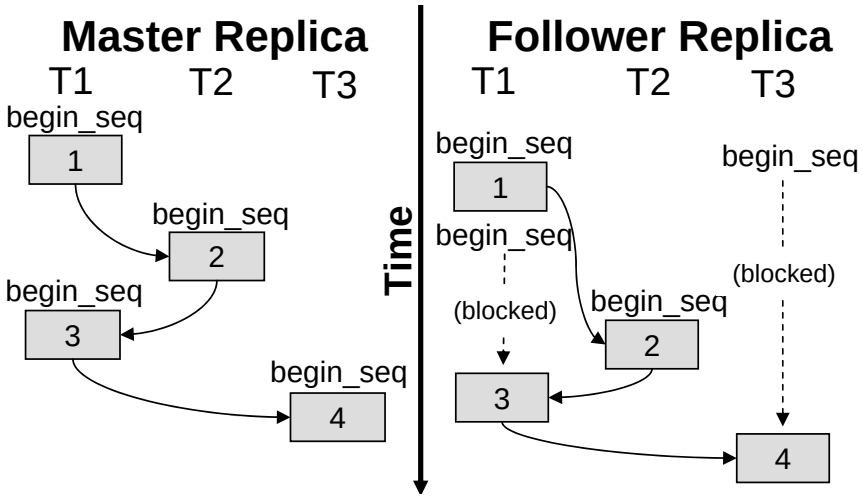

Figure 5.2: Enforcing Thread Ordering. Replicant ensures that peer threads pass through sequential regions (shown as grey boxes) in the same order regardless of what order they call `begin_seq` in the follower replica.

5.3 GNU/Linux POSIX Support

Sequential regions often form a one-to-one mapping to lock and unlock API calls in user space. However, *conditional variables* in the POSIX standard, which are synchronization calls, can have implicit lock/unlock semantics. As a result conditional variables require special consideration, specifically due to their current implementation on a GNU/Linux system. This section will describe the the mechanisms that we have established to support conditional variables without modifying the GNU C Library, `glibc`. Modifying `glibc` directly is complex and will be discussed further toward the end of this section.

Conditional variables allow threads to wait for events without entering a busy wait. When a thread reaches a point in execution where it cannot continue until a particular condition is true, it can wait on a conditional variable. A conditional variable can only be entered while holding a lock. On entry that lock is released, and the thread is removed from the operating system's runnable tasks list and placed on a queue of waiting tasks. If another thread changes the condition to the true state, it may also wake up one waiting thread, called a *signal*, or wake up all waiting threads, called a *broadcast*. The prototypes

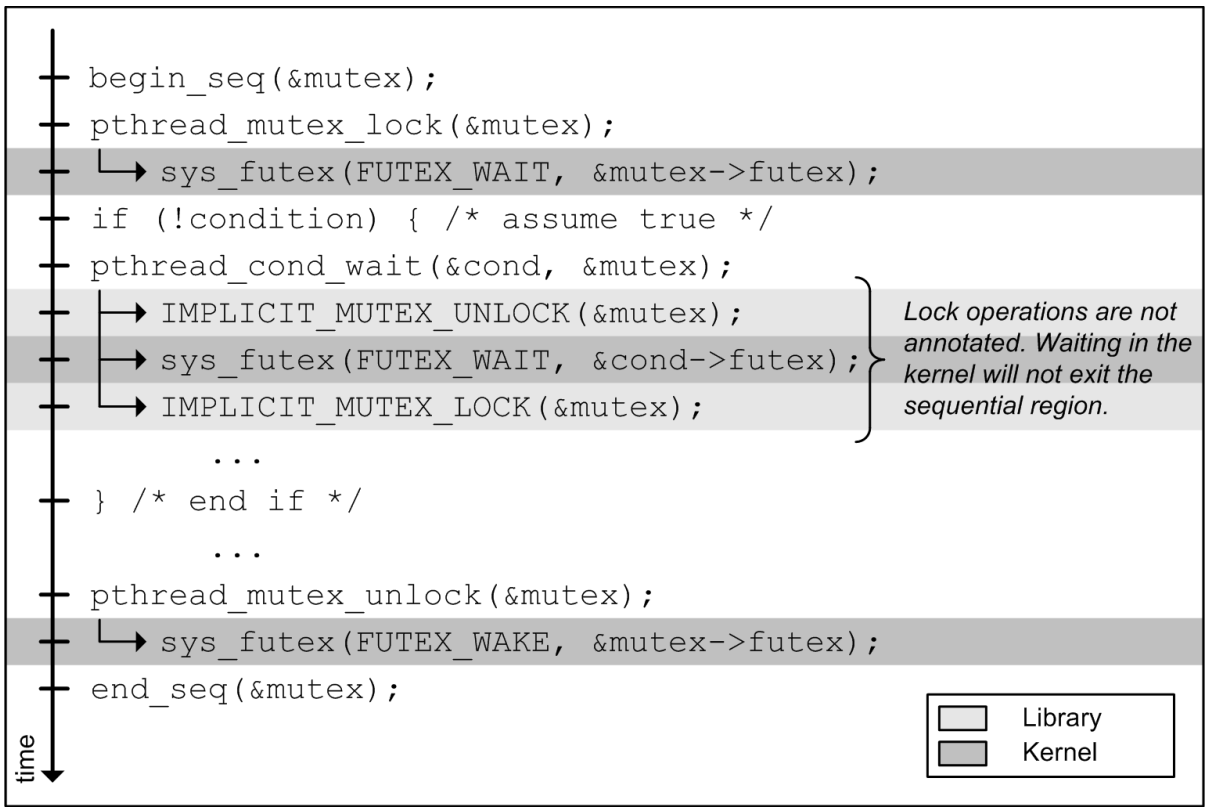

Figure 5.3: Conditional Variable Execution. Conditional variable waiting causes implicit calls to underlying mutex lock and unlock operations, which are unannotated.

used for conditional variables in a Pthreads application are shown in Figure 2.1, prefixed by "`pthread_cond`". An example of a thread waiting on a conditional variable is depicted in Appendix A.

Based on the description of sequential regions thus far, a `begin_seq` call is placed before lock operations and an `end_seq` call is placed after unlock operations. Locking synchronization primitives are used to direct the placement of these annotations so that any access to shared memory occurs inside a sequential region. Figure 5.3 illustrates the sequence of calls made in a typical conditional variable wait operation and also highlights the type of code being executed. Dark grey lines are code executed in the kernel, while light grey lines are executed in a library between the application and the kernel. Any code that is not highlighted is implemented directly in the application. When the `sys_futex`

call is made with a parameter of `FUTEX_WAIT`, the kernel will block the calling thread until a corresponding call is made to wake up the same futex value (passed as the second argument in this example). Since the mutex unlock remains unannotated in the library call, the kernel will wait in `futex_wait` without exiting the sequential region that was acquired before locking the mutex. Any threads that require access to the same area of shared memory will then be blocked waiting for the sequential region to become free, and as a result, the thread waiting in `sys_futex` may never be woken up. Note that the `IMPLICIT_MUTEX_LOCK` and `IMPLICIT_MUTEX_UNLOCK` macros may also call `sys_futex` as well. However, these implicit operations may not trap to the kernel and so cannot be relied on for interposition by the kernel.

An implicit lock release in a library between the application and the kernel may cause a deadlock. We can solve this problem by implicitly exiting the sequential region in the kernel when the `sys_futex` call is made. The sequential region can then be implicitly entered again before the application returns to user space. The issue is further complicated, however, because the kernel does not know which sequential region to exit. A thread may hold an arbitrary number of locks, and the POSIX standard does not specify that conditional waits can only unlock the most recently acquired lock. Any held lock in user space will also map to a busy sequential region domain and the kernel does not have enough semantic information to correctly identify the domain to be released.

To decide on which domain to release, we could heuristically exit the most recently entered sequential region when a call is made to the kernel to wait. However, this solution is not general. Instead, we bridge the gap with the use of an additional system call that allows the developer to associate the conditional variable with the mutex in the form of an *alias*. The user space application calls the `alias_dom` system call with the address of the `pthread_mutex_t` structure as the label and the address of the `futex` member of the `pthread_cond_t` structure as the alias. This system call will associate an alias label to the specified domain which can then be used interchangeably with the real label. Ideally,

an arbitrary number of aliases could be applied to any label. However, we have not had the need for more than one, and so our system limits the number of aliases that a domain may have to one at any given time. If another `alias_dom` call is made with the same label, the first alias is overwritten. With this infrastructure the kernel is then able to determine if the `futex` address observed is the alias of a specific sequential region domain and the appropriate sequential region can be exited. Whenever the kernel is asked to wait on a `futex` address, it checks if the calling process is currently executing in a sequential region with domain label, or alias, equal to the value of the `futex` address. If this is case, the sequential region is implicitly exited before waiting and re-entered before the thread exits the system call. This means that in Figure 5.3 when `sys_futex` is called to wait on `&cond->futex`, the kernel will recognize the address of the futex as being an alias for the domain with label `&mutex` and will implicitly call `end_seq(&mutex)`.

It is important to note that domain aliasing is not a fundamental requirement for a sequential regions implementation, but rather an artifact of adding annotations around the POSIX Threads API. If the `glibc` source code is modified directly, below the Pthreads API, domain aliases are not required. Although this is not a fundamental component of our approach, it is necessary if the application source code uses the `glibc` libraries on a Linux 2.6 kernel and annotations are made around the Pthreads API. We discuss our experience in directly annotating `glibc` in Section 9.3.1.

5.4 Automatic Annotations

To ease the porting of existing multi-threaded applications to Replicant, we have created a library that interposes between the application and calls that are made to Pthread functions. The goal of this library is to promote as few modifications as possible to the original application.

The interposition library is compiled as a shared object and dynamically loaded at

run time via the `LD_PRELOAD` environment variable. The dynamic linker in UNIX-like operating systems, will automatically load the libraries specified by `LD_PRELOAD`. This allows our library to replace symbols to Pthread functions with our own implementation. Our functions are used to make system calls to initialize sequential region domains, assign aliases, enter sequential regions, and so on. The Pthread calls are then forwarded on to the original `glibc` implementation using `dlsym` and `dlvsym`, which allow us to find the address of the real Pthread functions compiled into `libpthread`.

When our interposition library is loaded, each call to `pthread_mutex_init`, made by the application, will result in a corresponding `init_dom` where the label is set to the address of the `pthread_mutex_t` structure. This effectively creates a domain for every lock variable. Lock initializations are ordered by placing them in sequential regions of the default domain, label 0, that is initialized at application startup by the kernel. Each time the application locks or unlocks the mutex, the interposition library makes a `begin_seq` or `end_seq` call respectively, again passing the address of the mutex variable as the domain label. In the case of a read-write lock, where more than one thread can hold a read lock at a time, the interposition library only places a sequential region around the lock acquisition to permit concurrency. As described in the previous section, conditional variables require special attention. Although only required once, a call to `alias_dom` is placed before each invocation of `pthread_cond_wait`, however, no annotation is required for signal and broadcast operations. Many of the Pthread calls outlined in Figure 2.1 are intercepted annotated and forwarded on to the underlying `glibc` implementation.

As discussed in Section 4.4, some annotations cannot be inferred through the use of locks and the method described here cannot be used to automatically annotate shared memory access outside of a critical section. Automatically annotating accesses to shared memory made outside of a critical section is left as interesting future work, and discussed further in Section 9.3.2.

```
1  /* These are the structs that implement sequential region
2     domains.  Linked lists are the primary data structure. */
3
4  typedef struct seq_domain_sub {
5      wait_queue_head_t waiters;  /* tasks waiting to acquire */
6      long label;                 /* label (e.g. address of lock) */
7      long alias;                 /* alternate, equivalent label */
8      int busy;                   /* task pid in the domain */
9      int destroyed;              /* indicates no longer used */
10     int nesting;                /* nesting oh the seq region */
11     void *private;              /* pointer to my domain struct */
12 } domain_sub_t;
13
14 typedef struct seq_domain_tsk {
15     struct list_head list;      /* order of allowed tasks */
16     struct task_struct *task;   /* corresponding task */
17 } domain_tsk_t;
18
19 typedef struct seq_domain {
20     struct list_head list;      /* domains by init'ing */
21     struct list_head mru;       /* domains by most recent used */
22     struct list_head tasks;     /* task order on this domain */
23     unsigned long count;        /* order of init'ing */
24     domain_sub_t sub[2];        /* per replica data */
25     int master;                 /* replica identifier */
26 } domain_t;
27
28 /* The Linux PCB, struct task_struct, is modified to add several
29    members that implement the sequencial region domains. */
30
31 typedef struct task_struct {
32     /* ... */
33
34     spinlock_t *dom_lock;       /* big lock for the whole thing */
35     atomic_t *dom_count;        /* reference count */
36
37     struct list_head *dom_list; /* domains by init'ing */
38     struct list_head *dom_mru;  /* domains by most recent used */
39     struct list_head *dom_next; /* next domain to be init'd */
40
41     domain_t *dom_cache;        /* avoid searching */
42 }
```

Figure 5.4: Sequential Region Data Structures. The data structures that have been defined in our modified Linux kernel in support of sequential regions.

Chapter 6

Applications

In order to evaluate the feasibility of Replicant with realistic multi-threaded workloads, we have chosen four threaded applications that benefit from multi-core hardware and use shared memory for inter-thread communication. In this chapter we describe each of these applications and our efforts to port them to Replicant.

6.1 Overview

As described in Section 3.3, Replicant is a unique redundant execution system that allows replicas to diverge in their execution and only commits outputs that have been agreed upon by the majority. Unfortunately, applications that communicate through shared memory often have divergent outputs that are not the cause of a security compromise or fault. As a result, an application that wishes to be replicated that uses shared memory communication may require annotations that reveal inter-thread communication to Replicant. Making *all* inter-thread communication explicit to Replicant, using the sequential region infrastructure described in the previous sections, will yield replicas that will always have deterministically identical executions. However, since sequential regions reduce opportunities for concurrency, the developer may also try to increase performance by only annotating communication where necessary – when it can directly affect sys-

tem calls with externally visible output. The difficulty of this task is dependent the complexity of the inter-thread communication and the size of the application.

In general, application threads communicate with each other via two methods. They may communicate by reading and writing to variables in their shared address space or they may use OS primitives such as signals. In an application where accesses to shared memory are always protected by locks, creating a sequential region domain for each lock variable, preceding every lock with a `begin_seq` and following every unlock with a `end_seq`, will make all shared memory communication deterministic. In general, signal events need to be ordered deterministically relative to other inter-thread communication events as well.

While signals are a common interprocess communication construct in process-based server applications, applications that have been optimized to use shared memory rarely communicate through signals. Signals that are sent from the external environment to a Replicant peer group are replicated to all members in the group. We have observed very few internal signals, sent from one thread to another, in the applications that have been ported to date. Most of the signals that have been observed were handled, without any explicit effort, through Replicant's ability to handle out of order system calls. In one case, deterministic signal delivery was required. However, this signal was received from a `sigtimedwait` system call and could easily be made deterministic with the use of sequential regions. Although sequential regions were successfully used to synchronize signal delivery in this case, we have not studied the general applicability of this method. Sequential regions make shared memory communication deterministic for use in redundant execution and further uses, such as deterministic signal delivery, are left as potential future work.

To allow for accelerated porting of applications, we use the automatic instrumentation that is described in Section 5.4. However, in all large applications that we have ported to date, the developers take advantage of opportunities to increase performance

Application	Lines of Code	Variables	Calls	Extra	Manual	Effort
Apache	224,195	34	144	2	2	days
Squid	111,061	1	12	3	14	days
MySQL	1,126,283	131	1,436	11	1,447	weeks

Table 6.1: Lock and Sequential Region Statistics. We give the number of lines of code in the application [58], the number of lock variables that are declared as well as the number of lock and unlock operations that appear in the source code. Finally, we indicate the number of "extra" sequential regions we had to add after the interception library had been applied, how many sequential regions a fully manual annotation required, and an approximation of how much time was needed to port the application.

by accessing shared variables without first acquiring a lock. In every case, these accesses are safe because of the semantics of the application, but they introduce non-determinism. Thus, while our interposition library provided much of the annotation automatically, a reasonable amount of effort was required to find and annotate shared variable accesses that were not protected by locks. We describe our experiences porting several applications below. In each case, we started by using the interception library to automatically annotate every lock in the application and then manually annotated any shared variable accesses that were not protected by locks. We then attempted to minimize the number of sequential regions by manually studying the application and placing annotation only where they were needed.

6.2 Apache HTTP Server

We have ported the Apache HTTP Server [62], which can be configured to use worker threads to handle client requests, rather than the default process forking model. In Apache, the listening socket that clients connect to is bound by the main Apache process, that runs as root. This process must run as root, because it may be required to bind to a port number between 0-1023. The bound socket is inherited by each forked child.

Children of the main process run as a non-root user, because they interact directly with input from the network. In the worker configuration, each forked Apache process contains a single listener thread that is responsible for calling `accept` on the listening socket. Each forked process also contains a configurable number of worker threads. When the listener thread returns from the `accept` system call, it creates a work item linked to the new connection. This work item is placed as an element on a linked list and dequeued by one of the waiting worker threads. The worker thread then interacts directly with the client through the socket.

When we applied the interposition library to Apache, we found a shared variable that was accessed without a lock. While non-deterministic accesses to this shared variable has no effect on the output of Apache, the operation caused Apache to non-deterministically acquire a lock. Since the interception library annotates all locked regions with a sequential region, this caused a spurious `begin_seq` to occur in one replica and not the other. As mentioned in Section 5.2, spurious sequential regions will eventually prevent the following replica from making forward progress.

In order to prevent the spurious sequential region that was caused by an unsynchronized access to shared memory, we could have simply annotated the operation with a sequential region. However, a more optimal solution was to remove the sequential region annotations from the variable that was being locked spuriously. The only inter-thread communication in Apache that can affect external outputs consists of enqueue and dequeue operations on the shared work queue described above. As a result, we could leave all other locks unannotated and only annotate two sequential regions in Apache – one when the listener thread enqueues requests, and one where each worker thread dequeues requests. This scenario is described in detail in Section 4.4.

6.3 Squid Web Proxy Cache

The Squid Web Proxy implements a high performance web caching proxy [59]. It has an event driven design and can be configured with an I/O thread pool, which prevents Squid from blocking on disk accesses. Squid's main event loop handles most of the server's processing. It accepts connections, proxies the request and handles each connected socket. The I/O threads are only used for creating, fetching and removing files from the on-disk cache.

Squid was straightforward to port with the aid of the interception library. We found two variables that were accessed without locks and would result in non-deterministic divergences in the application's output. Once access to these variables were placed in their own sequential region, Squid produced correctly confirmed outputs.

6.4 MySQL Database Server

Threads in MySQL database server (InnoDB storage engine) [40] concurrently access many shared data structures such as database tables, work queues, in-memory caches and various logging mechanisms. As a result, the communication patterns in MySQL are much more complex than in the other applications. When we applied the interception library, we found that there were many instances where MySQL accesses shared variables without locks. We instrumented these shared variables with sequential regions to make accesses to them deterministic under Replicant.

As an example, the InnoDB storage engine has a looping server thread that checks several fields in a global log data structure periodically. If the number of unflushed writes to the log exceeds a preset threshold, the server thread flushes the in-memory log file to disk – an operation that will acquire a lock on the log data structure. On every transaction, worker threads update the log data structure fields after each write to the log. Since it does not matter whether a worker's log updates are flushed in this period

or the next, the server thread can safely read the shared variables in the log structure without locking. However, if these reads are not done deterministically under Replicant, one replica's server thread might decide to flush the log to disk while its peer might not, resulting in a spurious acquisition of the log data structure's lock, and hence a spurious sequential region.

We found other events in MySQL interesting. MySQL includes a salt (a random string) in its *server hello* message so that the client can use it to hash its password. The string is derived using randomness from sources such as heap addresses. Because Replicant randomizes the addresses in each replica and because memory allocation is non-deterministic, the random string is different across replicas. To make this string identical in all replicas, we modified MySQL to obtain its randomness from the `/dev/urandom` facility provided by the OS kernel instead. Since this is a read from an external input, the Replicant matcher automatically replicates the random value to all replicas. In summary, applications running under Replicant must derive their randomness through the kernel so that it can be reproduced across replicas. Generally, applications should not rely on address layout for seeding random numbers and this practice adds no additional security.

Acquiring a good understanding of the communication that occurs between threads in a program is crucial to being able to come up with an optimal set of sequential regions for an application. MySQL's large code base and the many data structures it uses to synchronize and protect access to the shared database made MySQL more difficult to port than our other applications. The MySQL source code contains over 100 lock variables and over 1400 lock operations on those variables. At run time, we noted that MySQL initializes more than 40,000 locks during startup alone. We see our success with MySQL as an indication that once the hurdle of understanding an application's inter-thread communication has been overcome, adding sequential regions to the application can allow deterministic shared memory redundant execution.

6.5 Firefox

We have also made some preliminary progress in porting the Firefox web browser [63], which has also turned up an interesting scenario. Firefox allocates packet buffers that it sends to the X server, but does not necessarily initialize all fields in the packet, depending on type of packet being sent. Because memory allocation is non-deterministically ordered, the stale values in uninitialized fields in an allocated data structure may differ across replicas, resulting in a spurious divergence when this structure is written to the network. The error described here is actually in The GIMP Toolkit (GTK) library used by Firefox to communicated to the X server. The GTK library understands the packet format used in communication and may leave some fields blank when it is known that they have no effect.

Aside from manually finding and correcting all cases where this error occurs in the source code, another option is to replace all calls to `malloc` with calls to `calloc` using our interposition library, so that newly allocated heap space is automatically filled with zeros. If structures are allocated on the stack we would need to use a compiler option to automatically zero-out any buffers allocated on the stack (unfortunately, this option does not always exist).

Sending out uninitialized data is dangerous as it may leak confidential information to a potential attacker who exploits this side channel to attain information held by Firefox. Previous work classifies this type of behaviour as a vulnerability, and claims that it should be removed from applications [10, 11].

6.6 Discussion

Our experiences with porting applications have shown that some applications can be ported in several days with minimal effort, such as Apache and Squid. On the other hand, complex applications, such as MySQL, require an order of magnitude more effort.

To develop a heuristic for determining what applications are difficult to port, we examine statistics on the application source code to ascertain the application locking behaviour as shown in Table 6.1. As we can see, the number of locks and the size of the application are indicative of the memory communication complexity of the application.

While it is the number of accesses to variables that are not accompanied by locks that is problematic, not the number of locks themselves, the statistics are telling – the greater the number of locks, the greater the number of independent shared variables. The availability of many shared variables leads to more opportunity for optimizations that can avoid the use of locks, which means that these optimizations need to be understood and properly annotated for Replicant. Complex communication patterns also make it difficult to recognize which locks do not need annotations, making the performance of the application under Replicant difficult to optimize.

Qualitatively, we have found that inserting sequential regions is as difficult, and very similar in process, to inserting locks to parallelize an application. The developer must have a good understanding of the sharing patterns, as well as opportunity for races and concurrency. While the proper use of locks is certainly not trivial, they are in common use in concurrent applications today. Therefore, we feel that if done at the time of development, the addition of Replicant sequential regions will not be an overly heavy burden on the application developer.

Chapter 7

Evaluation

In this chapter we evaluate the performance of applications described in the previous chapter on Replicant when compared to running the same applications on an unmodified system and to a projected estimate performance. We then evaluate correctness of the outputs produced by each application. Finally, We evaluate the portability of our annotation infrastructure to another kernel level redundant execution system.

7.1 Performance

In this section we discuss our benchmarking methodology and provide a performance analysis of the ported applications.

7.1.1 Methodology

Since we are not aware of any existing redundant execution systems that can support threaded workloads on multiprocessor hardware, we develop a *projected estimate* of the overhead of a kernel level redundant execution system, against which we can measure performance of Replicant. Our projected estimate is computed by measuring the ratio between the time an application spends executing user space code, and the time the

application spends in the kernel. Any kernel level n-replica redundant execution system will have to execute the user space portion n times, and ideally only execute the kernel space portion once. Thus, to compute the projected performance for a particular application, we use the following method: in a run of an unmodified (vanilla) application on an unmodified kernel, suppose the amount of time spent in user space is u, the amount of time spent in the kernel is s, and the total execution time requires t seconds. Thus, in the case where *all processors are fully utilized by the application*, the projected execution time t' for the same application on a n-replica system can be estimated as:

$$t' = \frac{n \cdot u + s}{u + s} \times t \tag{7.1}$$

Where $n = 2$ in our 2-replica prototype of Replicant. By comparing Replicant against this estimated performance, we gain an understanding of the extra overhead Replicant adds with the additional bookkeeping associated with the harness, as well as lost concurrency due to the sequential regions.

All benchmarks were performed on an Intel Core 2 Duo 2.13GHz machine with 1GB of memory running Fedora Core 5. All applications were run over a gigabit LAN unless otherwise stated. The working set of all benchmarks fit in memory and the number of threads was increased until the vanilla benchmark could no longer utilize any more CPU time. We note that this does not mean that applications were necessarily able to utilize both CPUs to their maximum utilization. We then compare the performance of Replicant against a vanilla application with only 1 CPU enabled, both CPUs enabled and our projected estimate, derived from our dual processor runs, as described in Equation 7.1. The comparison against the vanilla application running on a single CPU is indicative of the case where the vanilla application is not able to use all the cores available to it. This is a reasonable scenario considering that future processors are projected to have over 80 cores [29].

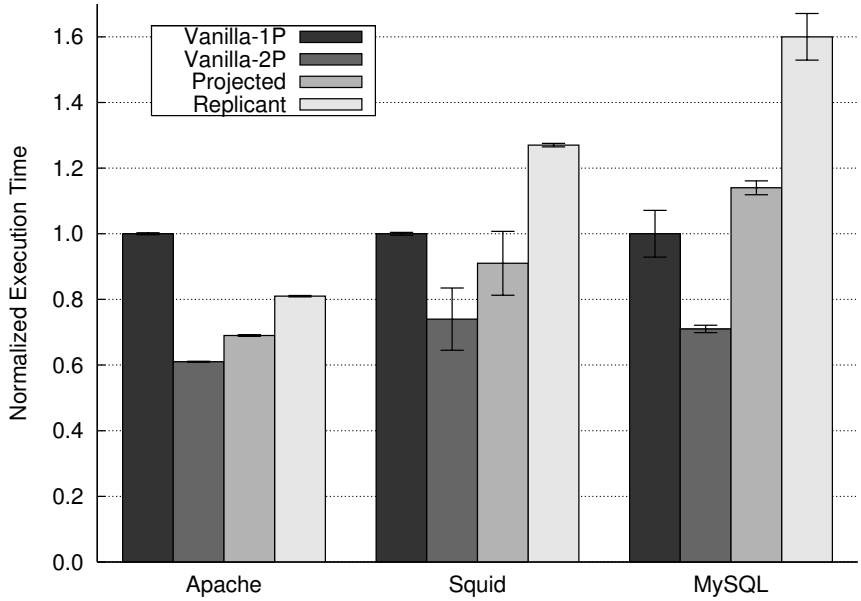

Figure 7.1: Replicant Application Overhead. Comparison of execution time of applications ported to Replicant, normalized to the single processor case.

7.1.2 Results

We will now present and analyze the performance of our application benchmarks on Replicant. Our results are summarized in Table 7.1 and compared in Figure 7.1.

We evaluated Apache 2.2.3 with the worker Multi-Process Module (MPM) enabled. Apache was configured with 2 processes and a maximum of 128 clients (64 worker threads per process), which is the default maximum. We configured Webstone [66] with its standard file distribution test and 40 clients, which allowed Apache to reach its maximum throughput. Table 7.1 gives the average over 5 runs in seconds per kilobit transferred. This result was converted from the throughput that is reported by Webstone.

Apache has very good performance when running on Replicant for two main reasons. First, the stalls incurred due to sequential region ordering are hidden by other threads in the follower replica thanks to the large amount of parallelism available in Apache. In addition, Apache spends much of its time in the kernel, which means that very little computation needs to be repeated. This is evident in the modest 33% slowdown that

Application (units)	**Vanilla**		**Projected**	**Replicant**
	1P	**2P**		
Apache (s/kb)	3.16 (±0.01)	1.93 (±0.01)	2.19 (±0.01)	2.56 (±0.01)
Squid (s/kb)	4.23 (±0.02)	3.14 (±0.40)	3.85 (±0.41)	5.37 (±0.02)
MySQL (s)	11.36 (±0.03)	8.04 (±0.13)	12.93 (±0.24)	18.18 (±0.81)

Table 7.1: Performance of Replicant on three representative multi-threaded application workloads. We also provide measurements of the unmodified application on both one processor and two processor hardware, as well as an estimate of the projected performance of Replicant. The numbers in the brackets indicate standard deviation.

Apache experiences against the two processor vanilla run.

We also evaluated the Squid 2.3.STABLE9 web proxy with the asynchronous I/O option enabled, which spawns a pool of 16 threads to handle disk operations. Squid also functions as a web cache in this configuration. Webstone was configured with 80 clients, which we found allowed Squid to reach its maximum throughput. The standard Webstone test was modified to increase the number of unique files but still maintain the same file size distribution. This was necessary so that the working set of files would not fit in Squid's in-memory file cache, forcing it to exercise the I/O thread pool for disk accesses. Note, that the new file set still fit in the kernel's buffer cache such that no I/O was actually made to disk during test runs. As in the case of Apache, we averaged the results over 5 runs and converted throughput into seconds per kilobit transferred, as shown in Table 7.1. The high variance exhibited on dual processor Squid runs was attributed to a Linux kernel helper thread, `ksoftirqd`, which inconsistently affected throughput. According to its `man` page:

> `ksoftirqd` is a per-cpu kernel thread that runs when the machine is under heavy soft-interrupt load. Soft interrupts are normally serviced on return from a hard interrupt, but it's possible for soft interrupts to be triggered more quickly than they can be serviced. If a soft interrupt is triggered for a

> second time while soft interrupts are being handled, the `ksoftirq` daemon is triggered to handle the soft interrupts in process context. If `ksoftirqd` is taking more than a tiny percentage of CPU time, this indicates the machine is under heavy soft interrupt load.

This thread was not able to acquire significant CPU cycles in any of the other Squid benchmarks.

Squid's main thread is an event-driven processing loop, which handles all connections and only uses its thread pool for I/O requests to disk. As a result, Squid has poor load balancing between its threads. In our tests the main thread became the bottleneck on the dual processor, leaving Squid unable to make good use of the second processor. Unfortunately, Replicant is also unable to take full advantage of the free CPU cycles, as the main thread in both replicas are tightly coupled to the socket input. This result leads us to believe that event driven design is not ideal for replication on multiprocessor systems. Performance could be improved by adding more parallelism to Squids main processing loop.

We evaluated MySQL 5.0.25 (InnoDB storage engine) with the SysBench v0.4.6 OLTP benchmark [61] over the loopback interface. SysBench generated a load of 100,000 read-only queries using 16 threads to the MySQL server which was running a database with 500,000 entries. The average execution time over 5 runs are reported in Table 7.1. The primary source of overhead in MySQL was the large number of sequential regions invoked. Each sequential region causes a trap into the kernel, and may also cause stalls in the follower replica as threads are forced to wait for their turn into the sequential region. We feel that MySQL's pervasive use of shared memory throughout the application presents a worst-case for our sequential region infrastructure.

7.1.3 Output Correctness

We tested each of our ported applications to confirm that the outputs produced by Replicant were correct. Tests for correctness were performed as follows. To test Apache and Squid, we used the Webstone [66] benchmark, which tests the static performance of web servers using a standard file size distribution. Webstone checks the correctness of all results returned from the web server and reports any errors it encounters. For MySQL, we used SysBench [61] to generate load on the database. We then checked the consistency of the database produced after executing queries on a Replicant run. The database was not found to be corrupt after any of these test runs. Finally, we have also directed our web browsers to use the Squid web proxy running on Replicant for everyday web browsing. Squid served both dynamic and static content through its proxy interface and used its on disk cache whenever possible. There was never any perceivable delay over that experienced when Replicant was disabled and multiple simultaneous connections, from several distinct clients, were handled without error.

7.1.4 Discussion

We find that there are three major application-dependent factors in Replicant performance on multi-threaded shared memory workloads. The first is how well the application balances load among threads. Squid has poor load balance and exhibits poor performance, while Apache, a very similar application, has good load balance and enjoys good performance. The second is the number of sequential regions that need to be invoked. MySQL has many sequential regions, which we were not able to remove. Sequential regions reduce opportunities for concurrency. Finally, the last factor is the ratio of user space to kernel space execution. Applications that spend much of their time in the kernel will experience less overhead.

7.2 General Applicability

In order to evaluate the general applicability of our shared memory determinism annotation system, we have attempted a port of sections of our code to a well known kernel level redundant execution system which does not have support for shared memory workloads, such as those that were evaluated in the previous section.

The N-Variant system [14] is a redundant execution system that can be used to detect attacks on application integrity by executing n instances of an application and varying either the instruction set encoding or using disjoint address spaces for each instance. Each of these forms of diversity allow applications running above N-Variant to be protected form specific attack vectors. (1) Disjoint address spaces among replicas protects an application from attacks that attempt to jump to an absolute address, such as return to libc attacks. (2) By varying the instruction set encoding for each replica, code injection attacks are mitigated. N-Variant was implemented by modifying the Linux 2.6.16 kernel, for which the source code has been made available [13].

We have added 1250 lines of C code [58] to the N-Variant 2.6.16 kernel, including additional support for 4 system calls that were previously disabled as well as bug fixes in `sys_poll`. The additional system calls are `sys_set_tid_address`, `sys_socketcall (recvmsg)`, `sys_epoll_create` and `sys_epoll_ctrl`. These system calls were disabled with the use of wrapper macros, where invocation caused the kernel to return an error value of `ENOSYS`.

Apache 2.2.3 was successfully run on our modified N-Variant kernel in a uniprocessor virtual machine. Apache was manually annotated with one sequential region on a shared work queue, which is the same modification that was made for our evaluation on the Replicant kernel discussed in the previous section. Apache was also configured with 2 processes and a maximum of 128 threads. Webstone was again used to generate system load with 40 client threads from a remote machine over a 100 Mbps switch. The virtual machine was hosted in Fedora Core 5, using VMware Workstation 5 [64].

Chapter 8

Related Work

This thesis draws on a large set of prior work from research in the areas of intrusion detection systems, fault-tolerance and application replay. In this chapter, we highlight those areas which influenced the design and implementation of the sequential region mechanism and our redundant execution system, Replicant, as a whole.

A preliminary description of Replicant can be found in [48] and the details of its replication internals are described in [67].

8.1 Intrusion Detection

Intrusion detection has been an active area of research for many years. Many systems and methods have been proposed that attempt to identify malicious behaviour. These are often divided into two groups, signature detection and anomaly detection [21]. In signature detection, a known pattern is matched to signal intrusion. In this case, it is assumed that the malicious behaviour has been previously observed and future occurrences can be identified. Anomaly detection takes the opposite approach. Here, we presume that malicious behaviour is unknown and that deviation from *normal* behaviour will signal an intrusion.

The concepts of signature and anomaly detection have been applied to both net-

work [8] and host [21, 51, 65] based systems. Each provides advantages and disadvantages over the other. Host based systems tend to have greater visibility of the attack vector and allow a closer examination of the vulnerability. However, these systems may expose vulnerable applications to greater risk if the attack is not contained and also use up valuable system resources such as memory and processing cycles.

Host based intrusion detection systems often use the concept of system call introspection to detect anomalous behaviour, as was demonstrated in early work by Forest and Longstaff [21]. These detection systems are able to intercept calls made to the operating system kernel by a potentially vulnerable process. System calls can be sanitized, verified and authorized before execution. Identifying a valid sequence of system call invocations can be a powerful mechanism for identifying processes that have been compromised. The two main approaches to this end are dynamic analysis and static code analysis.

Replicant shares many similarities with system call introspection intrusion detection systems of the past. While most systems use either static or dynamic methods to differentiate between valid and invalid system call traces, Replicant relies on running multiple concurrent instances of the same application binary. The valid sequence of system calls is determined by consensus among replicas, which are diverse enough to prevent a deterministic response to malicious input. Since system call traces are used for comparison, strict determinism is often required. Multi-threaded, shared memory applications are inherently non-deterministic, and our sequential region infrastructure provides the necessary mechanism that allows deterministic replication of shared memory operations.

8.1.1 Static Analysis

Wagner and Dean demonstrated how static code analysis can be used to model typical application behaviour [65]. This approach verifies that the application system call trace is consistent with what is expected from the source code. While this static analysis has no false positives and does not require training data, it suffers from several drawbacks.

Firstly, application code is required to perform the analysis, which may not be available in all cases. Secondly, achieving thorough coverage of all code paths can be very difficult, if not impossible for reasonably large programs. This is further complicated when considering system calls made my libraries that are dynamically linked in and due to the large state space.

8.1.2 Dynamic Analysis

Dynamic analysis is an alternative which gathers data through a runtime training period. Sekar et al. use this training data to produce a finite state automaton (FSA) that represent *normal* behaviour [51]. This FSA is used at runtime to validate the sequences of system calls made by the program being monitored. Using dynamically generated training data still suffers from the potential of poor coverage, because all runtime paths must be exercised in order to have a complete FSA. This is not trivial considering that there are typically several paths which are not commonly exercised. Other approaches such as VtPath [18] uses call stack information to improve on the accuracy.

Training data used in system call sequence validation, can be generated statically or dynamically. Generally these systems tend to tradeoff performance and accuracy. This area has been a hot topic in recent years and several pieces of work such as [44, 25, 19] have looked toward improving this tradeoff. [23] tries to bring the static and dynamic approaches together.

8.2 Program Isolation

Replicant guarantees that, although an application may be compromised by a malicious attack, attempts to modify system state by a compromised application will remain isolated. This is similar to other isolation techniques, such as FreeBSD Jail [30] and `chroot`, which, at a minimum, isolate the file system name space available to an appli-

cation. Replicant is able to enforce isolation without limiting an applicant's view of the file system, at the cost of maintaining block level Copy-on-Write (COW) metadata.

Systems that interpose on system calls to isolate untrusted processes are referred to as sandboxes. They limit an application's ability to interact with the operating system by sanitizing and verifying system calls. A sandbox may also provide a private view of system resources such as the work by Liang and Sekar has shown [36]. In their Alcatraz system, file operations are redirected to a modification cache and only committed once modification have been verified. This prevents malicious file modification by an untrusted process. This is very similar to Replicant's *harness*. The harness is responsible for providing each replica with its own private view of the operating system, allowing them to execute independently without being aware that outputs are being buffered.

8.3 Redundant Execution

Replicant is a kernel level redundant execution system. It allows mostly unmodified, modern applications to be replicated by the operating system to enforce security guarantees. Many redundant execution systems have been implemented in the past, with varying goals and motivations. The most relevant redundant execution systems to our work stem from the areas of fault-tolerance and diversity.

8.3.1 Fault-Tolerance

Fault-tolerance in distributed systems has long been an area of intense research focus. The Byzantine Generals Problem [34] outlined the difficulty in achieving consensus among a set of independent, potentially malicious, modules. Byzantine fault tolerance is an area of research which explores algorithms for mitigating failed components in a system that may continue interacting and exchanging messages inconsistently [31, 39, 7]. This is similar to Replicant's consensus among replicas in some respects. However, the problem

is simplified, because replicas are not distributed and do not rely on messages passed between one another, but rather use the omnipotent kernel to arbitrate on decisions.

Algorithms designed for consensus among replicas for fault tolerance generally assume very simple applications. For example, the Paxos algorithm for implementing fault-tolerant distributed systems is a simple state machine voting algorithm [33]. It assumes that there are several replicas that can be modelled as *deterministic* finite state machines (FSM). The applications we have studied do not generally have deterministic states. The sequential region mechanism described in this thesis could be extended to provide deterministic shared memory access across remote hosts. Such a mechanism would be needed to ensure *deterministic* execution of a distributed set of replicas.

Hardware redundancy has enjoyed a long history of use to improve system reliability and availability. Complex hardware systems have long since promoted redundancy for mission critical tasks [1]. For example, HP's NonStop architecture has its roots in systems designed in 1974 by Tandem Corp. [4], and IBM's S390 Microprocessor also featured redundant hardware functional units [57]. More recently, commodity hardware trends toward simultaneously threaded and multi-core processors have renewed interest in hardware-based redundant execution systems, such as SRT [50] and SlicK [46]. Because of the better visibility into the hardware that these solutions have, they are better able deal with the non-determinism that occurs between replicas. Unfortunately, they are at the wrong semantic level to be able to correlate system calls among replicas that are slightly different, as is needed to detect security violations. It will also be difficult for virtual machine monitor-based (VMM) solutions to compare replica outputs due to the semantic gap that exists between the VMM and the OS [9, 38, 12].

Rx attempts to recover from software faults through the use of a checkpoint rollback system [49]. Rather than executing the replicas simultaneously, Rx repeatedly detects a system crash and re-executes the application in a slightly different environment until one of the re-executions does not crash. Rx solves a different problem than the one

Replicant solves since Rx is trying to allow crashed applications to continue executing, while Replicant is trying to detect and eliminate malicious or erroneous activity from a group of replicas. However, Rx and Replicant are similar in that they both attempt to add diversity into re-execution. Future work in Replicant may explore Rx-like diversity in redundant execution, such as varying the version of `glibc`.

8.3.2 Security and Diversity

The idea of application-level redundancy for reliability was introduced in 1977 by Avizienis et al. with N-version programming [2] and recently there have been a plethora of projects that introduce diversity into replicas for the purposes of increasing security and reliability. The N-Variant [14] framework aims to provide highly secure systems by introducing differences between replicas such that it becomes very hard for an attacker to compromise them all with the same input. On the other hand, TightLip [68] provides one replica with access to sensitive data, while providing the other replica with "scrubbed" data. If the outputs of the replicas diverge, then the kernel can detect that the application may be leaking sensitive data and take appropriate action. Similarly, Doppelganger uses two web browsers with different cookie jar contents to detect which cookies need to be stored and which can be safely discarded [54].

An important short coming of previous work in this area is the inability to deterministically replicate shared memory accesses. Our work focused on filling in this gap, in a way that would allow shared memory workloads to be replicated on multiprocessor systems.

8.4 Replay Debugging

Because Replicant intercepts, records and replays system call inputs, it is also related to systems that support deterministic replay. For example, both LibLog [24] and Flash-

back [60], record system calls from within an OS kernel, and ReVirt [17] supports replay for an entire virtual machine. ReVirt can replay asynchronous interrupts by recording the instruction pointer and branch taken counter, thus allowing ReVirt to replay the scheduling of processes precisely. Unfortunately, as the ReVirt authors point out, this technique does not enable deterministic replay on a multiprocessor.

The DieHard project randomizes heap usage across several replicas to detect memory safety violations [3]. Replicant can provide similar guarantees, but does so from the operating system kernel rather than in user space. This allows Replicant to be used for security applications. Also, DieHard cannot deterministically replicate shared memory workloads and only supports very limited applications. A similar project, Exterminator [43], uses the DieHard memory randomization engine, but is able to automatically generate a patch when an error occurs. One of Exterminator's modes of operation uses replication to compare multiple heap layouts when performing its analysis. However, Exterminator's replication determinism is limited to application that do not communicate through shared memory.

Finally, our work is most closely related to previous work by LeBlanc et al. and their efforts in debugging parallel programs [35]. With *Instant Replay*, the authors were able record the relative ordering of significant events and use that order during replay to ensure that processes accessing shared recourses would do so in a deterministic ordering. This also included the order in which shared memory regions were accessed. However, Instant Replay benefited from many simplifying assumptions. The applications supported for replay did not receive any non-deterministic input from the operating system on successive runs, such that no input buffering was required. Also, Instant Replay is simply an event log suitable for replay of simple applications. Our sequential region approach has been demonstrated to effectively replicate access ordering to shared memory on large scale, modern applications in real-time. Further our infrastructure is implemented in the operating system kernel, allowing it to be suitable for security sensitive applications.

Chapter 9

Future Work

The methods outlined in this thesis have identified the feasibility of annotating application source code in order to support redundant execution of shared memory workloads on multiprocessor systems. While our prototype implementation achieves the goals we set out in Section 4.1, several important improvements warrant discussion: performance, n-replica generalization and automatic annotation.

9.1 Performance Improvements

There are two inefficiencies that were described in our implementation that deserve special consideration. (1) We have to search for the correct domain structure in the kernel when starting or ending a sequential region. Although caching is used to eliminate much of the searching, an application is often required to perform a linear search with the added benefit of a most recently used (MRU) optimization. (2) Applications will experience slowdown due to out of turn waiting.

Although linear search performs well on applications with a small number of sequential regions at any given time, such as Apache and Squid, applications with many sequential regions will suffer drastic performance degradation when searching. While the MRU optimization is sufficient for our test environment, a more robust and complete implemen-

tation should consider a data structure that facilitates scalable search while maintaining efficiency in insertion and deletion. Two potential data structures that would serve our purpose are hash tables and trees.

A hash table could provide *O(1)* lookup, but would require some static amount of memory to be allocated upfront. A hash table implementation would also likely use chaining to handle collisions, which has the potential to degrade to the performance of the chain data structure, such as a linked list or a tree. A better solution may be to use a balanced tree data structure, such as a red-black tree [27]. A red-black tree is a self balancing binary search tree that is already widely used in the Linux kernel, particularly in the virtual memory management subsystem. This solution would have no memory preallocation requirement, which is better suited to the typically small number of sequential region domains that we have observed. Also, because of the binary search property, lookup is performed in *O(log n)* time. For an application such as MySQL, which initializes slightly more than 40,000 sequential region domains in our test configuration, a red-black tree lookup would require only slightly more than 10 node traversals in the worst case.

To reduce out of turn waiting, applications should use as few sequential regions as possible, and sequential regions that do exist should be distributed over fine grained domains. This mantra maps well to existing lock practice. However, applications that experience slowdown due to out of turn waiting can also benefit from more intelligent scheduling. An intelligent, or sequential region aware, scheduler may be able to help avoid out of turn waiting by only scheduling an application that is next to acquire a sequential region. Unfortunately, the scheduler does not have *a priori* knowledge of when an application will attempt to enter a sequential region. However, if the scheduler could detect when a sequential region domain has a backup of waiting threads, it could attempt to prioritize the known next task. The main disadvantage of implementing an intelligent scheduler would be the potential for task starvation, or priority inversion [52].

9.2 N-replica Generalization

Extending the current implementation to n-replica is conceptually straight forward. Each domain structure, depicted in Figure 5.4, currently contains 2 sub-domains. The purpose of the sub-domain is to maintain replica specific data, such as the current task holding a domain busy and a list of waiters. In order to support any number of replicas, n, we first extend the domain structure to contain n sub-domains.

A generalized solution would still require one replica to act as master on a per-domain bases. As in the 2-replica case, the replica acting as master would be dynamically assigned to the replica that is currently executing furthest ahead. All other replicas would then follow the ordering specified by the master replica. Our current implementation maintains the ordering task list as a member of the domain structure. It is possible to maintain this property in a n-replica implementation, however, it would require that each replica maintain its progress in the tasks list. Alternatively, the tasks list can be moved into each sub-domain, which would require more allocation, but result in a simplified algorithm. The only other added complexity introduced by generalizing to a n-replica design would be storing the order for each replica and correctly waking up waiting threads from each replica. Each of these requirements may be implemented independently and would only require a modest engineering effort above what is currently implemented.

Our system currently maintains one list of domains per replicated application. The list has two sortings where one is maintained in initialized order and the other is maintained in most recently used order. It would be more efficient to maintain a data structure of existing domains per-replica. If the searching performance improvements described in the previous section are implemented, we could have a tree data structure linking each of the sub-domains of a particular replica. This would allow each replica to search the list of domains efficiently while still allowing arbitrarily different label values to identify a sequential region domain across replicas.

9.3 Automatic Annotations

Our current heuristic is to map POSIX Threads API calls to sequential regions in the application. Although this method seems logical, it has several drawbacks in practice. First, we require constructs such as domain aliasing to bridge the gap between the application source code and our in kernel implementation. Secondly, many shared memory accesses are performed outside of critical sections protected by locks. This section outlines two potential alternative approaches to automatically insert sequential region annotations.

9.3.1 `glibc` Integration

While annotating an application around the POSIX Threads library API provides a portable and intuitive solution, it is not able to take advantage of the underlying implementation of the GNU C Library, `glibc`. This is exemplified by the requirement of domain aliasing described in Section 5.3. By directly instrumenting the `glibc` library, we can remove the gap that exists between application level annotations and our kernel level support. This does not provide a method for annotating code segments that access shared memory outside of locks, but does allow for a cleaner interface than that described in Section 5.4.

As a first attempt at annotating the `glibc` library, we have compiled version 2.4-4 with sequential regions in all of the *low level lock* macros, as well as the macros used for atomic operations. Our annotations have focused only on the `i386` architecture. These macros are used throughout the `glibc` source code to support mutual exclusion. For example, a call to the `stdio` library function `printf` will result in a low level lock operation on the file stream for standard output. This prevents multiple threads from mangling their output when writing to the same file descriptor. The low level lock facility in `glibc` is also used in the NPTL implementation of the POSIX Threads standard. By annotating the low level lock macros and those used for atomic operations, the kernel is

able to interpose on most of the explicit shared memory access made by `glibc`.

Unfortunately, all of our added sequential regions currently belong to be same domain. This is due to the lack of an internal lock initialization interface in `glibc`. Any internal function needing mutual exclusion support can initialize a lock statically with the value `LLL_MUTEX_LOCK_INITIALIZER`, or simply set the value to zero. Locks are often members of a structure, which become initialized when the structure is zeroed out with a `memset`-like operation. A significant engineering effort would be required to locate all of the lock initializations and in turn insert the initialization of a sequential region domain. An alternative solution would be to automatically infer a new domain in the kernel on the first observation of a new domain identifier. We have not explored this approach and it remains as an interesting future work.

If `glibc` is annotated with extra calls to the kernel on every shared memory access, then it would also be desirable to enable the sequential region annotations at runtime, depending on the needs of the application. The `pthread_create` function takes a `pthread_attr_t` pointer as an argument that can be used to specify attributes to be applied to the new thread. Several of the available attributes are platform specific and are denoted with a trailing `_NP`, indicating that these options are *not portable*. Some attributes include error checking enforcement, `PTHREAD_MUTEX_ERRORCHECK_NP`, and also allowing recursive calls to locking and unlocking functions (recursive calls are typically not defined under the POSIX standard), `PTHREAD_MUTEX_RECURSIVE_NP`. A similar option could be added for redundant execution support, e.g. `PTHREAD_MUTEX_REPLICATABLE_NP`. By enabling this attribute, the `glibc` implementation could set a flag in the per-thread state that indicates that sequential region calls must be made to the kernel on shared memory accesses.

There remain many cases in the `glibc` source code where hand written assembly files use atomic operations, such as compare-and-exchange, which fall outside of our instrumentation. In order to have a completely annotated library, all of these cases must be

located. Unfortunately, this requires significant engineering effort due to locating target code sections and understanding the semantics well enough to place efficient instrumentation. However, it is important to note that, because Replicant allows replicas to execute independently, strict determinism in `glibc` is not required, and we have not had the need to locate these remaining unannotated regions. The following section describes a potential method for inferring the complete placement of sequential regions based on source code analysis.

9.3.2 Static Analysis

In general, the use of locks as a mechanism for determining sections of code that access shared memory has proved inadequate. This is best demonstrated by the effort required to port applications for redundant execution on Replicant in Table 6.1. A potentially promising alternative may be to perform static analysis on the application source code to determine all instructions that operate on shared memory. In this effort, a source-to-source transformation tool, such as CIL, could be used to insert the required annotations for sequential region initialization, entry and exit points, and so on [42].

There are several drawbacks to this approach. First, the translation tool would need to understand all language constructs that are used in manipulation of shared memory, including inline assembly code. The tool would also need to understand how annotations should be placed for efficiency (e.g. fine grained sequential regions). Second, it would require a very intelligent source code analysis tool to identify the minimum set of sequential regions that are needed for replication on Replicant. Recall that Replicant only requires shared memory accesses that effect the value of external outputs to be annotated. A formal description of this class of sequential region has not been identified that would be suitable for use in source code analysis.

Chapter 10

Conclusion

We feel the recent trend toward multiprocessors on commodity architectures will facilitate the use of redundant execution as an inherently parallelizable application for under-utilized processing cores. Previous work has shown that kernel level redundant execution can be successfully applied to security applications [14, 68]. However, these systems have been unable to support redundant execution of shared memory workloads. This is an important class of application because multi-threading is commonly used to parallelize applications, where shared memory is used for inter-thread communication and message passing. As multiprocessors become more prevalent on commodity hardware, the use of multi-threading will increase in an effort to effectively utilize processing cycles.

Multi-threaded applications are non-deterministic because the scheduling of individual threads will depend on many non-deterministic factors, such as the amount of time spent waiting for device input. If a thread is able to acquire a lock without waiting in one run of a multi-threaded application, there is no guarantee that the same thread will deterministically acquire the same lock at the same time on successive runs of that application. In order to make shared memory access deterministic, we record the order of threads that access each area of shared memory and enforce that this order is replicated in each instance of the application. Since access to shared memory is not made explicit

to the operating system kernel by user space threads, we use source code annotations to inform the kernel before and after shared memory access is made. While manual annotation can be a burden on the application developer, we have found that, for the most part, annotations can be inferred from the use of locks already present in an application. Further, when a threading library, such as `libpthread`, is used, access to application source code is not required. This is because the use of synchronization primitives are typically made through library calls, which can be intercepted and annotated at runtime.

Multi-threaded applications normally protect shared memory access with synchronization primitives. Our system uses synchronization primitives, such as lock and unlock operations, to automatically infer the placement of annotations. However, we have found that there are cases where developers are able to increase performance by avoiding synchronization. Narayanasamy et al. define this behaviour as as benign data races and have shown how to automatically classify these races into five descriptive categories [41]. Our system cannot currently detect these races, and manual effort is required to find and properly annotate their use. Automatically annotating benign data races remains interesting future work.

Kernel level redundant execution is an attractive use for the spare processing cycles that are predicted on future commodity multiprocessors. While redundant execution is complicated by non-deterministic multi-threaded applications, our work demonstrates, as proof-of-concept, that deterministically scheduling threads over shared memory operations is a feasible approach for enforcing the required determinism with acceptable overheads.

Appendix A

Example Annotated Application

A section of a medium sized application, depicted below, has been annotated with our added system calls so that it may now be deterministically replicated. The application is slightly simplified for clarity, error checking has been removed, and only details the consumer thread function.

In actuality, the placement of annotations are entirely inferred automatically from the use of `pthread` functions.

```
1  struct queue {
2          pthread_cond_t not_empty;
3          pthread_mutex_t lock;
4          int count;              /* number of items */
5          struct node *head;
6  };
7
8  void * consumer (void *arg)
9  {
10         struct queue *queue = (struct queue *) arg;
11         struct node *node;
12         int id = gettid();      /* linux only, not portable. */
13
14         while (1) {
15                 /* begin the critical section */
16                 begin_seq(&queue->lock);
17                 pthread_mutex_lock(&queue->lock);
```

```
18
19                  /* if the queue is empty, then we must wait for
20                     the producer to feed us something. */
21                  if (queue->count == 0) {
22                          pthread_cond_wait(&queue->not_empty,
23                              &queue->lock);
24
25                          if (queue->count == 0)
26                                  goto out_unlock;
27                  }
28
29                  node = dequeue(queue);
30
31                  /* end the critical section */
32                  pthread_mutex_unlock(&queue->lock);
33                  end_seq(&queue->lock);
34
35                  printf(``-- thread %d consumed %d\n'', id,
36                      node->value);
37                  free(node);
38          }
39  out:
40          return NULL;
41  out_unlock:
42          pthread_mutex_unlock(&queue->lock);
43          end_seq(&queue->lock);
44          goto out;
45  }
46
47  int main (int argc, char **argv)
48  {
49          struct queue queue;
50          pthread_t p, c;
51
52          /* initialize the mutex and the conditional */
53          pthread_mutex_init(&queue.lock, NULL);
54          pthread_cond_init(&queue.not_empty, NULL);
55
56          init_seq(&queue.lock);
57          alias_seq(&queue.lock, &(queue.not_empty).__data.__futex)
58
59          /* create the consumer */
60          pthread_create(&c, NULL, consumer, &queue);
61  ...
```

Bibliography

[1] The National Aeronautics and Space Administration (NASA). Report on a Technical Comparison of the Apollo Spacecraft Guidance Computer with a Proposed New Design, July 1963. 8.3.1

[2] Algirdas Avizienis and Liming Chen. On the Implementation of N-Version Programming for Software Fault Tolerance During Execution. In *Proceedings of the 1977 IEEE International Computer Software & Applications Conferencen (COMPSAC)*, pages 149–155, November 1977. 8.3.2

[3] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 158–168, June 2006. 1, 8.4

[4] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. NonStop Advanced Architecture. In *International Conference on Dependable Systems and Networks (DSN)*, pages 12–21, June 2005. 1, 2.1, 8.3.1

[5] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., November 2005. 1, 5.1

[6] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault-tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, February 1996. 2.1, 3.2

[7] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, November 2002. 8.3.1

[8] Brian Caswell, Jay Beale, James C. Foster, and Jeremey Faircloth. *Snort 2.0 Intrusion Detection*. Syngress, February 2003. 8.1

[9] Peter M. Chen and Brian D. Noble. When Virtual Is Better Than Real. In *Proceedings of the 8th Usenix Workshop on Hot Topics in Operating Systems (HOTOS)*, May 2001. 8.3.1

[10] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 13th USENIX Security Symposium*, pages 321–336, August 2004. 6.5

[11] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding Your Garbage: Reducing Data Lifetime. In *Proceedings of the 14th USENIX Security Symposium*, pages 331–346, August 2005. 6.5

[12] Alan L. Cox, Kartik Mohanram, and Scott Rixner. Dependable ≠ Unaffordable. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability*, pages 58–62, October 2006. 8.3.1

[13] Benjamin Cox. Using the N-Variant System Framework, 2007. `http://www.cs.virginia.edu/ btc4w/software/nvariant/` (Last accessed: 2007/07/30). 7.2

[14] 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. 1, 2.1, 4.1, 7.2, 8.3.2, 10

[15] Edsger W. Dijkstra. Hierarchical Ordering of Sequential Processes. In *Acta Informatica*, pages 115–138, June 1971. 2.2.1

[16] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux, February 2005. `http://people.redhat.com/drepper/nptl-design.pdf` (Last accessed: 2007/08/11). 2.2.2

[17] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, December 2002. 3.1, 3.2, 8.4

[18] Henry Hanping Feng et al. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003. 8.1.2

[19] Henry Hanping Feng, Jonathon T. Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P. Miller. Formalizing Sensitivity in Static Analysis for Intrusion Detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004. 8.1.2

[20] Michael J. Fischer and Alan Michael. Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. In *Proceedings of the 1982 Symposium on Principles of Database Systems*, pages 70–75, 1982. 2.1

[21] Stephanie Forrest and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, May 1996. 8.1

[22] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, June 2002. 2.2.3

[23] Debin Gao, Michael K. Reiter, and Dawn Song. On Gray-Box Program Tracking for Anomaly Detection. In *Proceedings of the 2004 Usenix Security Symposium*, August 2004. 8.1.2

[24] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay Debugging for Distributed Applications. In *Proceedings of the 2006 Annual Usenix Technical Conference*, pages 189–195, June 2006. 8.4

[25] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Efficient Context-Sensitive Intrusion Detection. In *Proceedings of the 11th Network and Distributed System Security Symposium*, February 2004. 8.1.2

[26] The Open Group. IEEE Std 1003.1: pthreads.h, 2004. `http://www.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html` (Last accessed: 2007/06/14). 2.2.1

[27] Leo J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *Proceeding of the 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978. 9.1

[28] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the 22nd Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, July 2003. 2.2

[29] Intel Corp. Teraflops Research Chip, 2007. `http://www.intel.com/research/platform/terascale/teraflops.htm` (Last accessed: 2007/07/30). 4.1, 7.1.1

[30] Poul-Henning Kamp and Robert N.M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE)*, May 2000. 8.2

[31] Kim Potter Kihlstrom, L.E. Moser, and P.M Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences*, January 1998. 8.3.1

[32] Rivka Ladin, Barbara Liskov, and Sanjay Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactmns on Computer Systems*, 10(4):360–391, November 1992. 2.1

[33] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):18–25, December 2001. 2.1, 8.3.1

[34] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982. 2.1, 8.3.1

[35] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471–482, April 1987. 8.4

[36] Zhenkai Liang, V.N. Venkatakrishnan, and R. Sekar. Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC)*, December 2003. 8.2

[37] Inc. Linux Kernel Organization. The Linux Kernel Archives, 2007. `http://kernel.org/` (Last accessed: 2007/07/30). 1

[38] Dominic Lucchetti, Steven K. Reinhardt, and Peter M. Chen. ExtraVirt: Detecting and Recovering from Transient Processor Faults. In *Work-in-progress, ACM Symposium on Operating Systems Principles (SOSP)*, October 2005. 8.3.1

[39] Dahlia Malkhi and Michael K. Reiter. Secure and Scalable Replication in Phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 1998. 8.3.1

[40] MySQL AB, 2007. `http://www.mysql.com/` (Last accessed: 2007/08/11). 6.4

[41] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2007. 4.4, 10

[42] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 2002 Conference on Compiler Construction*, pages 213–228, 2002. 9.3.2

[43] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: Automatically Correcting Memory Errors with High Probability. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, June 2007. 1, 8.4

[44] Yoshihiro Oyama, Koichi Onoue, and Akinori Yonezawa. Speculative Security Checks in Sandboxing Systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, April 2005. 8.1.2

[45] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 Usenix Annual Technical Conference*, pages 199–212, June 1999. 2.2

[46] Angshuman Parashar, Anand Sivasubramaniam, and Sudhanva Gurumurthi. SlicK: Slice-based Locality Exploitation for Efficient Redundant Multithreading. In *Pro-*

ceedings of the 12th International Conference Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 95–105, October 2006. 8.3.1

[47] The PaX Team, 2007. `http://pax.grsecurity.net` (Last accessed: 2007/08/11). 2.1

[48] 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)*, May 2007. 1, 2.1, 3.3, 8

[49] Feng Qin, Joe Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failure. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 235–248, Oct 2005. 8.3.1

[50] 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. 8.3.1

[51] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001. 8.1, 8.1.2

[52] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990. 9.1

[53] 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. 2.1

[54] Umesh Shankar and Chris Karlof. Doppelganger: Better Browser Privacy Without the Bother. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 154–167, October 2006. 8.3.2

[55] Nir Shavit and Dan Touitou. Software Transactional Memory. *Distributed Computing, Special Issue*, 10:99–116, 1997. 2.2

[56] Vikram Shukla. Linux threading models compared: LinuxThreads and NPTL, July 2006. `http://www-128.ibm.com/developerworks/linux/library/l-threading.html?ca=dgr-lnxw07LinuxThreadsAndNPTL` (Last accessed: 2007/06/14). 2.2.2

[57] Timoethy J. Slegel, Robert M. Averill III, Mark A. Check, Bruce C. Gaimei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. IBM's S/390 G5 Microprocessor design. *IEEE Micro*, 19(2):12–23, March 1999. 8.3.1

[58] SLOCCount: Counting Physical Source Lines of Code, 2007. `http://www.dwheeler.com/sloccount/` (Last accessed: 2007/08/11). 6.1, 7.2

[59] Squid Web Proxy Cache, 2007. `http://www.squid-cache.org/` (Last accessed: 2007/08/11). 6.3

[60] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the 2004 Annual Usenix Technical Conference*, pages 29–44, June 2004. 3.1, 8.4

[61] SysBench: A System Performance Benchmark, 2007. http://sysbench.sourceforge.net/ (Last accessed: 2007/08/11). 7.1.2, 7.1.3

[62] The Apache HTTP Server Project, 2007. http://httpd.apache.org/ (Last accessed: 2007/08/11). 6.2

[63] The Firefox Web Browser, 2007. http://www.mozilla.com/firefox/ (Last accessed: 2007/08/11). 6.5

[64] VMware, Inc., 2007. http://www.vmware.com (Last accessed: 2007/08/11). 7.2

[65] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001. 8.1, 8.1.1

[66] Webstone: The Benchmark for Web Servers, 2007. http://www.mindcraft.com/benchmarks/webstone/ (Last accessed: 2007/08/11). 7.1.2, 7.1.3

[67] Ian J. Sin Kwok Wong. Kernel Support for Redundant Execution on Multiprocessor Systems, July 2007. 1, 2.1, 3.3, 8

[68] Aydan Yumerefendi, Benjamin Mickle, and Landon Cox. TightLip: Keeping Applications from Spilling the Beans. In *4th Symposium on Networked Systems Design and Implementation (NSDI)*, April 2007. 1, 2.1, 4.1, 8.3.2, 10