Kernel Support for Redundant Execution on Multiprocessor Systems

by

Ian J. Sin Kwok Wong

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 Redundant Execution on Multiprocessor Systems

Ian J. Sin Kwok Wong

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2007

Redundant execution systems increase computer system reliability and security by simultaneously running multiple replicas of an application and comparing their outputs. Currently, no redundant execution system can account for the non-determinism that occurs when multi-threaded applications execute on multiprocessors, making such systems ineffective on the very hardware that could benefit them most.

This thesis is part of a larger project called *Replicant* where we explore a fundamentally different approach to redundant execution. Rather than requiring the execution of the replicas to be identical, Replicant permits replicas to diverge and only makes outputs that a majority of application replicas agree upon externally visible. Output value divergences are suppressed using determinism annotations, where needed, at some performance cost. This removes a great deal of synchronization among replicas and improves performance. This thesis focuses on the mechanisms that support redundant execution and handle non-determinism in order. We implemented and evaluated a 2-replica prototype.

Acknowledgements

I would express my gratitude to Professor David Lie for his patient supervision, his financial support and the large amount of time we spent discussing this research.

I am thankful to Jesse Pool, with whom I worked on many aspects of this project, and to Lionel Litty, Tom Hart, Richard Ta-Min and Professor Ashvin Goel for their continuous feedback.

I am also very grateful to my family and my girlfriend Mary Jane for their much needed moral support.

Finally, I would like to thank the Edward S. Rogers Sr. Ontario Graduate Scholarship fund as well as the department of Electrical and Computer Engineering for their financial support.

Contents

List of Tables

List of Figures

List of Acronyms

Acronym	Definition
ASLR	Address Space Layout Randomization
COW	Copy-On-Write
FDT	File Descriptor Table
IDS	Intrusion Detection Systems
IPC	Inter-Process Communication
OS	Operating System
MMU	Memory Management Unit
SMT	Simultaneous Multithreading
VMM	Virtual Machine Monitor

Chapter 1

Introduction

Recent trends in computing hardware indicate that the vast majority of future computers will contain multiple processing cores on a single die. By the end of 2007, Intel expects to be shipping multi-core chips on 90% of its performance desktop and mobile processors and 100% of its server processors [15]. These multiprocessors can offer increased performance through parallel execution, as well as more system reliability and security through redundant execution.

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 then compares the outputs of each replica, relying on the premise that their execution is deterministic based on their inputs, so that any divergence in their outputs must indicate a problem. For example, executing identical replicas has been used to detect and mitigate soft-errors [5]. More recently, there have also been several proposals to execute slightly different replicas to detect security compromises [10], and private information leaks [45].

Unfortunately, redundant execution systems to date have not been able to support multi-threaded programs on multiprocessor systems, even though the growing preva-

lence of multiprocessors will encourage the use of multi-threaded programming. 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 through shared memory. Allowing the order of this communication to diverge among replicas can cause a *spurious divergence*, which is not the result of a failure or violation. This undermines the primary premise on which redundant execution depends. Naïve solutions to make communication deterministic, such as trapping on each shared memory access, can result in unacceptable performance degradation. This inability to efficiently deal with the non-determinism that exists when running multi-threaded programs on multiprocessors threatens the future feasibility of redundant execution systems on the very hardware that benefits them the most.

This thesis is part of a larger project called *Replicant*, where the key insight is that redundant execution systems can be made to run efficiently on multiprocessors by enabling them to tolerate non-determinism, rather than forcing them to eliminate it completely. Replicant places each replica in an OS sandbox and only loosely replicates the order of events among the replicas. Replicant then compares the outputs of the replicas and only externalizes outputs that occur in the majority of the replicas, thus making the replicas appear to the outside world as one process whose behavior is determined by the majority. When identical event ordering among replicas is required, e.g. when replica output values diverge under normal execution, Replicant can be instructed to enforce such an ordering through *determinism annotations*, which need to be inserted by the application developer [27, 26]. Our experiences show that the number of determinism annotations required is related to the nature and amount of communication among threads in an application, and can, for the most part, be inferred from the use of locks in the application.

The goal of Replicant is to increase the security and reliability of computing systems at reasonable performance costs. As an example, Replicant can be used to prevent an adversary from exploiting buffer overflow vulnerabilities in applications by varying the

address space layout of replicas. Address space layout randomization (ASLR), on its own, has been shown to be only a probabilistic defense mechanism and can be brute-forced [31]. While an adversary may be able to successfully overflow a subset of replicas, to subvert the *externally visible* behavior of the application, an adversary must compromise and control a majority of the replicas. By increasing the number of replicas, we can make it arbitrarily improbable that an adversary will be able to simultaneously compromise enough replicas with the same attack. Replicant can also improve the availability of a system by removing any crashed or unresponsive replicas, thus allowing the remaining replicas to carry on execution.

As will be explained in the next chapter, there are two classes of non-determinism that are exhibited by multi-threaded applications, both of which can be handled by Replicant: *non-determinism in order* and *non-determinism in value*. The focus of this thesis is on the first sub problem. We discuss the design and implementation of the kernel mechanisms, which handle the redundant execution of multi-threaded applications that do not exhibit non-determinism in value. Those that do exhibit such non-determinism require determinism annotations, a concept which we will introduce in Chapter 3. The design and implementation of determinism annotations is the topic of another thesis and is discussed in [27, 26].

1.1 Contributions

The contributions of this thesis are two fold. First, we describe the mechanisms that enable Replicant to support redundant execution and to tolerate non-determinism in order between replicas. We then evaluate the correctness of the output produced by applications running on Replicant as well as the performance of a 2-replica system using three SPLASH-2 parallel benchmarks [44] and some of our own microbenchmarks.

1.2 Thesis Structure

In the next chapter, we give some background and explain in detail the difficulties associated with executing multi-threaded applications redundantly on multiprocessors, which we classify into two problem categories. Chapter 3 provides an overview of the complete Replicant system that handles both non-determinism in order and non-determinism in value, and as a result any multi-threaded application. The rest of the thesis then focuses on the mechanisms that support redundant execution and non-determinism in order specifically. Chapter 4 elaborates on the architecture of Replicant, and Chapter 5 describes the implementation of a 2-replica system, which can be generalized to an n-replica system. We then evaluate the performance of Replicant, reported in Chapter 6, both at the macro and micro level for applications that do not exhibit non-determinism in value and discuss Replicant's limitations in Chapter 7. Related work is dealt with in Chapter 8 and we conclude in Chapter 9.

Chapter 2

Background

This chapter gives some background on the concept of redundant execution and ASLR. The effectiveness of ASLR for an application running redundantly is then illustrated by way of an example. However, redundantly executing multi-threaded applications on multi-core hardware have a few problems, which are described in detail in the last section of this chapter to motivate Replicant.

2.1 Redundant Execution

The idea behind redundant execution is to perform the same task multiple times and ensure that all re-executions produce the same consistent result. These systems are typically supported by a voting mechanism to determine which results are correct, in the face of inconsistency produced by some of the re-executions.

Redundant execution has been successfully applied for decades in expensive high-availability systems such as ATMs and life-critical systems such as aircrafts and space-crafts, which require fault-tolerant systems. There is a rich literature on fault tolerance and only a facet will be introduced here [39, 33]. Fault-tolerant systems can be implemented at the system-level and leverage design diversity of redundant sub-systems, i.e. each sub-system is implemented independently, but all of them conform to a common

specification. An example is the Boeing 777, where the redundant flight sub-systems exchange proposed outputs as votes before sending them out to the actuators [39].

Software-level fault tolerance uses the same idea of design diversity, with multiple versions of an application being developed for a common specification, a concept known as N-version programming [3]. The N-versions leverage language diversity and algorithm diversity among others. Such software systems also have a voting algorithm to decide on the outputs, which are determined by the majority.

Apart from fault-tolerant applications, the concept of software redundant execution has been used to detect security compromises [10] and private information leaks [45]. Our approach to redundant execution is to support multi-threaded applications on multiprocessor systems, which can be used to detect security compromises or improve reliability. A more comprehensive survey of existing redundant execution systems is presented in Chapter 8 alongside other related work.

2.2 Address Space Layout Randomization

Design diversity is a very desirable property but usually a very expensive one. Designing, manufacturing and testing specialized hardware is a very costly venture and the same is true for N-version software. This is only warranted in life-critical applications. Similar to how the cost of redundant systems that employ hardware design diversity can be reduced by utilizing cheaper commercial off-the-shelf components, the cost of diverse software can be reduced by using cheaper methods to inject diversity automatically. There are different means of achieving this goal, e.g. using different compilers on the same source tree [33] and ASLR [6, 7] among others.

In a security context, diversity in software is also desirable since it makes software appear different to attackers, making them harder to attack and more resilient against a fast propagating attack [36]. Since Replicant uses ASLR to diversify replicas to detect

memory corruption attacks, we focus on the concepts of ASLR in this section and delay discussion of other methods of diversity in Chapter 8 and Chapter 9.

ASLR reorganizes the layout of code and data in virtual memory without affecting the application's semantics and has low runtime overhead. It is effective against memory corruption attacks (e.g. buffer overflow attacks) and makes the job of the attacker harder. Typically, attack payloads consist of a hard-coded absolute address, also known as the *jump address*, which is based on the adversary's prior knowledge of an application's address space layout. The jump address defines the location in memory where the attacker wants to re-direct the control flow of an application, i.e. to the malicious code in the overflowed buffer in the case of a buffer overflow attack. With ASLR, the attacker now has to guess this jump address since it changes every time an application is loaded in memory and is thus different in each instance of an application.

The example in Figure 2.1 (a) shows how a simple buffer overflow attack works but more details can be found in [2]. The adversary crafts an attack payload that would overflow the buffer on the stack and overwrite the return address with a hard-coded jump address (`0xbeaddead`). When the return address is loaded into the program counter, control is transferred to the malicious code inside the overflowed buffer. Consider the case where an ASLR-protected application runs on Replicant, but unfortunately, the attacker correctly guesses a valid jump address in one of the replicas, thus compromising an instance of the application. This is illustrated in Figure 2.1 (b), where the attack payload successfully compromises Replica 1. Since the application is running on top of Replicant, the same attack payload (application input) is replicated to Replica 2. However, the same attack fails on Replica 2 because its address space layout is different from Replica 1, meaning that the hard-coded jump address does not point back to the overflowed buffer but to an arbitrary (illegal) address in memory. The system will fail-stop as Replica 2 will crash, making the attack detectable.

While ASLR is a very light-weight memory corruption detection technique, it is only

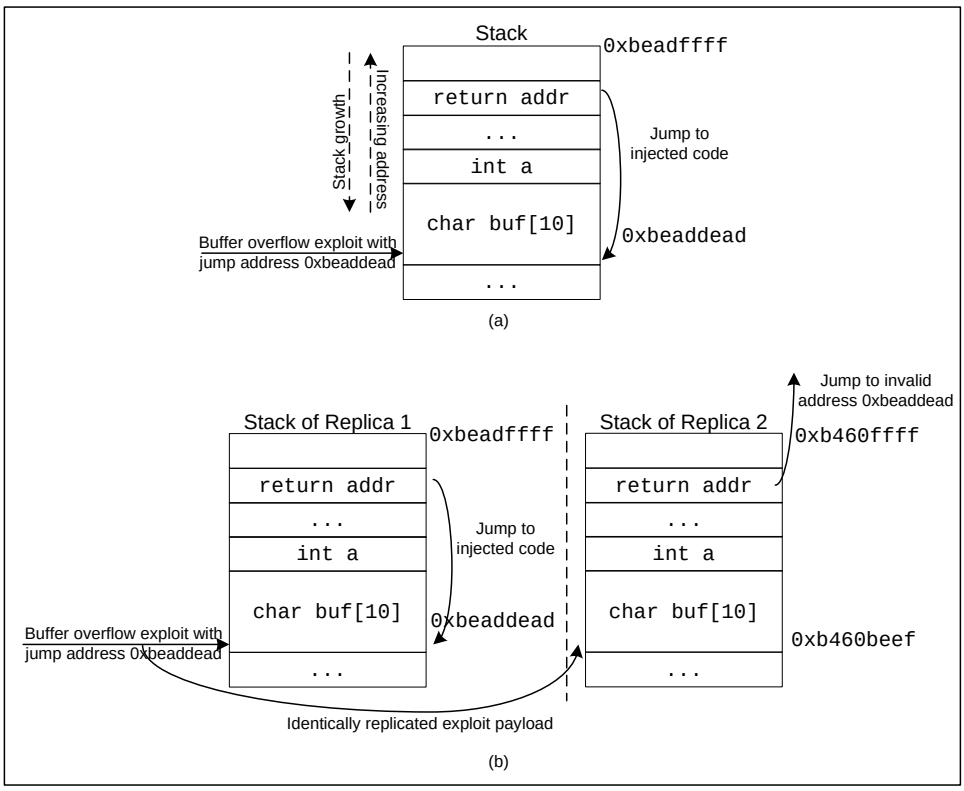

Figure 2.1: Address space layout randomization. (a) gives an example of how buffer overflow attacks typically work and (b) illustrates how replicas with ASLR would thwart an attack, even if one replica was successfully compromised by guessing the absolute jump address correctly.

probabilistic. It has been shown that on 32-bit systems, ASLR can be brute-forced within minutes [31]. In Replicant however, in order to subvert the externally visible behavior of an application, the adversary would have to compromise a majority of the replicas. Since the same exploit is replicated to all replicas, it can be made arbitrarily improbable that the attacker would be able to compromise the majority of replicas simultaneously, by increasing the number of replicas.

2.3 Problem Description

Redundant execution systems rely on the presumption that if inputs are copied faithfully to all replicas, any divergence in behavior among replicas must be due to undesirable

```
 1: int counter = 0;
 2:
 3: void thread_start() {
 4:     int local;
 5:
 6:     lock();
 7:     counter = counter + thread_id();
 8:     local = counter;
 9:     unlock();
10:
11:     printf("%d\n", local);
12: }
13:
14: void main() {
15:     thread_create(thread_start); // thread_id = 1
16:     thread_create(thread_start); // thread_id = 2
17:     thread_create(thread_start); // thread_id = 3
18: }
```

Replica 1:	**Replica 2:**
Thread 1 prints "**1**"	**Thread 2** prints "**2**"
Thread 3 prints "**4**"	**Thread 3** prints "**5**"
Thread 2 prints "**6**"	**Thread 1** prints "**6**"

Figure 2.2: Code example illustrating non-determinism in a multi-threaded program. Not only can the order of the thread outputs between Replica 1 and Replica 2 differ, but the contents of the outputs may differ as well.

behavior, such as a transient error or a malicious attack. On such systems, the replication of inputs and comparison of outputs are typically 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 2.2. 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, as illustrated below the program, the threads may update and print the counter in a non-deterministic order between the two replicas. In Replica 1, the threads print "1", "4" and "6" because they execute the critical

section in the order (1, 3, 2) by thread ID. On the other hand, the threads in Replica 2 print "2", "5" and "6" because they execute the critical section in order (2, 3, 1). This example demonstrates that multi-threaded applications may non-deterministically generate outputs in both different orders and with different values.

To avoid these spurious divergences, the redundant execution system must ensure that the ordering of updates to the counter is the same across 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 the `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 [11] and Flashback [35] 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, in order to support multi-threaded applications on a multi-core architecture, the redundant execution system must be able to handle outputs produced in non-deterministically different orders (non-determinism in order) 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 (non-determinism in value). 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.

Chapter 3

The Replicant System

The previous section illustrated the problems that redundant execution systems face when running multi-threaded applications on multiprocessors. In this chapter, we give an overview of Replicant and discuss how it handles both non-determinism in order and non-determinism in value. Mechanisms that handle non-determinism in order allow Replicant to support only multi-threaded applications where the non-determinism does not affect external output, but when determinism annotations are used, Replicant can handle any multi-threaded application.

3.1 Redundant Execution and Non-Determinism in Order

Replicant is a redundant execution system that supports multi-threaded applications on commodity multi-core processors with the goal of improving system security and reliability. For example, by randomizing the address space layout of each replica, Replicant can detect memory corruption attacks. This is an improvement over existing systems such as N-Variant [10] that do not support multi-threaded applications.

Replicant implements an input replicating and an output matching architecture that

is tolerant to the non-determinism in order, and only uses determinism annotations to enforce the ordering of events that can cause divergence in replica output values. Replicant loosely replicates the ordering of events among replicas and compares outputs, externalizing only those *confirmed* (i.e. independently reproduced) by the majority of replicas. From Figure 2.2, Replicant will resolve the different ordering of `printf` by buffering the outputs, then matching up the same `printf` instances and externalizing the outputs as they are confirmed.

A unique aspect of Replicant is that it allows replicas to execute independently in an OS sandbox, as opposed to executing in lockstep. This creates greater opportunities for concurrency both among threads within a replica and among replicas, hence leveraging the parallelism available on multiprocessor platforms.

Moreover, from the stand point of an external observer (e.g. the user or other applications running on the system), replicas appear as a single application whose behavior is determined by the majority. Redundant execution and dealing with non-determinism in order are the focus of this thesis, which we discuss in subsequent chapters.

3.2 Non-Determinism in Value

While Replicant can match and externalize outputs that occur in different order, it will not externalize divergent output values. Depending on the application, some non-determinism in the code execution will result in divergent output values while others will not. As shown in our code example in Figure 2.2, non-deterministic accesses to the counter will result in different output values. On the other hand, non-deterministic ordering of calls to `printf` will only result in different ordering of outputs, which Replicant can resolve. Further, there are many events whose ordering generally will not have any effect on the ordering or value of outputs, such as calls to `malloc`, the heap allocator. Replicant provides the application developer with a determinism annotation that can be

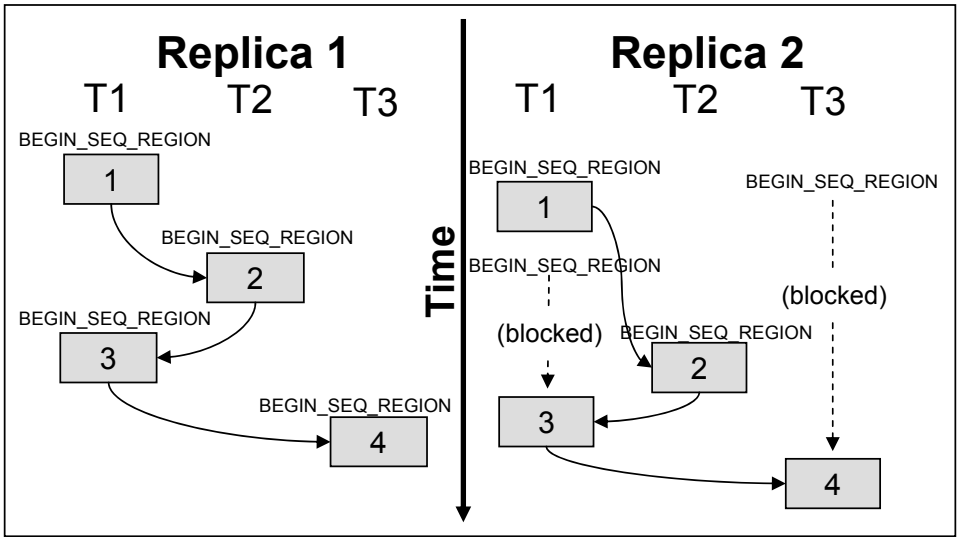

Figure 3.1: Determinism annotations. This example shows how determinism annotations (`BEGIN_SEQ_REGION` and `END_SEQ_REGION` – not shown for simplicity) are used to enforce the order in which sequential regions are crossed among replicas. The order is defined by threads in Replica 1 and Replicant forces threads in Replica 2 to cross the sequential regions in the same order.

used to remove the non-determinism and hence eliminate the resulting divergent outputs.

Replicant's determinism annotations are analogous to memory barrier instructions in relaxed memory consistency models [1] where, in the common case, memory accesses and modifications are not ordered unless explicitly specified by the application developer. Enforcing the order of memory operations incurs some performance penalty, but relaxing them allows aggressive compiler optimizations as well as hardware optimizations to be leveraged by the application. Like memory barrier instructions, determinism annotations are used to suspend the relaxations in Replicant that allows replica threads to execute independently. When specified by the application developer, Replicant enforces deterministic ordering of thread execution across replicas. However, this operation has a performance cost since the execution of threads in replicas are serialized.

Determinism annotations are used to remove the non-determinism in value, which arises from different ordering of events across replicas, such as inter-thread communication events, that are invisible to Replicant but affect externally visible outputs. The

application developer can explicitly identify the events, whose ordering must be deterministically replicated, by annotating the code that executes those events with a determinism annotation that defines the bounds of a *sequential region*. Replicant ensures that the order in which threads enter and exit a sequential region is the same across all replicas. As illustrated in Figure 3.1, determinism annotations (`BEGIN_SEQ_REGION` and `END_SEQ_REGION`) are used to annotate the application code. The `END_SEQ_REGION` annotation, which is not shown on the diagram for simplicity, appears after each grey box. The order in which sequential regions should be crossed is defined by threads in Replica 1 and Replicant forces the corresponding threads in Replica 2 to cross the sequential regions in the same order. This concept is similar to the shared object abstraction introduced by LeBlanc et al. [19]. In short, the determinism annotations make the invisible inter-thread communication deterministic such that the contents of the outputs they affect are deterministic.

In the example given in Figure 2.2, the developer should place a sequential region around the critical section bounded by the `lock` and `unlock` operations at lines 6 and 9 respectively. This ensures that corresponding threads in each replica pass through this region in the same order and update the `counter` variable in the same order. Thus, the threads will produce the same output, even though they may still print out their results in a different order. Replicant can then match the out of order outputs. We have found that a good heuristic for using sequential regions is to place them around locks that protect shared variables so that communication through shared memory is performed in the same order across all replicas.

3.3 Summary

We have given an overview of Replicant and showed how applications are redundantly executed. Replicant can handle both non-determinism in order and non-determinism

in value that multi-threaded applications exhibit. Non-determinism in order is handled by matching up the outputs of the different threads in an application. For applications that have divergent outputs, only the non-deterministic code section affecting the output values must be annotated with determinism annotations to eliminate the divergence. Replicant then enforces the order in which the sequential regions are executed across replicas, which suppresses the non-determinism in value at some performance cost.

The topic of determinism annotations is that another thesis that was developed in parallel and its design and implementation are discussed in [27, 26]. Although determinism annotations are a crucial component that allows Replicant to handle any multi-threaded application, it is orthogonal to this thesis. The remainder of this thesis describes redundant execution and the mechanisms to deal with non-determinism in order. The resulting system is then evaluated with applications that do not exhibit non-determinism in value (e.g. SPLASH-2 benchmarks [44]), and hence do not need determinism annotations. However, it is worth keeping in mind the guarantees that determinism annotations provide, i.e. enforce the order in which sequential regions are crossed across all replicas such that events invisible to the kernel are made deterministic.

Chapter 4

System Architecture

This chapter describes the architecture of the Replicant. Conceptually, Replicant implements an input replicating and an output matching architecture that is tolerant to re-ordering of events. A unique aspect of Replicant is that it permits replicas to execute independently and diverge in their behavior for performance gains. However, only outputs that a majority of replicas have confirmed are externalized outside of the redundant execution system. In the next sections, we outline the design considerations for Replicant and elaborate on how redundant execution is performed and how Replicant deals with non-determinism in order.

4.1 Design Considerations

Replicant allows replicas to execute independently and tolerates non-determinism for performance reasons, as long as it does not affect externally visible output. As discussed in Section 2.3, existing techniques for enforcing a completely deterministic execution of replicas have very high performance costs and thus are not practical.

Like other redundant execution systems [10, 45], Replicant manages the inputs and outputs of the replicas at the system call interface for two reasons. First, we would like Replicant to be isolated from the applications for security, as a compromised application

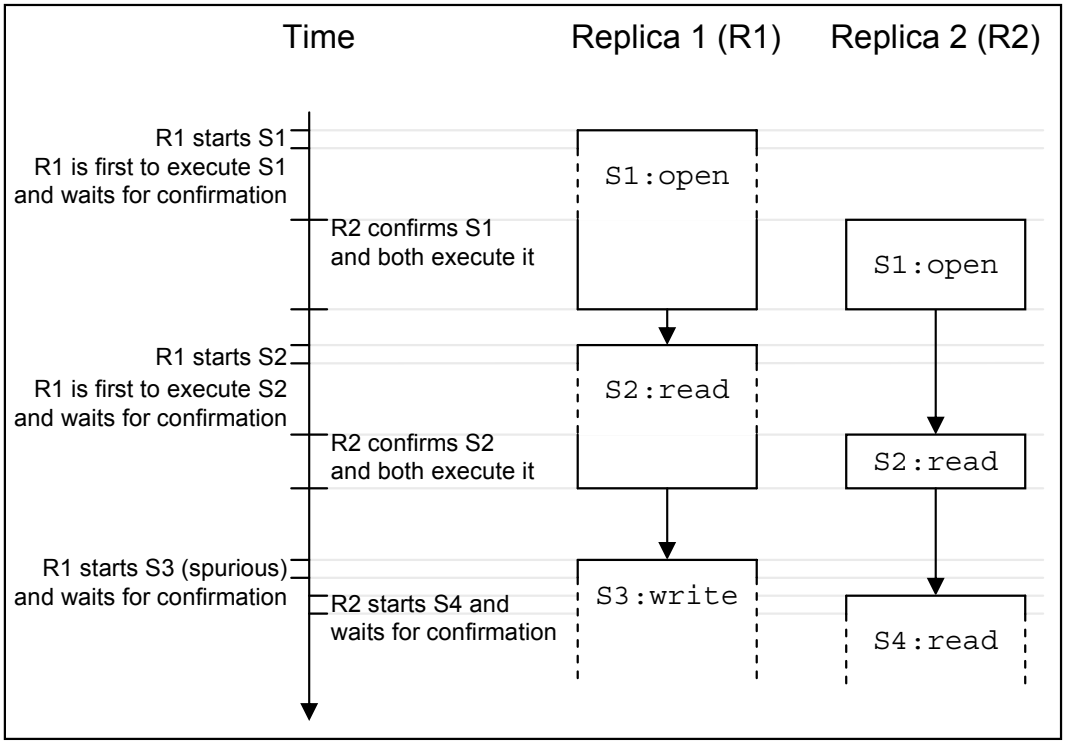

Figure 4.1: Deadlock scenario for a simple rendez-vous approach in a 2-replica system, where one of the replicas (R1) makes a spurious system call (S3) which R2 does not make. By deadlock, we mean that both replicas cannot make forward progress.

would be able to corrupt and disable mechanisms implemented in user-space, e.g. in a shared library. Replicant thus has to be in kernel-space for isolation and since the system call interface is the communicating interface between user-space applications and the kernel, it is an intuitive interface for Replicant to interact with applications. Secondly, it offers more contextual information that makes it significantly easier to replicate inputs to replicas, buffer unconfirmed outputs until confirmed and discard unconfirmed system calls, if and when needed.

A simple approach for confirming system calls would be to stall execution or *rendez-vous* on every system call until the majority of replicas confirm it and then proceed with their execution, like N-Variant [10]. A drawback of this approach is that it forces several context switches on each system call, which is likely to degrade performance if

the application makes many system calls.

While this approach works well for single-threaded applications, multi-threaded applications would cause any system that uses a rendez-vous mechanism to *deadlock*, i.e. replicas cannot make any forward progress, due to non-determinism. To illustrate, consider a scenario where non-determinism in an application causes one of its threads to make a *spurious system call*, which is a system call made by a minority of the replicas, as illustrated in Figure 4.1. Spurious system calls could be benign or malicious and thus cannot and should not be confirmed respectively. The example shows a 2-replica system, for simplicity, that confirms system calls in the above-mentioned rendez-vous fashion. Up until system call S1, all system calls made by the two replicas have been successfully matched and confirmed. Replica 1 (R1) is then the first to execute system call S1 and waits for Replica 2 (R2) to confirm it. When Replica 2 finally reaches system call S1, it confirms the system call and they both execute it and proceed. This is repeated for system call S2. However, if the next system call that Replica 1 makes is spurious (S3), i.e. Replica 1 executes system call S3 before it executes S4 while Replica 2 executes S4 next, then both Replica 1 and Replica 2 will wait for each other at different system call rendez-vous points and hence the system deadlocks. The same scenario can be applied to the re-ordering of system calls. Replicant solves this by allowing replicas to execute independently and confirms system calls by buffering them and matching them up.

This benign scenario arises in Apache where the first worker thread in a replica to execute the `libc` function `localtime_r` will read the current timezone from the OS and this is cached by `libc` to be re-used on subsequent calls. This operation may be performed by an arbitrary thread in each replica. As a result, the system calls associated with this operation will not match. Although this can be made deterministic across threads by using determinism annotations, we note that this non-determinism does not cause output divergence and thus can be omitted. Moreover, determinism annotations serialize the execution of the block of code annotated by a sequential region to make

the execution order deterministic across replicas and has a performance cost. As the number of determinism annotations inserted in an application increases, Replicant will unnecessarily incur an increasingly large performance penalty.

To illustrate how Replicant will incur a performance penalty with determinism annotations, let us reconsider the code example given in Figure 2.2. As previously mentioned, the threads in Replica 1 execute the critical section in the order (1, 3, 2) by thread ID while in Replica 2, the threads execute in order (2, 3, 1). In order to instruct Replicant to enforce deterministic execution of the critical section across replicas, lines 6 to 9 of the code is annotated with determinism annotations to identify a sequential region. Since Thread 1 in Replica 1 is the first to execute the critical section, whenever Threads 2 and 3 in Replica 2 try to execute the critical section, they will be stalled until Thread 1 in Replica 2 has finished executing the critical section. The stall time of Threads 2 and 3 translates into a loss of concurrency and hence a performance penalty. We observed that the performance penalty is proportional to the number of threads and the number of sequential regions in each replica, thus implying that the number of sequential regions should be minimized for performance.

Another example as to why a rendez-vous approach is not suited for multi-threaded applications is demonstrated by the non-determinism in `libc` itself. To illustrate, consider 2 threads in each replica that print out a string. Assume Thread 1 in Replica 1 and Thread 2 in Replica 2 invoke `printf` first, as shown in Figure 4.2. Since `libc` functions are thread safe and use locks to ensure mutual exclusion, the `printf` function will acquire the lock protecting the output stream before flushing the outputs in a `write` system call. Therefore, when Thread 1 in Replica 1 invokes `printf`, it will grab the `printf` lock in `libc` and then issue a `write` system call. This `write` will be stalled waiting for confirmation from Thread 1 in Replica 2 (Arrow #1). Similarly, since Thread 2 in Replica 2 invokes `printf` first, it will acquire the `printf` lock in `libc` and invoke the `write` system call. This `write` will also be stalled, waiting for confirmation from Thread 2 in Replica

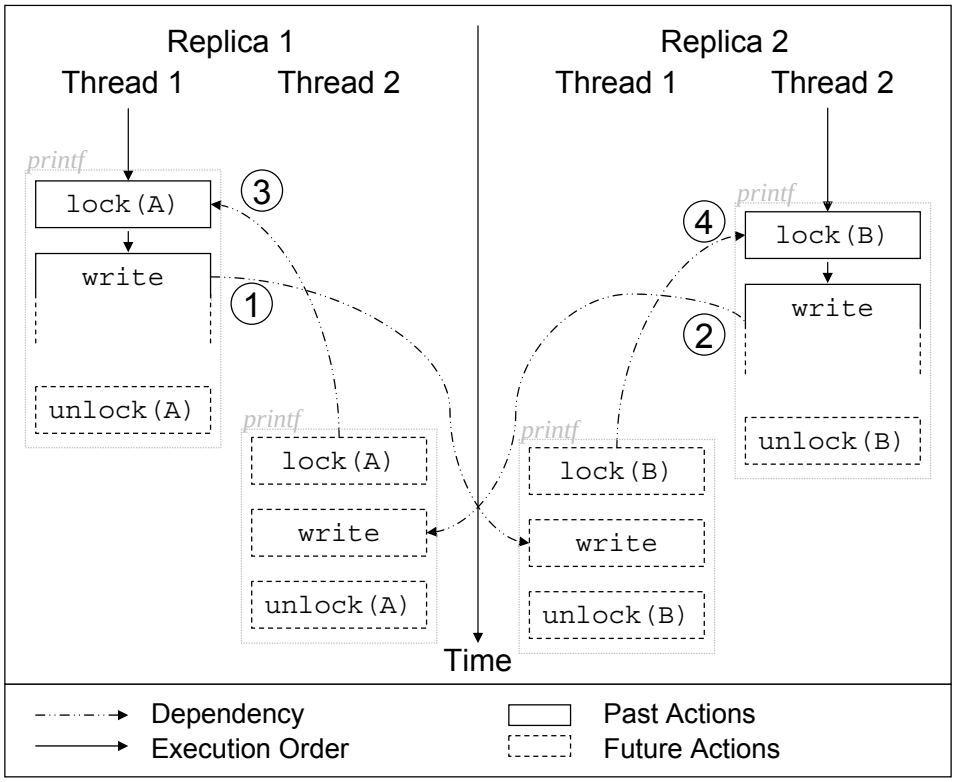

Figure 4.2: Deadlock scenario 2. This figure illustrates another problem with the rendez-vous approach for multi-threaded applications. Here the non-determinism in `libc` prevents forward progress.

1 (Arrow #2). However, Thread 2 in Replica 1 will never confirm the `write` of Thread 2 in Replica 2 because it cannot acquire the `printf` lock being held by Thread 1 in Replica 1 (Arrow #3). The same dependency occurs for Thread 1 in Replica 2 (Arrow #4), thus creating a cycle and preventing further progress of the application. The rendez-vous approach introduces an undesirable dependency between threads, even though they are unrelated (e.g. Thread 1 in Replica 1 and Thread 2 in Replica 2). Although this can be solved by annotating all locks in `libc` with determinism annotations, this is a daunting task and is likely to severely degrade performance. Replicant eliminates this dependency (Arrow #1 and #2) by allowing replicas to execute independently in an OS sandbox and buffering and matching outputs as they are confirmed.

In summary, the architecture we propose for Replicant is a kernel-based system that

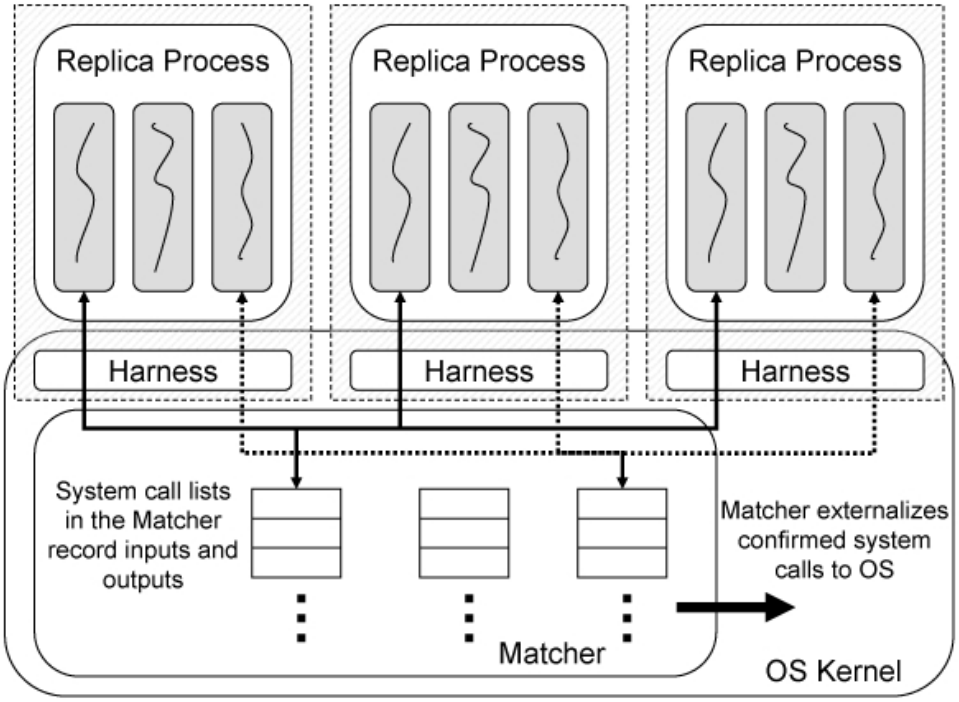

Figure 4.3: The Replicant architecture.

interacts with applications at the system call interface. Replicant allows replicas to execute independently for performance and tolerates non-determinism in order as well as spurious system calls. Moreover, Replicant will use determinism annotations only when needed to handle non-determinism in value, at some performance cost.

4.2 Replicant Architecture

Based on the design considerations we outlined earlier, we elaborate on the architecture of Replicant, shown in Figure 4.3, which consists of two main components: the *harness* and the *matcher*.

4.2.1 Harness

A key requirement for supporting independent execution of replicas is that each replica needs to execute in an isolated and initially identical environment. Replicant places each

replica in an OS sandbox, called a harness, which is composed of a copy-on-write (COW) file system as well as other private copies of OS state. In addition, each replica has its own address space, which is randomized to detect memory corruption. The harness state is visible only to the replica that owns it, i.e. all threads in the same replica share the harness and can communicate through it, if needed, without affecting other replicas. The harness is kept up-to-date by applying the outputs and effects of all system calls a replica makes, even if they are not confirmed. 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.

A nice side effect of the COW file system is simplicity. Since replicas can act on their environment independently, they can create new files or delete existing files in their harness without affecting the externally visible environment (e.g. the OS file system) or other replicas. Thus, Replicant does not need to track file system dependencies, e.g. if a replica deletes a file but the delete operation has not yet been confirmed, then that replica should no longer have access to the file but the other *trailing replicas* should still be able to access the file until they confirm its deletion. Similarly, replicas can also write unconfirmed data to a file in their harness and then read back the modifications without having to worry about *read after write* dependencies in the OS file, which lives outside the harness. Our initial implementation of Replicant included a file system dependency tracking mechanism but the level of complexity it introduced was not justified. Moreover, it made a copy of each file that was opened such that modifications to the file could be made in isolation until confirmed. This copying penalty was very expensive for large files.

Although each replica executes independently of the others, each thread within a replica keeps information about its *peers*. Each thread in a replica is associated at birth with exactly one thread in every other replica, and this group of threads forms a *peer group* across all replicas as shown in Figure 4.3. Threads in a peer group are all created by the same thread creation event. In the contrived example given in Figure 2.2, threads

with the same thread ID form a peer group across the replicas and share the same system call list. Within a peer group, the thread that executes a system call first is called the *leading peer* thread while the rest are called *trailing peer* threads. It is worthwhile to note that due to thread independence and different relative rates of execution, the currently leading peer may become one of the trailing peers and vice versa.

4.2.2 Matcher

Replicant also includes a matcher component for each set of replicas. In this section, we introduce the matcher and classify the system calls handled by the matcher into 4 categories, each of which is supported by an example. In particular, these examples illustrate how the matcher identifies equivalent system calls, replicates inputs, buffers outputs and does system call matching in the presence of spurious system calls.

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, and then match up the system calls on a per-peer group basis. Threads in a peer group share a system call list in the matcher as shown in Figure 4.3. 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.

As summarized in Table 4.1, Replicant splits the handling of each system call invoked by the replicas between the replica's harness and the matcher depending on whether the system call requires inputs or creates outputs, and whether those inputs and outputs are external or non-external. We illustrate how each of these classes of system calls is handled using examples in the context of a 2-replica system for simplicity, but this can be

	Does not Require External Input	**Requires External Input**
Does not have Externally Visible Output	Execute within harness.	**If system call matches a list entry:** Replay recorded inputs to the harness. **If system call does not match any list entries:** Execute system call on OS and record system call in the list.
Has Externally Visible Output	Execute system call within harness and buffer the output in the system call list until confirmed.	Extrapolate the result based on current OS state and return it to the harness. Defer execution on OS until the system call is confirmed.

Table 4.1: Replicant's handling of system calls from replicas.

generalized to an n-replica system. In each example, the order in which the peer threads make the system calls is shown with respect to a timeline. Each system call is tagged by the replica ID and the thread ID, both of which are used during matching. The scenario is also represented on a schematic, with numbered arrows illustrating the flow of actions that Replicant undertakes. In a 2-replica system, the leading peer invokes a system call first and it is confirmed when the trailing peer invokes the same system call instance.

Scenario 1. If the system call transfers inputs to the application, Replicant considers whether the input is generated from state external to the harness or not. System calls that require non-external inputs are executed on the harness, which emulates the underlying OS, and do not need the aid of the matcher (top-left quadrant of Table 4.1). Consider the example of a non-external input from Figure 4.4, where the application makes a `read` from a file stored on the COW file system in the harness (Step 1). The system call is executed on the harness and the outputs are returned to the application (Step 2). When the trailing peer makes the same instance of the system call (Step 3), the system call is executed on its harness as well (Step 4), without the intervention of the matcher. Other examples are `getpid`, `brk`, `mmap`, etc.

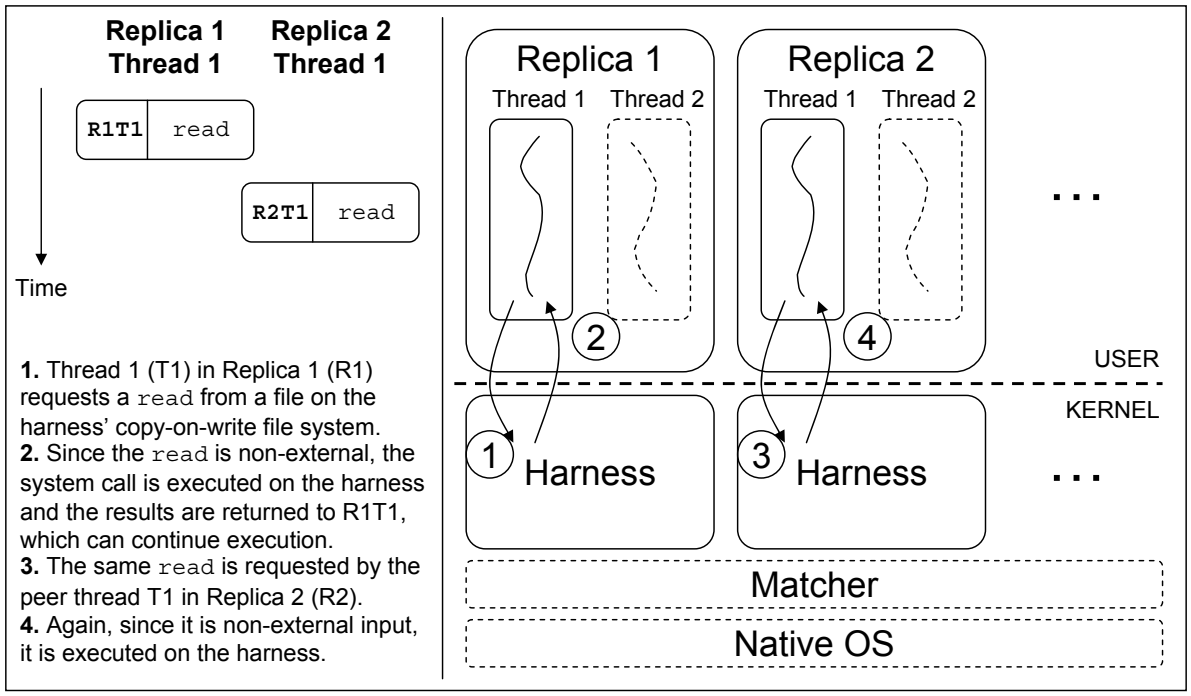

Figure 4.4: Scenario 1 (top-left quadrant of Table 4.1). This scenario illustrates Replicant's handling of system calls that do not have externally visible outputs and do not require external inputs. They are executed without the help of the matcher.

Scenario 2. On the other hand, external inputs (top-right quadrant of Table 4.1) are those that cannot be handled by the harness alone, such as a `read` from the network or from a device, and must be forwarded to the matcher as illustrated in Figure 4.5. This is necessary since consecutive reads from the network (e.g. one from each replica) will not return the same input data to the replicas, which will cause them to diverge. Therefore, the matcher records any external inputs and replays it to the other replicas, when they invoke the same system call instance, thus ensuring that the replicas get the same inputs.

As shown in the example, when the matcher receives a system call that requires external input (Step 1), it first picks the system call list corresponding to the correct peer group, i.e. `R1T1|R2T1` in Figure 4.5. It then checks the system call list to see if another thread in its peer group has already made a matching system call. The matcher determines that two system calls match if the name and arguments of the system calls

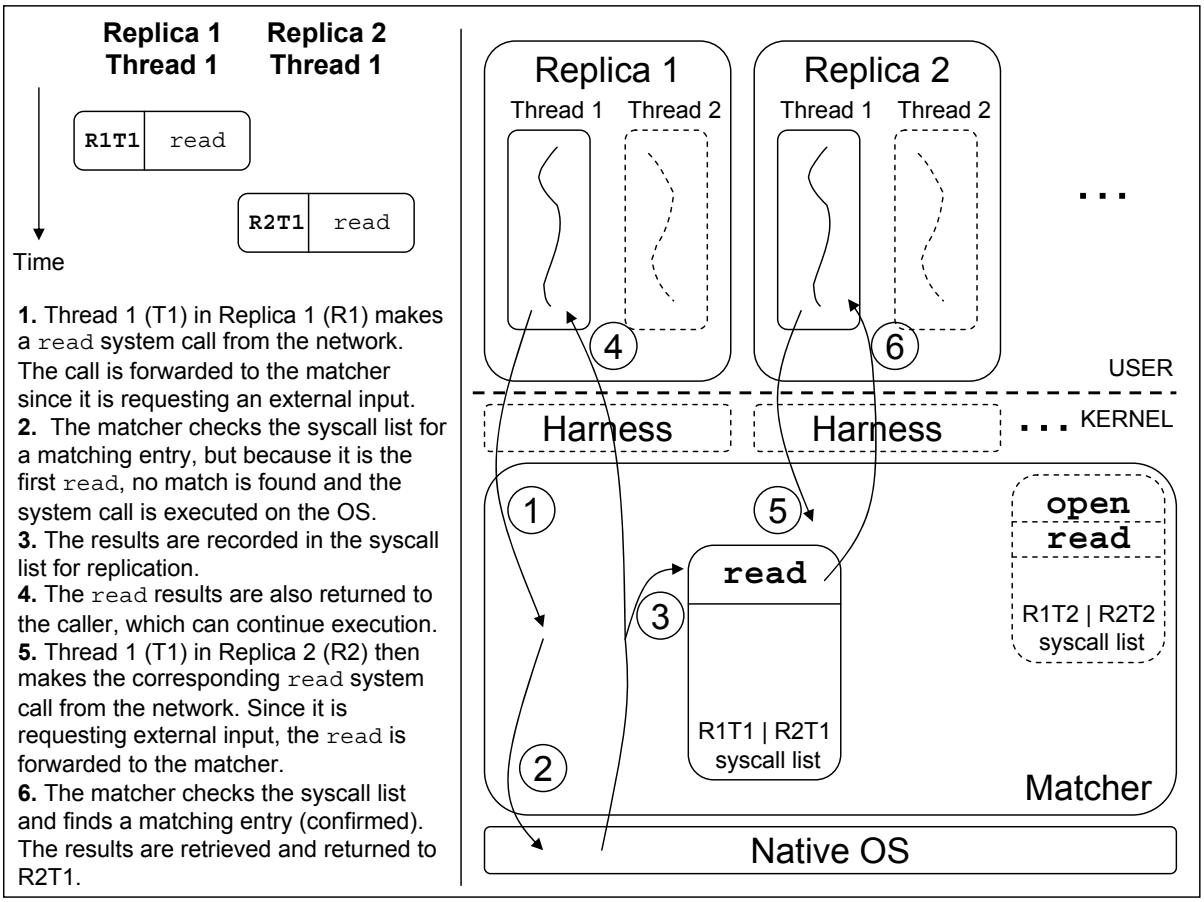

Figure 4.5: Scenario 2 (top-right quadrant of Table 4.1). This scenario illustrates how the matcher handles system calls that do not have externally visible outputs but require external inputs.

are the same. When trying to match system calls, the matcher searches the entire list for a matching system call, thus allowing the matcher to tolerate system calls that occur out of order among peers. If there is no match, the matcher executes the system call on the OS to fetch the external inputs (Step 2). It then records the system call arguments and its results in the system call list of the thread's peer group for replication (Step 3) and returns the external inputs to the caller thread (Step 4). When the trailing replica makes the corresponding system call (Step 5), the matcher will find a matching system call and return the same result that the previous replica received (Step 6). System call entries are removed from the list when all threads in the peer group have matched the system

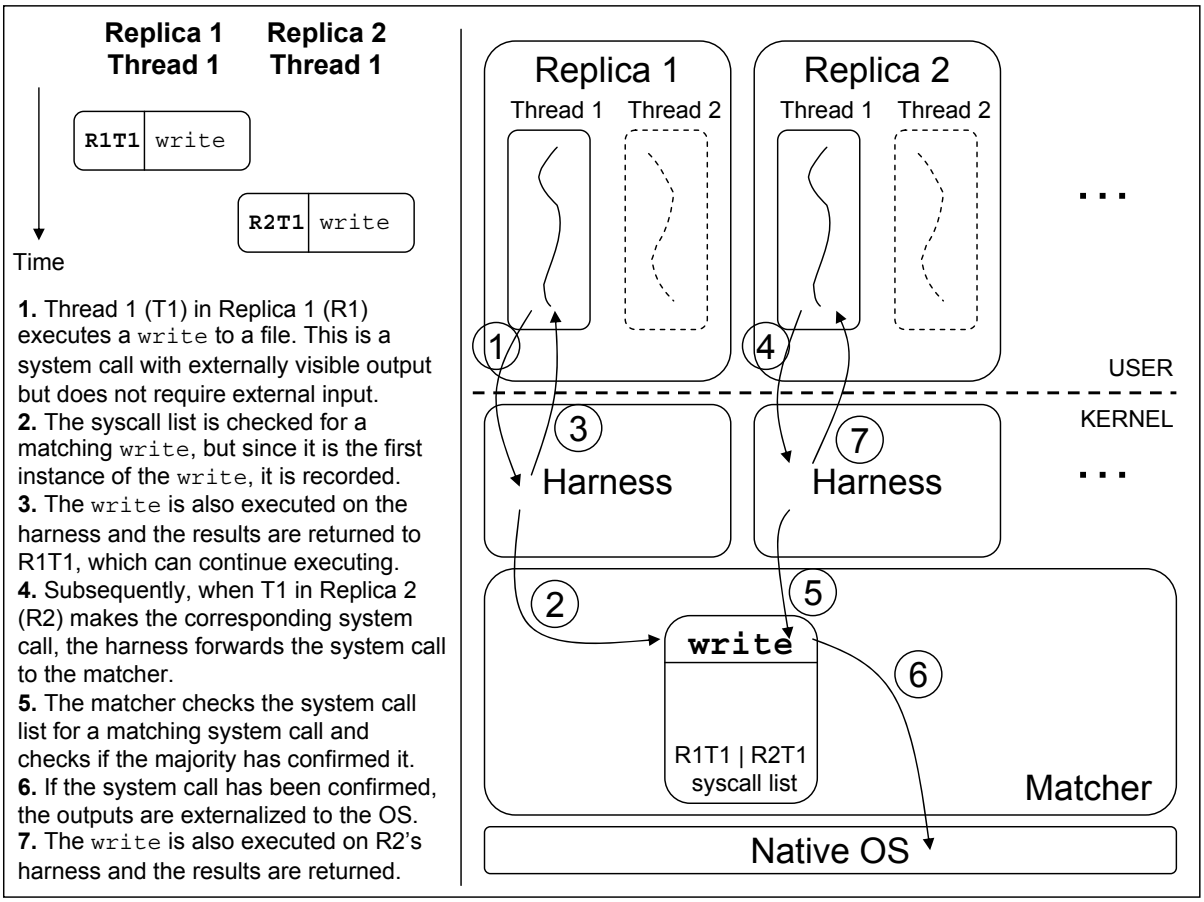

Figure 4.6: Scenario 3 (bottom-left quadrant of Table 4.1). This scenario illustrates how the matcher handles system calls that have externally visible outputs but do not require external inputs.

call. Other examples are `gettimeofday`, `read` from a device such as `/dev/random`, etc.

Scenario 3. Replicant also takes into account whether the system call has outputs that are only visible to the replica itself, or whether the system call has outputs that will be externally visible. Recall that externally visible outputs must be confirmed before they are externalized. System calls with no externally visible outputs, such as a `write` to a pipe between two threads in a replica, are only run on the harness state (Scenario 1). However, system calls with externally visible outputs (bottom-left quadrant of Table 4.1), such as `unlink` (to delete a file) or a `write` to a file, are run on the harness state and then forwarded to the matcher. The system call has to be run on the harness to keep

the harness state up-to-date, such that the application does not realize that its outputs are being buffered before they are externalized on the OS by the matcher.

Consider the example of a `write` to a file (Step 1) in Figure 4.6. The outputs of the `write` are buffered by the matcher in the system call lists (Step 2) until they are confirmed, at which point they are externalized by the matcher. The `write` is also run on the harness (Step 3) and the results of the system call are returned to the replica, which can proceed with its execution. Thus, any subsequent `read` from the file, by the same replica, to retrieve the previously written but unconfirmed data will succeed. When the `write` is confirmed by the trailing thread (Steps 4 and 5), the matcher externalizes the outputs by executing it directly on the OS (Step 6) – which will succeed unless there is a catastrophic failure of the disk. The trailing thread also executes the system call on its harness (Step 7) to update its state and the return value is compared with the results from Step 6 for consistency.

Scenario 4. Finally, a `write` to a socket is a system call with external outputs but also requires external inputs derived from the OS socket as opposed to the harness (bottom-right quadrant of Table 4.1). In this case, the external input refers to the return value (or error code) of the `write` when it is executed on the OS socket because there is no equivalent socket in the harness, as will become clear in Section 5.3.1. This scenario is illustrated in Figure 4.7. As before, the application makes a `write` to the network which is recorded in the system call list until confirmed (Steps 1 and 2). Since the system call cannot be executed on the OS until it is confirmed, Replicant extrapolates the input from the state of the OS socket at the time of the `write` and allows the replica to proceed (Step 3). When the `write` is confirmed by the trailing thread (Steps 4 and 5), the outputs are externalized (Step 6). The results are returned to the trailing replica (Step 7), but are also checked against the extrapolated results returned to the leading peer for consistency. Other system calls that require extrapolation are `writev`, `send`, `sendto`, `send_msg` which are all used to write to a socket, `sendfile64` which is used by

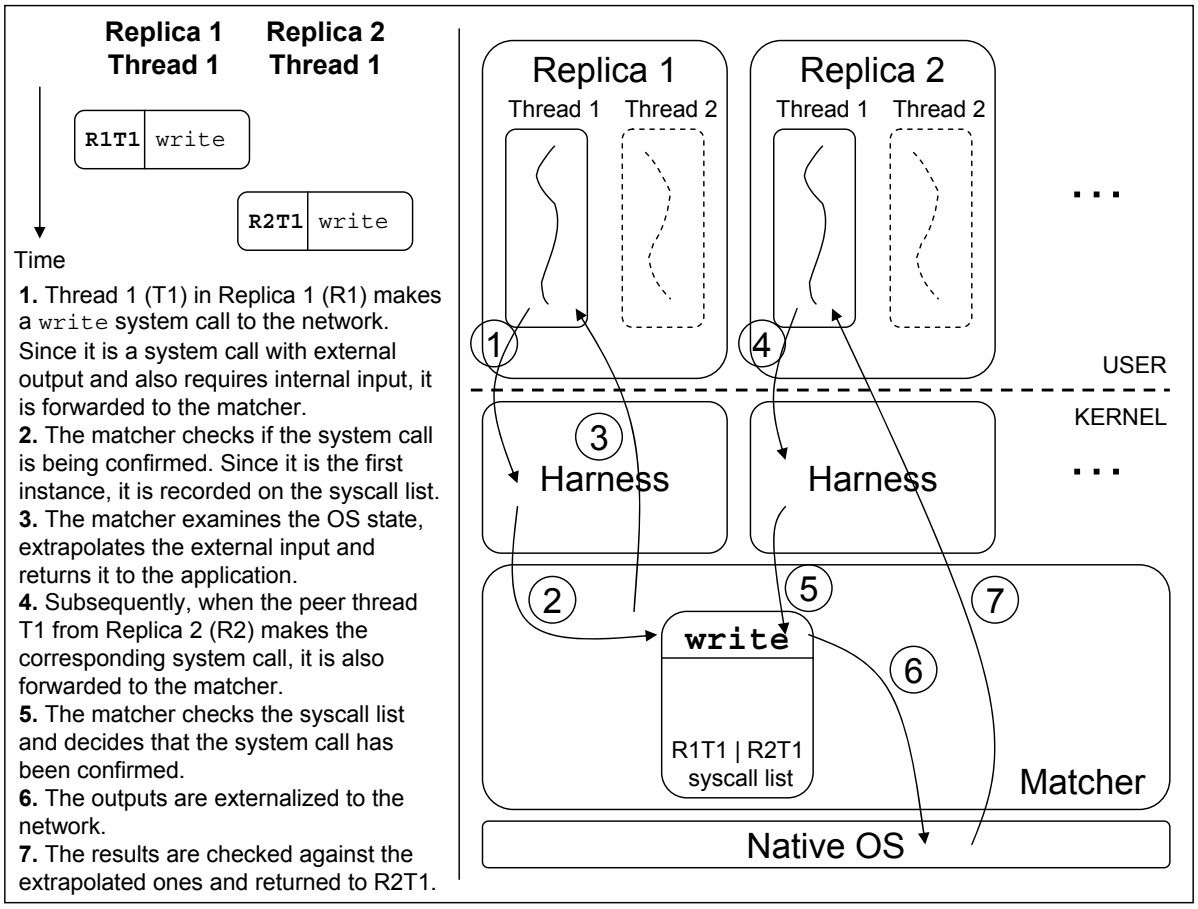

Figure 4.7: Scenario 4 (bottom-right quadrant of Table 4.1). This scenario illustrates how the matcher handles system calls that have externally visible outputs and require external inputs.

some applications to efficiently transfer data from a file to a socket and `shutdown` which is used to shutdown part of the duplex connection.

Extrapolation is done by making a few simple checks on the file descriptor, e.g. whether the specified file descriptor is a valid socket and whether the socket is connected, and returning success or the appropriate error code to the application. Unfortunately, extrapolation can return inconsistent results to the peers, if not done carefully. For example, this can occur if the leading peer extrapolates success but the remaining peers return failure because the remote host subsequently disconnected. Inconsistent return values can cause the replicas to diverge and therefore, it is only safe to extrapolate

if the paths taken by the peers do not diverge or converge after a short divergence, without loss of critical outputs. In cases where the inconsistent return value affects the code path (hence the system call sequence) and the replica does not converge again, Replicant provides a developer annotation `make_sync` to suppress extrapolation. When an annotated system call is invoked, Replicant blocks all replicas in a rendez-vous fashion until the system call is confirmed, thus ensuring that all replicas get a consistent and correct return value. The drawbacks of this approach is the loss of concurrency and the fact that a `make_sync` annotated system call cannot be spurious or the system will deadlock. This user-space annotation can easily be converted to a kernel mechanism, if needed. TightLip [45] performs a similar rendez-vous, using barriers and conditional variables, for system calls that modify kernel state.

System Call Matching. The matcher does system call matching for input replication or output confirmation as illustrated in the previous examples. System calls are matched both by the system call name and its arguments. For arguments that are set by the application but passed down to the kernel as addresses, the matcher checks the content of the buffers as opposed to the address values for better accuracy. Moreover, addresses cannot be compared since dynamic memory allocation is non-deterministic and the address space of each replica is randomized.

Consider the case where a replica makes 2 spurious (or malicious) `write` system calls – shown in striped boxes in Figure 4.8, a legitimate `write` and a `close`, all of which remain to be confirmed by the other replica. The prototype function of a `write` system call is as follows: `write(fd, buf, num_bytes)`, where `fd` is the file descriptor to write to, `buf` is a pointer to the buffer to be written and `num_bytes` is the length of the buffer. When the legitimate `write` system call is made by the second replica (Step 1), the matcher searches the corresponding system call list for a matching `write` by comparing the file descriptor number, the number of bytes to be written as well as the contents of the write buffer (Step 2). In this example, the matcher proceeds to the next entry in the system call list

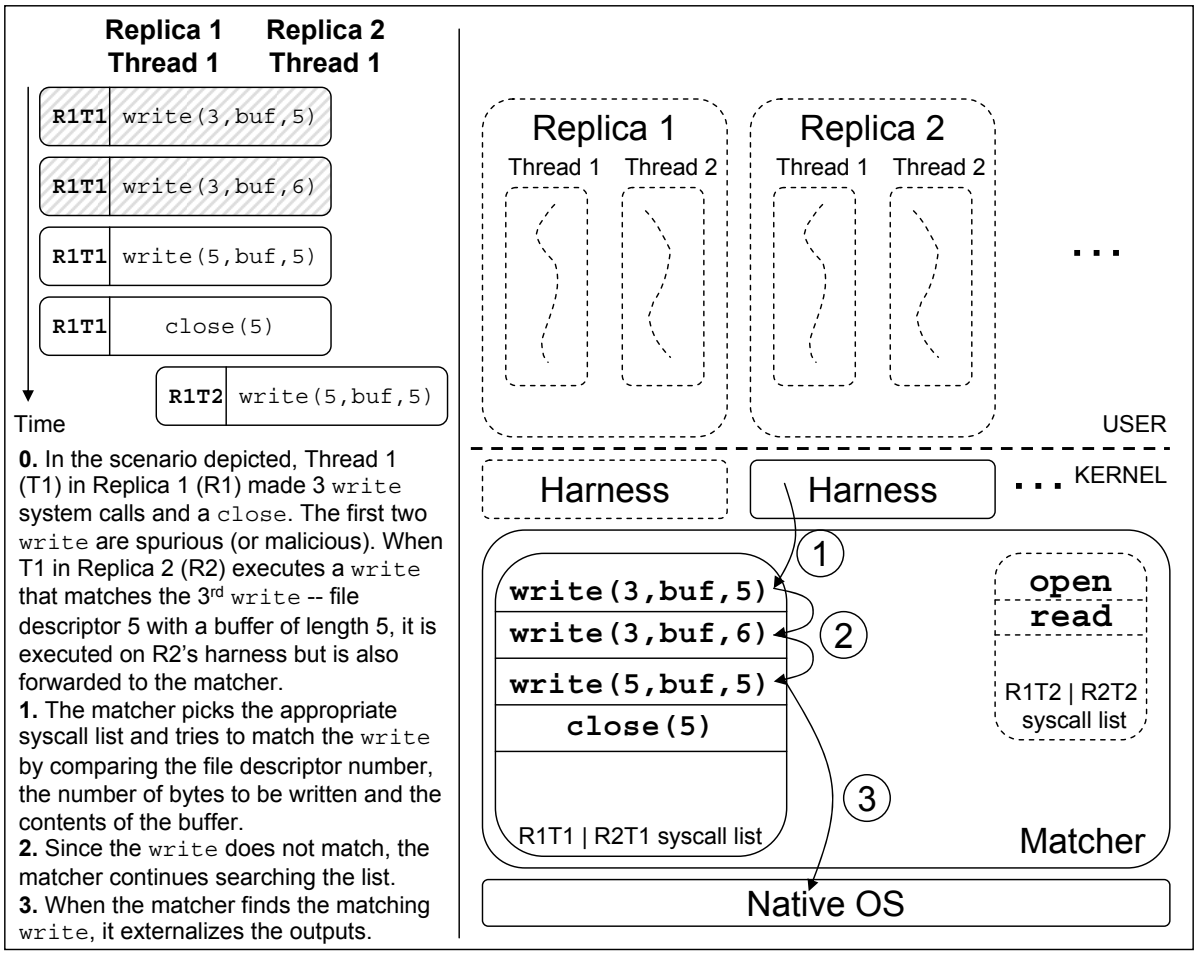

Figure 4.8: System call matching. This scenario illustrates how the matcher does system call matching in the face of spurious (or malicious) system calls by comparing all arguments in matching, including buffer contents. The spurious (or malicious) system calls are never matched and thus their outputs are never externalized.

after a simple check of the file descriptor numbers, i.e. the legitimate write is to `fd = 5` while the spurious ones are to `fd = 3`, which do not match. If the file descriptor numbers did match, the matcher would then try to match the number of bytes to be written and only if both `fd` and `num_bytes` were equal would the matcher perform the most expensive operation of matching the contents of the buffer using a byte-by-byte comparison. When a match is found, the `write` is confirmed and the output is externalized (Step 3). Note that the spurious (or malicious) `write` system calls are nonetheless executed on the harness, but are never made externally visible.

Because a thread restricts its search to its system call list, it will only match system calls with other threads in its peer group. One might be tempted to allow a thread to match system calls with any other thread in the other replicas. However, certain input system calls, such as `gettimeofday`, do not have arguments that are set by the application. As a result, the matcher cannot use the arguments to match the right system call if there are several instances available. By restricting threads to only match with other threads in the same peer group and taking into account that the system call lists maintain temporal order information, the matcher ensures that system calls always match with the appropriate system call instances.

4.2.3 Summary

In summary, the harness allows the replicas to execute independently for performance and replicates enough OS state such that the replicas are not aware that their outputs are being buffered.

The matching algorithm handles system calls based on their classification as external or non-external. It matches system calls from the same peer group and compares a system call's name and arguments. Matching system calls from the same peer group with the help of contextual information allows Replicant to deal with re-ordering of system calls among threads, while buffering outputs until they are confirmed solves the problem of spurious system calls. This also implies that the outputs of malicious system calls are never externalized unless a majority of replicas are compromised, thus preserving data integrity of the system.

Once a system call has been confirmed, its outputs are made externally visible, and thus outputs are externalized in the order in which they are confirmed by the replicas. The examples used to illustrate how the matcher works were in the context of a 2-replica system for simplicity. These mechanisms can be generalized to an n-replica system.

Chapter 5

Implementation

In this chapter we elaborate on the implementation of our Replicant prototype, which was implemented by modifying a standard Linux 2.6.16 kernel [38, 8]. The prototype consists of two components, namely the harness and the matcher, which we will discuss in turn. As a proof of concept, our prototype currently supports two replicas, but can be extended up to any number of replicas. We end this chapter by discussing some interesting aspects of implementing a Replicant system.

5.1 Harness

Each Replicant harness houses a single replica and supports independent execution. The file system namespace for each replica is directed to a COW image of the OS file system, which is implemented by the `dm-snapshot` module that is part of the Linux kernel. This allows a harnessed replica to open files in the harness that are not open or may not even exist on the underlying OS.

Linux's process descriptors were extended to include harness state, as shown in Figure 5.1. Each replica (hence each harness) has a unique replica number, `rep_id`, which is assigned at harness creation time. All threads within the same replica share the same `rep_id` and all subsequent processes or threads that are forked or cloned by a replica have

```
struct task_struct {
    ...

    /* HARNESS STATE */
    int rep_id;                                /* All threads in the same replica
                                                * have the same ID. Non-Replicant
                                                * threads have ID 0. */

    struct list_head *rep_peer_list;           /* list of all threads in peer grp */
    struct files_struct *rep_harness_files; /* harness FDT */
    ...
}
```

Figure 5.1: Harness state. The harness is implemented by a COW file system and harness state is kept in Linux's process descriptors.

the same `rep_id` as their parent. A `rep_id` of `0` denotes a process not running on top of Replicant and hence does not execute the instrumented system call handlers. Instead, non-Replicant threads or processes execute the unmodified system call handler code.

As mentioned previously, each thread in a replica is part of a peer group. The harness state includes a pointer to a list named `rep_peer_list`, which is shared among all threads in the same peer group. The peer list keeps track of all peers, from different replicas, that form the peer group and is initialized when the peer group is created.

The harness state also keeps a pointer to the harness' file descriptor table (FDT), `rep_harness_files`, which is shared among all threads in the same replica (assuming that `clone` is called with `CLONE_FILES` flag), but never among peers. The FDT keeps track of file descriptors owned by all threads sharing the same FDT, while the file descriptors are a reference to Linux's open file objects that represent opened resources and their associated state [8]. File descriptors that live in the harness FDT are backed by resources within the harness, e.g. files on the COW file system, where possible. External resources like sockets, which we discuss in Section 5.3.1, have no corresponding harness resource and thus their existence in the harness' FDT is merely a place holder that keeps track of the resource's state (e.g. socket state). Note that since each harness has its own COW file system, each requires its own FDT. The OS file system is accessed through another FDT,

the OS FDT, which is managed by the matcher. We elaborate on it in the next section.

A Replicant harness is created when the application process is created. Just as regular processes are created via a `fork` system call, replica processes are created via a new `rep_fork` system call, which will create two new harnessed replica processes and setup their relationships. In this way, each harness provides a replica with its own address space and its own private copy of the file system. `rep_fork` only needs to be called once to start replicating an application. Subsequent confirmed calls to `fork` or `clone` by the application will create two new replica processes. These new processes automatically inherit their parent's view of the file system, but other harness components, such as the FDT, may or may not be inherited depending on the options passed to the forking call. Similar to the default `fork` system call, the `rep_fork` system call provides the replicas with COW memory pages [8].

As discussed in Chapter 2, Replicant detects memory corruption by randomizing the address space layout between the two replicas. This is incorporated in Replicant by utilizing the ASLR facility that is part of the PaX/grsecurity Linux kernel patch [25]. PaX randomizes the brk-managed heap, the mmap-managed heap, the stack, the base address at which libraries are loaded and optionally the base of the executable itself.

5.2 Matcher

The matcher is implemented by extending Linux's process descriptors to include the matcher's state (Figure 5.2) and by modifying the Linux's system call handlers (Figure 5.5). We first describe the matcher's state and then elaborate on the matcher's mechanisms, which adhere to the design outlined in Chapter 4. We end this section by describing enhancements to the matcher. Note that resources that are shared are also protected by spinlocks, which are omitted here for simplicity.

```
struct task_struct {
    ...

    /* MATCHER STATE */
    int rep_trailing_peer;                  /* leading or trailing peer flag */
    struct files_struct *rep_os_files;      /* OS FDT */

    struct list_head *rep_syscall_list;     /* per-peer group system call list */
    int rep_syscall_count;                  /* Number of pending syscalls */
    struct list_head *rep_search_start;     /* Pointer to next item where
                                             * searching should start */
    void *rep_current_elem;                 /* Current syscall being processed */
    wait_queue_head_t *rep_elem_done_wq;    /* Waitqueue for syscalls in
                                             * progress (e.g. blocking calls) */
    ...
}
```

Figure 5.2: Matcher state. The matcher's state is incorporated in Linux's process descriptors.

5.2.1 Matcher State

The matcher state, shown in Figure 5.2, includes a per-peer flag, `rep_trailing_peer`, which indicates whether the current peer is leading or trailing (note that this state can change back and forth during a peer's execution). It also includes a pointer to the system call list, `rep_syscall_list`, which is shared among all peers within the same peer group. The system call list is used by the matcher for replicating inputs and buffering outputs, as described earlier. When outputs are confirmed, the matcher uses the OS FDT, denoted by `rep_os_files`, to access the external resources and externalize the outputs. The file descriptors that live in the OS FDT are backed by OS file system inodes that includes external resources such as sockets.

The OS FDT is also used to simplify system call matching by restricting how file descriptors are assigned to resources. Normally, Linux assigns file descriptors to resources based on the order in which the resources are allocated. Since resources can be allocated in different orders among the replicas, the mapping between file descriptors and resources will be different in every replica, requiring the matcher to maintain a translation table for each pair of replicas. Instead, Replicant creates a unified file descriptor namespace

and ensures that all threads across replicas use the same file descriptor for a particular resource instance, regardless of the order in which resources are allocated. The OS FDT tracks the allocation status of file descriptors among all replicas. Once a file descriptor is allocated by one replica, it can only be assigned to threads in the same peer group that are accessing the same resource in other replicas. The descriptor cannot be reused for a different resource or by a different peer group until all threads in the current peer group have closed and released the file descriptor.

Consider an example of two replicas, each having two threads of execution. Thread 1 opens file "`foo.txt`", while Thread 2 opens file "`bar.txt`". If in Replica 1, Thread 1 invokes the `open` system call before Thread 2, while the opposite occurs in Replica 2, then on a vanilla Linux system, the same resource will have different file descriptors assigned to it in each replica. However in Replicant, regardless of the order, the same file descriptor is always assigned to the same resource instance.

A consequence of the unified file descriptor namespace is that the replicated application use more file descriptors than what the unmodified application would. This becomes more pronounced as the number of system calls on a system call list gets larger, a number we call the *system call distance*, because the file descriptors used and released by the leading peer are not deallocated until they are confirmed and released by the trailing peer. While the very nature of this side effect is benign, the implication is that the applications cannot assume anything from the file descriptor allocation mechanism, which a well designed application should not be doing anyway. For example, even if a single-threaded application closes file descriptor 4 and opens a file right away, it cannot expect to get the same file descriptor 4 back.

The matcher state also consists of a pointer to the current element of the system call list that is being processed (`rep_current_elem`), a waitqueue (`rep_elem_done_wq`) that we discuss in the next section, a pointer to the next element in the system call list where searching should begin, denoted by `rep_search_start`, and a pending system

```
typedef struct rep_list_elem {
    struct list_head list;
    pid_t pid;              /* thread ID of enqueuing task */
    int rep_id;             /* leading replica identifier */
    unsigned int sysnum;    /* system call number */
    void *args;             /* system call arguments */
    void *res;              /* system call return value and recorded inputs */
    atomic_t done;          /* indicates if the system call has completed  */
} rep_list_elem_t;
```

Figure 5.3: System call list element. This data structure keeps information about the system call and its initial caller.

call counter, `rep_syscall_count`. The last two are matcher enhancements that will be discussed in Section 5.2.3, after the system call handler modifications are described.

5.2.2 System Call Handler Modifications

While each system call requires modifications specific to its semantics, Figure 5.5 presents a general description of how each system call is modified. `generic_syscall` depicts the system call handler code and `search_for_matching_syscall` is a helper function that does the system call matching. When the matcher attempts to match a system call, it looks up the peer group's system call list and skips those entries that have the same replica ID (`rep_id`) as itself. If an entry matches on the system call number, the matcher invokes system call specific compare functions that will compare the arguments of the current system call against those recorded in the system call list entry, to ensure that the same instance is being matched. All system call specific functions are function pointers kept in a table that is indexed by system call number. As previously mentioned in Section 4.2.2, for system calls that lack context information in their arguments, the matcher relies on temporal order information stored implicitly in the peer group's system call list.

Each system call recorded in the system call list is of type `rep_list_elem_t` as shown in Figure 5.3. It records information about the caller (leading peer) such as its `pid` and `rep_id`. It also records system call information such as the system call number and its

arguments, which are used for comparison. When the leading peer completes a system call, it records the return value as well as external input, if needed, and sets the `done` flag. The arguments and results recording routines create system call specific data structures to buffer the information. For example, in addition to the file descriptor number and the number of bytes to be written, the contents of output buffers of a `write` are recorded in a kernel buffer and used for comparison during system call confirmation. Similarly, input buffers returned to system calls such as `gettimeofday` are recorded for replication.

The `done` flag in each list element indicates whether that system call has completed. It is particularly useful when the leading peer makes a blocking system call such as `accept`. If the trailing peer tries to confirm that system call while the leading peer has not yet returned, the trailing peer should wait for the leading peer's results as opposed to executing it or reading the results (not yet available). This is achieved by checking the `done` flag of a matching system call. If the flag is not set, the trailing peer then waits on the waitqueue denoted by `rep_elem_done_wq` in Figure 5.2. Upon completion of the system call, the leading peer checks this waitqueue and wakes up any waiting peers.

Finally, for system calls that require extrapolation, e.g. writing to a socket (Chapter 4), Replicant currently performs a few simple checks such as checking whether the specified file descriptor is valid and whether it is a socket (inode mode). A more complete and accurate solution would be dependent on the system call being invoked and resource type. For example, on a `send`, Replicant could check for all possible error conditions such as ensuring that the OS socket is connected, that the OS socket can be written to, that no invalid arguments are being passed by the application, and more. However, it will not always be able to predict the exact outcome of the `send` action on the OS socket. For example, the client could disconnect when the trailing peer is performing the confirmed `send` action. In this case, we rely on the application to handle the divergence and later converge, as explained in Chapter 4. Otherwise, extrapolation must be suspended.

As mentioned in Chapter 4, Replicant provides the application developer with an

annotation, called `make_sync`, that suspends extrapolation and stalls execution of the current peer until the annotated system call is confirmed. `make_sync` is implemented as a new system call, which informs Replicant that the next system call should be stalled until it is confirmed. This is done by placing the caller on a waitqueue until the system call is confirmed, at which point, both peers would perform the requested action and return consistent results.

5.2.3 Matcher Enhancements

The search start pointer, `rep_search_start`, mentioned earlier in Section 5.2.1 and shown in Figure 5.2, takes advantage of the fact that the list retains the order in which system calls were made by the leading peer to skip spurious system calls. It denotes the point at which the trailing peer starts its search for a match and is updated to point at the next entry in the list when a match is found, as shown in Figure 5.5. The next time the trailing peer makes a system call, it starts searching the list at that point and will wrap around to the beginning of the list if it reaches the end without finding a match. We have observed that the majority of system calls made are not spurious on real workloads; so the most likely match is the system call immediately after the one that was just matched.

The pending system call counter, `rep_syscall_count`, also shown in Figure 5.2 is a matcher performance optimization. We have found that the system call distance can be large for some workloads, which occurs when one replica is far ahead of the other in its execution. In addition, we also found that it is unnecessary for the leading peer to search the system call list if the list is populated only with system calls it has made. Thus, we allow the leading peer to proceed without searching the list if we can determine that it produced all the system calls in the list, i.e. if `rep_syscall_count` $= 0$. Only when the trailing peer enqueues a system call is the leading peer told to search the list (by incrementing the leading peer's `rep_syscall_count`). When a peer thread consumes a system call from the system call list, `rep_syscall_count` is decremented. Other optimizations

for speeding up the trailing peer path are also possible but not all of them have been implemented. As a simple example, applications use the `select` system call to sleep, for portability reasons. The leading peer will execute the `select` and sleep for the specified time, but in the trailing peer, we can skip this sleep and capitalize on this opportunity to catch up on the leading peer and thus reduce the system call distance.

Finally, system calls that have not been matched after a specified amount of time are removed from the list. As unmatched system calls age, the likelihood they will be matched becomes lower. As an example, consider the Apache scenario mentioned in Section 4.1, where only the first thread reads the timezone information from the OS. This operation may be performed by an arbitrary thread in each replica. As a result, the system calls associated with this operation will not match. Such spurious system calls do not affect correctness of Replicant's outputs, but consume extra memory – especially if large buffers are involved. In our prototype, we remove system calls from the system call list that have aged past a configurable threshold (10 minutes in our case). We also remove all system calls and deallocate the system call list when all threads in the peer group have exited.

5.3 Caveats

In this section, we discuss some interesting aspects of implementing a Replicant system on Linux. We identify cause of these problems and describe how Replicant solves them.

5.3.1 Files vs Sockets

In Linux, all resources are abstracted as files, which are referenced by file descriptors that reside in the FDT of the process that has opened the resources. This exports a clean abstraction that unifies pipes, regular files, sockets and character devices, and allows the kernel to use the same system call handlers for different resources, regardless

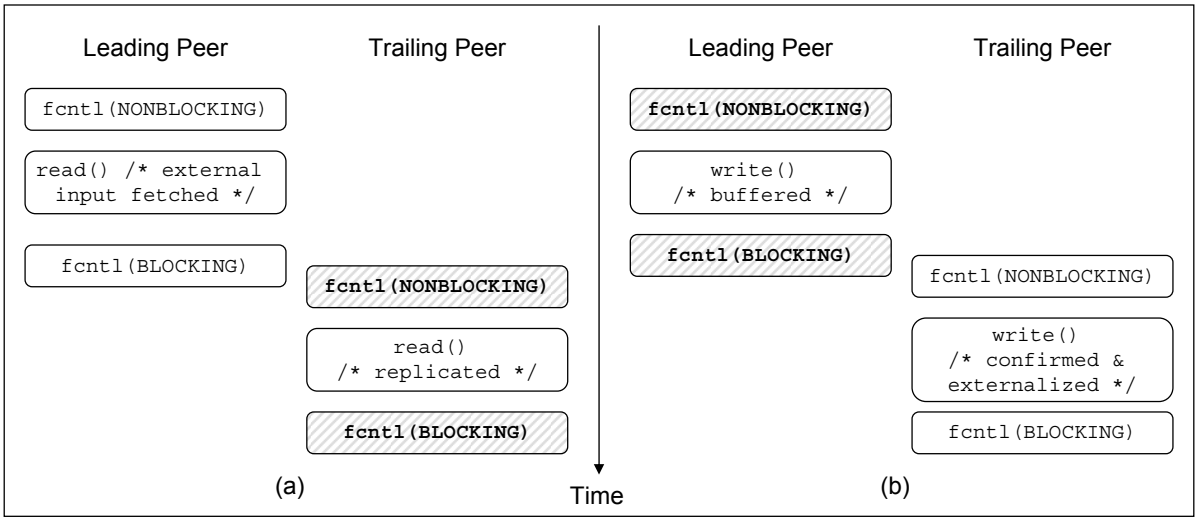

Figure 5.4: Socket state consistency. Scenario illustrating the problems associated with keeping socket state consistent across replicas. In (a) the `fcntl` side effects are applied to the OS socket on confirmation (shaded box), while in (b) they are applied during the first invocation.

of their type. Although this concept is attractive, it effectively complicates system call handling for Replicant since the resources backing a file descriptor might be providing non-external or external inputs and outputs. As a result, these system call handlers have to be modified to handle things differently depending on their semantics and the type of resource backing the file descriptor.

Pipes are easy to handle because they live only within a replica harness. Similarly, files are easily handled since our design provides a COW file system where inputs from files can be internally derived and outputs are applied to the harness, then externalized by the matcher when confirmed. However, sockets are associated with an entity external to the harness, typically a remote host. For several reasons, only one socket, the OS socket as opposed to the harness socket, is connected to the remote host. First, only one socket can be bound to the port requested by the application and this needs to be accessible to both the leading and trailing peers (through the shared OS FDT). Second, the replicas should appear as one application to the outside world and cannot create multiple connections to the same remote host. As a result, inputs cannot be derived

from and outputs cannot be applied to the file descriptor (socket) in the harness since it is not backed by the external resource. Therefore socket related system calls (except for those created from `socketpair` which are non-external sockets) require the help of the matcher to execute. Character devices are external resources that have similar behavior.

The unification of resources providing non-external and external input (files and sockets) under one abstraction makes system calls like `select` and `poll`, both of which operate on a set of file descriptors, tricky since the threads could be selecting or polling both files and sockets in a single call (i.e. requesting both non-external and external inputs simultaneously). To handle this, Replicant must split this call transparently and forward the resources that require external input (sockets) to the matcher and derive non-external inputs (files) from the harness. Here the inputs refer to the status of the sockets or files. The union of the result sets is then returned to the application.

Last but not least, dealing with resource state is tricky. We illustrate this through a comparison between socket state and regular file state. System calls that manipulate resource state, e.g. `fcntl` and `setsockopt`, are applied both to the harness resource and to the OS resource. Since regular files live in the harness and inputs are derived from the harness, any system call that manipulates file state has the desired effect when applied to the harness. It does not matter whether the side effects of `fcntl` are applied to the OS file at the time the leading thread makes the system call or when the trailing thread confirms it. However, this is not the case for sockets since they are external resources and exist only as place holders in the harness. As a result, inputs cannot be derived from the file descriptor in the harness and outputs cannot be applied to it. Here it does matter when the effects of `fcntl` are applied to the socket state, i.e. when the leading thread makes the system call or when the system call is confirmed. However, each approach has its own problems as we show next.

Consider Figure 5.4(a), where a socket is set to non-blocking mode (using `fcntl`), before reading from it, and set back to blocking after the `read` has completed. Also,

suppose that the effects of `fcntl` are applied on confirmation (shaded boxes in the figure indicates when the system call side effects are applied to the OS socket). In this scenario, the trailing thread has not yet confirmed the first `fcntl` to set the socket to non-blocking mode when the `read` is executed by the leading peer. The `read` is forwarded to the matcher to derive external input from the OS socket, which is still in blocking mode and thus not the behavior that was intended by the application. If we now consider another approach whereby the effects of `fcntl` are applied at the time the leading thread makes the system call, the problem for socket `read` is solved but we have a similar problem for socket `write` as shown in Figure 5.4(b). We solve both of these problems by applying any state change made by the leading and trailing peers to their harness place holder and copy the harness socket state to the OS socket right before a system call is performed on that socket, so as to reflect the socket's latest state consistently across replicas.

5.3.2 Unified File Descriptor Namespace

In Section 5.2, we argued why applications should not assume anything of the file descriptor allocator. In this section, we discuss the use of arbitrary file descriptor numbers in the Replicant framework. The Linux kernel provides a `dup2` system call that allows the application developer to create a copy of a file descriptor (`old`) to an arbitrary file descriptor (`new`). After the system call completes, the two file descriptors reference the same resource and can be used interchangeably. According to `dup2` semantics, if `new` is already backed by a resource, the latter is closed and `new` now references the same resource as `old`. Typically, this facility is used by applications to duplicate the application's log file onto `stdout` and/or `stderr`. While Replicant can handle the latter case or `dup2` to other file descriptors, those that Replicant returns to the application, it cannot handle `dup2` to arbitrary new file descriptors that have not been previously allocated by Replicant. We do not see this as a fundamental problem because it is bad program design in the first place.

5.3.3 Signals

Asynchronous signals are delivered to a process when it transitions from kernel mode to user mode. This is done by checking the signal descriptor of the process and if a signal is pending for the current process, either the default signal handler is called or an application specific signal handler is invoked [8].

Since signals can be raised at any point during a process' execution, they could be delivered at different instructions in each replica. This difference in invocation point of the signal handler might cause the execution path of the replicas to diverge and hence the ordering of system calls to be different in each peer. As long as signals do not result in divergent outputs, the matcher can handle the re-ordering of system calls.

However, Replicant also provides a mechanism to enforce deterministic delivery of signals to suppress divergent outputs. When a signal is delivered to the leading peer, Replicant delays the delivery of the signal until a thread transitions from kernel-space to user-space after a system call. As such, it records the system call on which the signal was delivered for the leading peer as well as the signal type. Replicant then ensures that the same signal is delivered to the trailing peer, on the same system call instance, and delays the signal delivery if it is raised too early. This mechanism restricts signals in two ways. First, signals are only delivered after a system call has been invoked. If the application makes few system calls, the signal delivery might be arbitrarily delayed, but signals may remain pending for an unpredictable amount of time in the normal case as well [8]. Second, signals cannot be delivered on spurious system calls. Otherwise, it will never be delivered to the trailing peer. Replicant's signal delivery mechanism is similar to TightLip's [45].

Lastly, signals can be sent using system calls such as `tgkill` specify a target thread ID and group ID. Since we strive to make replicas appear as one application to the outside world, the replicas are always returned the same thread ID and group ID, no matter which replica makes the system call first. As a result, both replicas executing a `tgkill`

system call will specify the same thread ID and the same group ID. Since the target for trailing replica would be incorrect, Replicant has to translate the specified target thread ID and group ID to the appropriate thread in the trailing replica.

In practice, we have not encountered the use of signals that result in divergent outputs in the multi-threaded applications we have tested.

5.3.4 Thread Pools

Multi-threaded applications often use thread pools as an optimization. In many server applications, there is a dedicated thread that listens for new incoming connections, accepts the connections and dispatches the work to worker threads. In a simple system without thread pools, a worker thread is spawned upon accepting a new connection and its resources are freed after serving the connection, but at a performance cost. The idea of a thread pool is to avoid spawning and destroying threads on each and every connection. Upon a new connection request, the dispatcher will look for a free thread in the thread pool and delegate the work to it using thread synchronization primitives. If there is none, it spawns a new thread to handle the connection. After handling a connection, a worker thread would mark itself as free and go back to the thread pool. In this model, the threads are memoryless, i.e. they do not remember anything from any past connections.

The important implication of thread pools is that due to non-determinism in relative thread execution rates, different threads in each replica could be picked from the thread pool by the listener thread to handle the same connection, as opposed to the same peer threads handling the same connection. This causes a divergence in the system call sequences of peer threads, since they are now serving two different requests and this is analogous to two different programs that cannot be matched.

Since the threads are memoryless, one might be tempted to dynamically *re-associate* the threads in each replica, effectively re-creating the proper peer groups on every work request, as if those worker threads were being cloned. However, it is difficult to infer

application semantics by observing the sequence of system calls made by the application. More precisely, it is difficult to infer when a new request is being dispatched to a worker thread in the thread pool, which is the *re-association point*, since this is done through shared memory using synchronization primitives. Using the first system call at the start of every new connection to infer this re-association point is not a solution. If the first system call of a new request is `gettimeofday`, there is not enough context information for matching the correct instance, especially since `gettimeofday` occurs frequently and at arbitrary points in an application's execution. If the first system call made by a worker thread, upon handling a new work unit, has enough context information in its arguments and does not occur at any point except as the first system call on each new connection, e.g. `getsockname` in Apache, then this approach would be feasible. However, which system call denotes this re-association point is application specific and might not always be usable due to the lack of context.

Our initial implementation of Replicant implemented a mechanism that supported dynamic re-association of threads. While it worked for some applications, it could not easily be generalized, precisely because of the above-mentioned problems. Instead, Replicant currently solves this problem with determinism annotations by forcing peers to always handle the same connections. We consider this deviation in system call sequence as non-determinism in value since the two peer threads are serving different requests and would be writing different buffers to the network. The details of how this is done are outside the scope of this thesis and are discussed in [26].

```
 1. search_for_matching_syscall(syscall_list, syscall_info) {
 2.     /* starting from search_start pointer */
 3.     for each element in syscall list {
 4.         if ((element->rep_id != this_thread->rep_id) &&
 5.             (element->sysnum == syscall_info->sysnum)) {
 6.             match = compare_syscall_arguments(element, syscall_info);
 7.         }
 8.     }
 9.     return match | no_match;
10. }
11.
12. generic_syscall(...) {
13.   /* fast path, does not need to check the system call list */
14.   if (this_thread->syscall_count == 0) {
15.     execute syscall;
16.     record syscall args and results;
17.     return results;
18.   }
19.   else {
20.    search_for_matching_syscall(syscall_list, syscall_info);
21.    if (no match found) {
22.       peer_thread->syscall_count++;
23.       execute syscall;
24.       record syscall args and results;
25.       return results;
26.     }
27.     else {
28.       this_thread->syscall_count--;
29.       set search_start pointer to next syscall;
30.       confirm syscall and externalize results;
31.       delete syscall list entry;
32.       return results;
33.     }
34.   }
35. }
```

Figure 5.5: Matching algorithm. Note that the `execute syscall` operation combines the logic that decides whether the system calls should be executed immediately (if they have no external outputs) or buffered until confirmed (if they do have external outputs). The logic that extrapolates results on system calls with external input and external output is omitted for simplicity.

Chapter 6

Evaluation

One of the goals of Replicant is to make redundant execution of multi-threaded applications practical on multiprocessor systems. In this chapter, we evaluate the performance of applications running on Replicant, which we compare to an unmodified (vanilla) application and a theoretical best-case estimate. We next examine the performance benefits of our matcher optimization and then explore the overhead of Replicant at the micro level by analyzing the cost of a few common system calls. For closure, we evaluate the correctness of the outputs produced by applications running on Replicant.

6.1 Application Benchmarks

In order to evaluate the feasibility of Replicant with realistic multi-threaded workloads, we have chosen three multi-threaded applications from the SPLASH-2 [44] benchmark suite. These workloads can run on Replicant without the need to add any determinism annotations because the non-determinism, due to shared-memory communication, does not affect the external outputs of the application.

The SPLASH-2 benchmark suite is a set of parallel computational workloads designed to test shared-memory multiprocessor performance. Out of the suite, we ported the LU, FFT and WATER-N^2 benchmarks. All of these benchmarks communicate exclusively

through shared memory, and use locks to synchronize accesses to shared memory. Access to shared memory outside of locks was restricted to a few stylized ways. For example, FFT uses a barrier to ensure that all threads have completed their writes to the shared memory, before allowing threads to perform unsynchronized reads.

6.1.1 Methodology

Since we are not aware of any existing redundant execution systems that can support multi-threaded workloads on multi-core hardware, we develop a best-case estimate of the overhead of a kernel-based redundant execution system, against which we can measure the performance of Replicant. Our best-case estimate is computed by measuring the ratio between the time an application spends executing user code, and the time the application spends in the kernel. Any kernel-based 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 best-case performance for a particular application, we use the following method: in a run of a 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 required is t seconds. In the case where all processors are fully utilized by the application, the best-case execution time t' for the same application on an n-replica system can be estimated as:

$$t' = \frac{n \cdot u + s}{u + s} \times t \tag{6.1}$$

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 that Replicant adds with the additional bookkeeping associated with the harness and the matcher.

All benchmarks were performed on an Intel Core 2 Duo 2.13GHz machine with 1GB of memory running Fedora Core 5. The working set of all benchmarks fit in memory and the number of threads was increased until the vanilla benchmark could no longer

Application	Vanilla (s)		Best-Case (s)	Replicant (s)
	1P	2P		
FFT	2.95 (±0.00)	2.21 (±0.01)	4.29 (±0.03)	3.44 (±0.03)
LU	61.06 (±0.02)	33.61 (±0.04)	67.17 (±0.08)	58.41 (±0.09)
WATER-N^2	12.10 (±0.04)	6.30 (±0.01)	12.59 (±0.03)	12.02 (±0.06)

Table 6.1: Performance of the Replicant on three SPLASH-2 benchmarks. We also provide measurements of the unmodified application on both one processor and two processor hardware, as well as an estimate of the best-case performance of Replicant. The numbers in the brackets indicate standard deviation.

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 one CPU enabled, both CPUs enabled and a best-case estimate, derived from our dual processor runs, as described in Equation 6.1. 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 [14].

6.1.2 Results

We will now present and analyze the performance of our application benchmarks on Replicant. FFT was benchmarked using a data set of 2^{22} complex data points while LU and WATER-N^2 used a matrix size of 4096×4096. The results are summarized in Table 6.1 and compared in Figure 6.1.

Execution times for FFT, LU and Water-N^2 were averaged over 5 runs. They are all heavily computational kernels that spend very little time in the kernel. However, because of lock contention the applications exhibit poor scalability and were not able to utilize both CPUs fully. As a result, all three applications beat the best-case estimates because

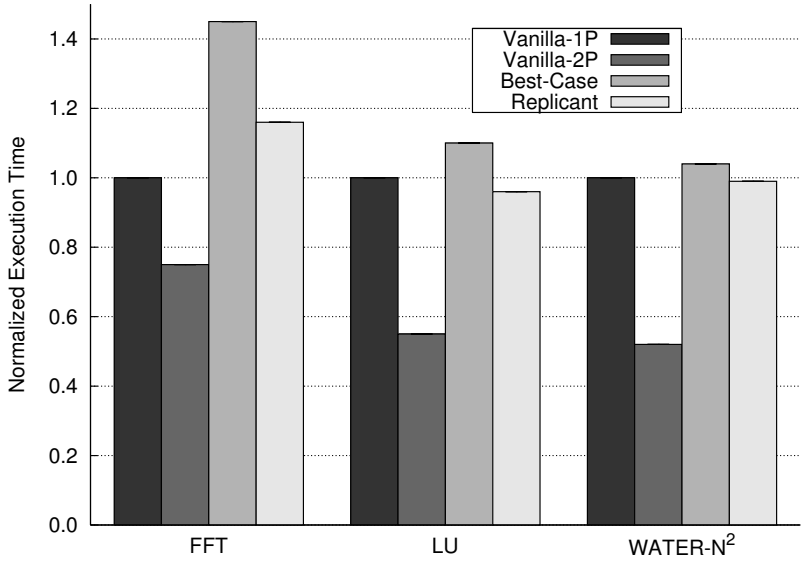

Figure 6.1: SPLASH-2 benchmarks. Comparison of the execution time of SPLASH-2 applications on Replicant, normalized to the single processor case.

Replicant was able to utilize the left over processor cycles that the vanilla applications were not able to use. Moreover, both LU and WATER-N^2 exhibited speedups on Replicant versus the single processor vanilla run, effectively illustrating that Replicant would benefit from unused cores.

FFT results are different from LU and WATER-N^2 because the data set is small and hence the runtime is small. During this short execution time, the threads are not able to ramp up CPU utilization. Increasing the data set size to the next allowable increment, i.e. 2^{24} complex data points, results in a data set that does not fit in memory and causes the CPU to thrash. The short runtime also explains why the speedup is not as significant under Vanilla-2P as compared to LU and WATER-N^2.

6.2 Matcher Optimization

In Section 5.2, we discussed an optimization that we incorporated into the matcher that allowed the leading peer to enqueue a system call in its system call list without

first searching the list. We evaluate the performance gains by using the WATER-N^2 benchmark.

6.2.1 Methodology

WATER-N^2 is very compute intensive and it spends a lot of time in user-space. In addition, it makes most of its system calls during its initialization phase and the execution time of the initialization phase is negligible as compared to the total runtime of the benchmark. In order to get meaningful results, we have benchmarked only the startup code of the WATER-N^2 benchmark, at which point it is still single-threaded.

Since the runtime of the benchmark is very short, the performance numbers are reported in clock cycles, which we obtained using the `rdtsc` assembly language instruction that reads the time stamp counter register. This instruction returns a 64-bit value that represents the number of clock cycles since processor reset and is incremented on each clock signal. Our processor is an Intel Core 2 Duo clocked at 2.13GHz with frequency scaling turned off to maintain a constant clock period across runs. The execution time of the leading and trailing peers, from start to finish, is measured by taking two time stamps (one at `execve` and the other at `exit`). In order to get accurate readings for each of the peers, without one interfering with the other, it is necessary to allow the leading peer to finish execution before the trailing peer is allowed to run. This is done using synchronization primitives in the kernel.

However, the above experiment is not a realistic scenario under normal execution. Therefore, additional measurements from a similar experiment were taken, with the difference that the leading peer and the trailing peer were allowed to execute concurrently.

6.2.2 Results

The results, averaged over 5 runs, are reported in Table 6.2. Although it is not representative of a typical run where the leading peer and the trailing peer would be executing

Configuration	Leading Peer (cycles)	Trailing Peer (cycles)
Optimized matcher	119.4×10^6	174.3×10^6
Leading peer always searches	32.7×10^9	169.7×10^6

Table 6.2: Benefits of matcher optimization whereby the leading peer does not search the system call list on every system call. The numbers, shown in clock cycles, are the total time taken to execute the initialization phase of WATER-N^2 with each of the peers executing serially.

concurrently, it does illustrate that, if the leading peer were to always search the list before enqueuing a system call, then the overhead would be proportional to the system call distance. The execution time of the trailing thread, without the matcher optimization, was unaffected.

For our second experiment, we noticed that the trailing thread spent a lot of time waiting on the spinlock protecting the system call list while the leading thread was searching the list. Since the leading thread spends more time searching the list, not surprisingly, the system call distance drops from an average of 5063 (for the optimized matcher) to an average of 603.

6.3 Microbenchmarks

We next evaluate the performance of Replicant on a few of the most frequent system calls both for the leading and trailing peers. We compare these numbers to the cost of a vanilla system call. By comparing the cost of the leading and trailing peers to the vanilla cost, we get an idea of how much overhead Replicant introduces on each of the leading and trailing peer path. It is worth noting that no efforts were put in optimizing Replicant and this is left as future work. The system calls we investigate are `time`, `read`, `write`, `open` and `close`. Note that LMbench [21] cannot be used to measure the latency of Replicant's system calls, on each of the leading and trailing paths, because of its timing harness that uses `gettimeofday`. The `gettimeofday` system call is intercepted when

the leading thread invokes it and is replayed to the trailing thread, effectively distorting real time. As a result, inaccurate time values will be returned to the timing harness in LMbench and both the leading and trailing peers will report the same latency.

6.3.1 Methodology

The microbenchmarks are single-threaded applications that make 20 system calls (excluding setup and tear-down system calls) and the average over 5 runs is computed. The cost of each system call is individually measured in clock cycles, using the `rdtsc` assembly language instruction at the entry (before line 14 from Figure 5.5) and exit (before lines 17, 25 and 32 from Figure 5.5) points of each system call, and then taking the difference. Again, we ensured that frequency scaling was turned off and additionally ensured that kernel preemption was turned off.

The numbers reported for opening, reading, writing and closing an on-disk file are all on a warm cache. The `read` and `write` microbenchmarks are reading and writing 4096 bytes respectively to the same page frame. By resetting the file position after each `read` and `write`, using `lseek`, we ensure that only one page is being accessed and that it is always in the page cache, thus eliminating the effects of kernel optimizations such as the read-ahead algorithm in Linux. The `open` and `close` numbers were obtained by running a single microbenchmark that repeatedly opens and closes the same file in a loop.

6.3.2 Results

The results are shown in Table 6.3 and compared in Figure 6.2. The `time` microbenchmark illustrates the cost of a NULL system call. It represents the best-case scenario where only user-space code execution is repeated while kernel-space code is executed only once. Moreover, since `time` is called with `NULL` as argument, the system call does not have to perform large argument copying and comparing like in `write` and there are no buffers to copy (for replication) after the system call has completed like in `read`. It is the

Microbenchmark	**Vanilla** (cycles)	**Leading Peer** (cycles)	**Trailing Peer** (cycles)
TIME	6437 (±39)	8006 (±35)	2757 (±86)
READ	4057 (±38)	11179 (±234)	6875 (±57)
WRITE	4069 (±16)	9258 (±353)	28159 (±181)
OPEN	4921 (±65)	12577 (±136)	10370 (±663)
CLOSE	1230 (±6)	2929 (±56)	4602 (±118)

Table 6.3: Performance of Replicant on common system call microbenchmarks. We provide vanilla, leading peer path and trailing peer path execution times for comparison. Numbers in brackets show standard deviation.

minimal overhead that can be incurred on Replicant, having a single word-size argument to copy and a word-size return value to buffer. From Figure 6.2, we can observe that the overhead incurred on the leading peer's execution path is not much higher than that of vanilla. The trailing peer performs even better as it does not execute the `time` system call *per se*. Instead, the trailing peer only copies the value that the leading peer read from the OS and buffered by the matcher, thus incurring purely Replicant overhead.

The `read` microbenchmark is representative of the base cost of operations by the leading and trailing peers when operating on the harness only. In this case, both peers have to execute the system call on the harness but not on the OS (by the matcher). As expected, they are both more expensive than vanilla because they incur vanilla base cost and Replicant overhead. The extra overhead of the leading peer as compared to the trailing peer is an implementation artifact due to an extra buffer copy. Recall that external inputs are replicated to the trailing peer, while non-external inputs are derived from the harness. Since `read` can be used on both external (e.g. sockets or devices like `/dev/random`) and non-external inputs (e.g. files), Replicant's default behavior (for simplicity) is to copy the buffer after the `read` system call in anticipation that it will be needed for replication. This additional copying is done on `read` from files as well, even

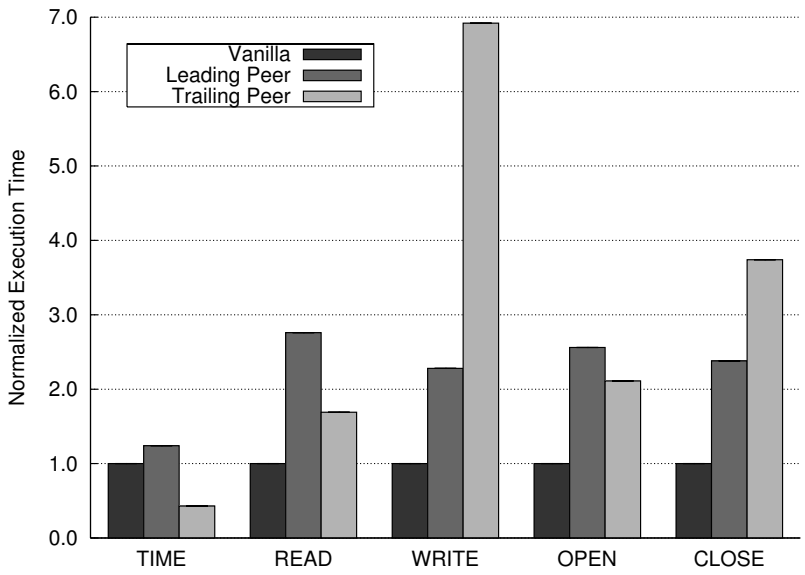

Figure 6.2: Microbenchmarks. Comparison of microbenchmark performance for vanilla, leading peer and trailing peer execution paths, normalized to the vanilla cost.

though it is not necessary. Hence, the additional overhead, which is proportional to the buffer size, is attributed to this extra copying together with the resource allocation and deallocation associated with it.

The `write` microbenchmark illustrates the cost incurred when the trailing peer confirms a system call and calls upon the matcher to externalize the system call by re-executing it on the OS kernel. In this case, the trailing peer bears the overhead incurred by the matcher, as illustrated in the pseudo-code from Figure 5.5. The extra overhead of the trailing peer is due to the byte-by-byte comparison of the write buffers when matching the system call, data structure deallocation and re-execution of the `write` system call by the matcher to externalize the buffer to the OS file. All these costs are not incurred by the leading peer and are proportional to the buffer size.

The `open` and `close` system calls are other frequent system calls that are interesting to analyze. It is worth noting that these system calls do not have external inputs but do modify the state of the harness as well as the matcher state. The update on the matcher state, e.g. the OS FDT used by the matcher, is not necessarily done at the time when the

system call is confirmed. For instance, `open` is re-executed by the matcher at the time the leading peer calls `open` because the leading peer might require external inputs from that file descriptor next, e.g. if it is reading from `/dev/random`, and therefore requires an open file in the OS FDT. Meanwhile, `close` is re-executed on the OS when confirmed because the trailing peer might still need to use this file descriptor to externalize some outstanding system calls, even though the leading peer has closed that file. This explains the higher cost of the leading peer on `open` since it incurs the cost of the matcher re-opening the file on the OS, as well as the higher cost of the trailing peer on `close`.

We do not report on the cold cache numbers but it is obvious that the leading peer incurs the high cost of a cache miss while the trailing peer does not, since they are reading the same physical page backing a file on the COW file system. This is a good example where the trailing peer would have an opportunity to catch up on the leading peer.

6.4 Output Correctness

Each of the application benchmarks were tested to confirm that the outputs produced by Replicant were indeed correct. LU and FFT both contain self-tests that check the consistency of their outputs and these were used to check the outputs that they produced when run on Replicant. The WATER-N^2 benchmark has no self-test, but did not report any errors during execution.

Chapter 7

Discussion

Although Replicant can support any multi-threaded applications with the help of determinism annotations, there are some limitations that are currently open problems. While we do not believe any of these are fundamental limitations, we point out that other redundant execution systems like [10, 45] are also not able to handle a few of them.

7.1 User-Space Randomness

Some applications rely on randomness and try hard to leverage different sources of randomness, some of which are not visible to the kernel and cannot be replicated in other replicas. For example, some `libc` library functions, like `mkstemp`, use `static` variables to store state from previous calls to this library function. Other applications, like OpenSSH and MySQL, gather randomness from the heap or the stack. Since this randomness is invisible to Replicant and is not deterministic across replicas, applications that rely on randomness will have to derive it from the kernel, by using facilities such as `/dev/random` and `/dev/urandom`. By deriving randomness from the kernel, Replicant can thus replicate the random inputs to the replicas deterministically.

7.2 Uninitialized Buffers

Since Replicant compares the contents of buffers during its matching phase, buffers that contain uninitialized data or partly initialized data will not match, and thus will not be externalized. We argue that this is bad practice because it leaks application information. This can be solved by intercepting and replacing all calls to `malloc` with `calloc` and using compiler support to zero out the stack buffers. However, this option is not available in all compilers.

7.3 `ioctl` System Call

The `ioctl` system call is used to manipulate device parameters. `ioctl` is hard to handle in a general way because its semantics, and hence the system call arguments passed to the kernel, are device specific. The manual pages for `ioctl` and `ioctl_list` illustrate the complexity of this system call, since there is no standard to `ioctl` usage and there are a large number of possible device-specific commands. Moreover, the parameters to the system call are encoded in the device-specific request code that defines which arguments are input or output buffers. These buffers, usually pointers, could in turn point to an array of pointers (buffers).

Replicant handles only a subset of the `ioctl` system call request codes, i.e. those used by the applications we benchmarked. Based on the device-specific request code, Replicant calls upon specialized functions to copy the arguments, to compare the arguments and output buffers (if needed), and to record and replicate inputs as needed.

7.4 Non-trapping Instructions

Non-trapping instructions such as the `rdtsc` assembly language instruction, used to read the time stamp counter register, are invisible to the kernel and thus are external inputs

that cannot be replicated to other replicas. If this external input value is used to influence control flow or if it affects external output, then the replicas will diverge. Another such instruction is `rdpmc`, which allows applications to read performance monitor counters.

7.5 Memory-Mapped Files

Just like inter-thread communication through shared memory is invisible to the kernel, accesses to memory-mapped files are also invisible. Since memory-mapped files are on the COW file system of the harness, the outputs are never confirmed (since they are invisible to Replicant), and thus are never externalized to the OS file. Replicant can handle this at a coarse granularity by comparing the memory-mapped regions on `unmap` and externalizing them. However, it could happen that the output values diverge due to non-determinism and therefore require determinism annotations support.

7.6 File-based Inter-Process Communication

Processes can communicate using numerous Inter-Process Communication (IPC) mechanisms. While Replicant can handle IPC through sockets and pipes in a general way, file-based IPC can only be handled if the communicating entities are within the Replicant framework.

If a process A running on Replicant performs file-based IPC with a process B not running on Replicant, then A will externalize its messages on confirmation to the OS file (which will be visible to B) but A will never see the messages that B writes to the OS file. Process A always reads from the COW file in the harness, while process B only knows about the OS file. This is because, by design, Replicant handles inputs from regular files as non-external inputs that are derived from the harness. If there were a way to differentiate between regular files and files meant for IPC (e.g. a UNIX socket – which Replicant can handle), then Replicant would be able to handle file-based IPC.

Chapter 8

Related Work

We compare Replicant to other projects along 4 major axes. We begin by discussing redundant execution, replay systems, the concept of external visibility and intrusion detection systems (IDS). We then briefly address common misconceptions about Replicant.

8.1 Redundant Execution

In Chapter 2, we provided a brief overview of redundant execution systems. In this section, we classify redundant execution systems into three categories, namely hardware-based systems, software-based systems implemented at the virtual machine monitor (VMM) layer and finally application-level redundant systems. We discuss each of them in turn.

8.1.1 Hardware

Redundancy has enjoyed a long history of use to improve system reliability and availability. For example, IBM's S390 microprocessor features redundant hardware functional units and employs aggressive error checking [34]. Results from the redundant functional units are compared on every clock cycle and rolled back on error detection to a check-

pointed state, also taken on every cycle. With similar goals for continuous availability, HP's NonStop Advanced Architecture (NSAA), which has its roots in systems designed in 1974 by Tandem Corp. [5], replicates all hardware components so that there is no single point of failure. NSAA runs an application on redundant processors in loose lockstep, i.e. they execute the same instruction stream independently and communicate their results to a voting hardware unit, which compares the outputs of I/O operations. System software running on NSAA is also redundant. They are implemented as *process pairs* where the *primary* copy executes and communicates state changes to the *backup* copy. The latter can take over should the primary copy fail.

The IBM's S390 microprocessor and HP's NSAA differ from Replicant since they feature redundant hardware units and implement error detection and fail-over mechanisms in hardware. Meanwhile, Replicant is a software layer that is part of the kernel and runs on commodity multiprocessor hardware. Moreover, while S390 and NSAA can tolerate hardware failures, Replicant cannot. Last but not least, NSAA requires a specialized middleware layer that allows applications to be run as process pairs without modifications and requires that memory state across redundant processors is synchronized to maintain determinism. In contrast, Replicant modifies the OS kernel and might require application source instrumentation with determinism annotations, where required, to maintain deterministic execution across replicas that exhibit non-determinism in value.

More recently, commodity hardware trends towards simultaneously threaded and multi-core processors have renewed interest in hardware-based redundant execution systems, e.g. SRT [29] and SlicK [24]. Both SRT and SlicK leverage simultaneous multithreading (SMT) [41], where multiple hardware contexts are provided to improve usage of superscalar microprocessors, for the purposes of redundant execution. These systems provide a cost-effective approach to transient fault detection but provide weaker guarantees than NSAA and S/390. The main difference between SRT and SlicK is that the latter does partial redundant execution, at the granularity of slices, for performance rea-

sons. Instead of executing all instructions redundantly, SlicK uses a set of predictors to predict store addresses and values. Only if the prediction fails or is indeterminate does SlicK perform redundant execution. The instructions lying on the backward-slice leading to the failed prediction is re-executed by the trailing thread.

Like Replicant, SRT and SlicK perform input replication and output checking by using two simultaneously executing threads, known as the leading and the trailing threads, which appear as one thread to the OS. However, unlike Replicant, SRT and SlicK mechanisms are all in hardware and are transparent to the application. Because of the better visibility into the hardware that hardware-based redundant execution systems 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.

Finally, hardware-based redundant execution has been used to increase application performance. For example, Slipstream processors are used to run two replicas in parallel, one of which (A-stream) runs slightly ahead of the other (R-stream) [37]. The R-stream is monitored at runtime and using predictors, useless instructions are accurately identified from the dynamic instruction stream and removed from the A-stream, which thus becomes shorter and runs faster. The A-stream in turn provides feedback (accurate branch prediction) to the R-stream, which also executes faster, while validating the execution of the A-stream. The end result is an application that runs faster than the original one. The goal of Slipstream is different from Replicant and it also requires hardware support.

8.1.2 Virtual Machine Monitor

A VMM is a thin layer of software that executes on bare hardware, below systems software. The VMM virtualizes the underlying hardware and exports an interface that allows multiple commodity operating systems to be run concurrently. Scheduling, memory management and accesses to devices are under the control of the VMM.

There have been several propositions to incorporate redundant execution logic in the VMM as a cost-effective alternative to full hardware replication for fault-tolerant systems [20, 9]. These systems implement redundant execution in software at the granularity of a virtual machine with the benefit that they do not require modifications to the operating system or the applications.

However, like their hardware-based counterparts, it will also be difficult for VMM-based solutions to compare replica outputs due to the semantic gap that exists between the VMM and the OS.

8.1.3 Applications

The idea of application-level redundancy for reliability was introduced in 1977 by Avizienis et al. with N-Version programming [3]. The idea is to generate different implementations of an application from the same initial specifications, using different programming languages, algorithms and development teams, with the assumption that there is a very low probability of identical software faults in the different implementations. Replicant is different in that it uses the same implementation of an application and introduces diversity automatically in the replicas, at a much lower cost.

More recently, there have been a plethora of projects that introduce diversity into replicas for the purposes of increasing security, privacy and reliability. The fundamental difference is that none of them can support multi-threaded applications while Replicant is able to support multi-threaded applications on multiprocessor hardware, with the help of determinism annotations where needed.

Like Replicant, the N-Variant framework [10] 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. Similarly, N-Variant uses ASLR as diversity, with the difference that the replicas have disjoint address spaces as opposed to a randomized base address, and in addition uses instruction set tagging. Moreover,

N-Variant also interacts with replicas at the system call interface. However, the difference in this interaction is that, N-Variant requires all replicas to rendez-vous and agree on every system call while Replicant allows replicas to execute independently.

On the other hand, TightLip [45] aims to protect a user's privacy through redundant execution. TightLip does not perform full redundant execution like Replicant, but instead executes redundantly only when sensitive data is accessed. It provides one replica with the requested 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. Like Replicant, TightLip compares outputs at the system call interface and performs rendez-vous on some system calls.

Similarly, Doppelganger uses two web browsers with different cookie jar contents to detect which cookies need to be stored and which ones can be safely discarded [32]. HTTP cookies are used to provide useful functionality like shopping carts and authentication to a website, but are also used to infringe on a user's privacy by tracking all websites visited by the user. The goal is of Doppelganger is to find the ideal cookie policy that would provide the user with desired functionality while preserving the user's privacy. Doppelganger forks a browser replica in the background and performs a cost/benefit analysis among alternative cookie policies, by comparing the outputs, in order to find the best policy for the user. This is fundamentally different from Replicant since it doesn't replicate inputs to the replicas to maintain deterministic execution, but instead uses different inputs to analyze similarities and differences in outputs.

DieHard implements a memory manager that approximates an infinite heap to provide probabilistic memory safety [4]. The goal is to avoid memory corruption errors and allow applications to continue executing soundly, even in the presence of these errors. To this end, the memory manager randomizes the address at which objects are allocated on the heap. DieHard also has a redundant execution mode in which the outputs of several replicas, each initialized with a different random number generation seed, are compared

to detect uninitialized reads. Unlike Replicant which intercepts system calls, DieHard intercepts library calls that it forwards to the memory manager. Moreover, DieHard does not support applications that write to the file system or to the network. As an extension to DieHard, Exterminator probabilistically detects, tolerates and corrects heap-based memory corruption error by executing replicas redundantly [23].

8.2 Replay Systems

Replicant is related to deterministic replay systems since it uses similar intercepting, recording and replaying of system calls. `liblog` [13] is a tool that helps developers debug distributed applications, which are non-deterministic in nature. `liblog` is a user-space library that is loaded in a process' address space and performs logging at the `libc` function call interface. Each process logs enough information locally during monitoring and these logs are used for deterministic replay. Like Replicant, which records the return values and input buffers of system calls for replication, `liblog` buffers return values of `libc` function calls for replay.

Flashback [35] is another replay system used for debugging software. It provides an in-memory checkpointing facility for process state and records system calls from within an OS kernel like Replicant. A process can then be rolled back and replayed by simulating the side effects of system calls previously recorded and returning the results to the replayed process. ReVirt [11] is different from Replicant in that it supports replay for an entire virtual machine. In ReVirt, the VMM replays network and keyboard input by logging accesses to these calls during the original run. ReVirt can replay asynchronous interrupts by recording the instruction pointer and branch taken counter, thus allowing it to replay multi-threaded workloads by scheduling the threads precisely. Unfortunately, as the ReVirt authors point out, this technique does not enable deterministic replay on a multiprocessor.

Rx executes several replicas to mitigate transient software faults [28]. Rather than executing the replicas simultaneously, Rx repeatedly replays the application in a slightly different environment after a crash 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.

8.3 Externally Visible Concept

Like Replicant, xsyncfs [22] also has a concept of external visibility which is called *external synchrony*. Xsyncfs implements an externally synchronous file system that provides the same guarantees as a synchronous file system, with improved performance. When an application performs a synchronous I/O operation, the outputs are buffered by the operating system and control is returned to the application, which proceeds with its execution, before the data is committed to disk. Outputs are batched and externalized when necessary, while maintaining causal output ordering to guarantee data durability. On the other hand, Replicant externalizes outputs whose content has been confirmed by the majority of replicas and the order in which outputs are externalized is dependent on the order in which system calls are confirmed.

8.4 Intrusion Detection Systems

Replicant is related to host-based IDS since it is detecting and preventing exploits from compromising vulnerable applications. Moreover, like host-based IDS, Replicant is introspecting systems calls made by the applications from within the OS kernel. We examine two techniques that IDS use, namely static code analysis and dynamic analysis.

8.4.1 Static Analysis

Wagner and Dean used static code analysis to model an application's expected behavior [42]. This approach verifies that the application's system call trace is consistent with what is expected from the source code. While static analysis has no false positives and does not require training data like dynamic analysis techniques, it suffers from several drawbacks. First, application source code is required to perform the analysis, which may not be available in all cases. Second, achieving coverage for all code paths can be very difficult, if not impossible for reasonably large programs. This gets more complicated when considering system calls made by libraries that are dynamically linked in and due to the large state space, this approach is usually slow. Replicant does not need to build a model prior to running the application but instead use the replicas as runtime models against which system calls are compared.

8.4.2 Dynamic Analysis

Dynamic analysis is an alternative technique that gathers data (system call traces of "normal" behavior) through a runtime training period. Sekar et al. use this training data to produce a finite state automaton (FSA) that represent "normal" behavior [30]. This FSA is then 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. Another approach [12] improved upon [30] by using more context information such as call stack information. Replicant does not require training to build a model but instead uses a replica of the application as model and feeds it with the same inputs. As a result, it does not suffer from poor coverage since replicas should execute the same system call sequence when given the same input.

8.5 What Replicant is Not

Replicant solves a different problem than fault-tolerant distributed systems. In a distributed setting, the problem is on how to reach consensus about the system state in a reliable way. The different components of a distributed system need to communicate their state to every other component (a vote), and together, they try to achieve a common decision. However, in a distributed environment, the problem is much harder since the system has to deal with Byzantine faults [18], i.e. messages may be dropped, faulty components may give conflicting information to the other components and it is assumed that any component can fail. Replicant solves a different problem and does not have to deal with Byzantine faults since it is a centralized system on a single host and has a centralized decision maker (the matcher), which is assumed to never be faulty.

Replicant also solves a different problem than Paxos [17]. Paxos is a three phase consensus protocol for implementing fault-tolerant distributed systems. The goal of Paxos is to enable a collection of agents in a distributed system to suggest proposals and reach agreement by majority vote, in the face of non-Byzantine faults. On the other hand, Replicant aims to detect divergent behavior among a group of independently executing replicas by comparing their outputs. Although Replicant also uses a majority vote of replicas for decision making, it does so with the help of a trusted entity – the OS kernel.

Chapter 9

Conclusions and Future Work

We have implemented and evaluated Replicant, a system that is able to efficiently support redundant execution for multi-threaded applications on multiprocessor hardware, in order to improve the security and reliability of applications. The class of multi-threaded applications that are supported without any determinism annotations on Replicant are those where the non-determinism, caused by invisible inter-thread communication, does not affect externally visible output. With the help of determinism annotations, which are used to suppress non-determinism in value, any properly instrumented multi-threaded application can run on Replicant.

Replicant leverages the independent execution of replicas for performance and tolerates replica divergence by buffering their outputs. Once the outputs are confirmed by a majority of the replicas, they are externalized. Independent execution is facilitated by sandboxing each replica in a harness, which provides the replica with enough OS state, such that it does not realize that its outputs are being buffered.

The evaluation of Replicant showed that it is able to offer good performance on the multi-threaded workloads, especially when the original applications are not able to take full advantage of all the processors available to them.

9.1 Future Work

9.1.1 Prototype Improvements

As discussed previously, no effort was put into optimizing Replicant prototype for performance. We could improve Replicant's performance by implementing its own memory management subsystem to reduce the number of data structure allocation and destruction that is currently being done on every system call interception. The current prototype can also be generalized to an n-replica system.

9.1.2 Future Research Directions

Diversity. Replicant currently uses ASLR as a form of diversity to detect memory corruption. However, many other forms of diversity could be explored to detect various classes of bugs and vulnerabilities. For instance, replicas could be compiled with different optimization levels to detect timing bugs or compiled with different compiler implementations as in [33]. Like Rx [28], Replicant could also use different library versions for each replica as another form of diversity. The caveat here is that, although the implementation of library functions can be different, they will need to invoke the same sequence of system calls with the same arguments so that the system calls can be matched.

Recovery. Replicant currently detects divergent behavior and other forms of diversity improve upon the detection aspect. Future work could also examine the interesting recovery problem upon detection of attacks or failures. Currently, in the face of attacks or failures that disrupt a minority of replicas, only data integrity is preserved. However, we could explore mechanisms to recover from attacks or failures. More specifically, can the faulty/crashed replicas be discarded and new ones spawned to replace them? This raises a number of interesting problems, e.g. how to get the new replicas up to speed and re-generate the appropriate application/replica state?

Part-Time Replicant. For the purposes of intrusion detection, Replicant replaces the application with diversified replicas to prevent memory corruption attacks. This improves upon running a single application with ASLR since the latter is only probabilistic and brute force attacks are possible within minutes [31]. However, while the performance degradation of an application with ASLR only is close to zero, it is significantly more in Replicant. Conceptually, we could think of a server application triggering Replicant if and only if it starts experiencing frequent crashes, which might be indicative of an attack attempt in progress. The concept of triggering the heavy-weight solution is similar to that of Sweeper [40] which uses light-weight mechanisms in the common case and falls back to heavy-weight analysis when the appropriate flags have been raised; or TightLip [45] which spawns a doppelganger process only when sensitive information is being accessed.

Multi-Versioning. Finally, Replicant could also be used for multi-versioning. Previous work [43] has shown that soon after a vulnerability is disclosed, application developers try to patch it as fast as possible. This results in a patch that has undergone limited testing and organizations rightfully delay the patching of their production systems for further testing in their customized environment. This delay opens an undesirable vulnerability window during which the production servers are vulnerable to attacks. Moreover, assuming that the patches are only a few lines of code [16], then it is very likely that the sequence of system calls are identical in both the patched and unpatched versions. Therefore, Replicant could potentially be used to address this problem by running the patched and the unpatched versions concurrently, replicating inputs to each version and verifying their outputs. If the patched replica crashed anywhere outside the patched region of code, this would be indicative of new bugs being introduced by the patch, and the unpatched replica would be allowed to complete the current request. However, if Replicant detected a crash in the patched region of code, this would be indicative of an attack attempt. At this point, the request would be discarded and a new patched replica would be spawned. This strategy effectively increases the availability of computing systems.

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, 1996. 3.2

[2] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 7(49), 1996. 2.2

[3] 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 Conference (COMPSAC)*, pages 149–155, November 1977. 2.1, 8.1.3

[4] 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. 8.1.3

[5] 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, 8.1.1

[6] Sandeep Bhatkar, Daniel C. DuVarney, and Ron Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Pro-*

ceedings of the 12th USENIX Security Symposium, pages 105–120, August 2003. 2.2

[7] Sandeep Bhatkar, Ron Sekar, and Daniel C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286, August 2005. 2.2

[8] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 3rd edition, 2005. 5, 5.1, 5.3.3

[9] 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. 8.1.2

[10] 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, 3.1, 4.1, 7, 8.1.3

[11] 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. 2.3, 8.2

[12] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 62–77, May 2003. 8.4.2

[13] 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. 8.2

[14] Intel. Teraflops Research Chip, 2007. `http://www.intel.com/research/platform/terascale/teraflops.htm` (Last accessed: 05/17/2007). 6.1.1

[15] Intel Corp., 2007. `http://www.intel.com/technology/magazine/computing/-quad-core-1206.htm` (Last accessed: 03/08/2007). 1

[16] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting Past and Present Intrusions Through Vulnerability-Specific Predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 91–104, October 2005. 9.1.2

[17] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News*, 32(4):18–25, December 2001. 8.5

[18] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982. 8.5

[19] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471–482, April 1987. 3.2

[20] 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. 8.1.2

[21] Larry W. McVoy and Carl Staelin. LMbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 Annual Usenix Technical Conference*, pages 279–294, January 1996. 6.3

[22] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the Sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, November 2006. 8.3

[23] 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. 8.1.3

[24] Angshuman Parashar, Anand Sivasubramaniam, and Sudhanva Gurumurthi. SlicK: Slice-based Locality Exploitation for Efficient Redundant Multithreading. In *Proceedings of the 12th International Conference Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 95–105, October 2006. 8.1.1

[25] PaX, 2007. `http://pax.grsecurity.net`. 5.1

[26] Jesse Pool. Kernel Support for Deterministic Redundant Execution of Shared Memory Workloads, August 2007. 1, 3.3, 5.3.4

[27] Jesse Pool, Ian Sin Kwok Wong, and David Lie. Relaxed Determinism: Making Redundant Execution on Multiprocessors Practical. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 25–30, May 2007. 1, 3.3

[28] Feng Qin, Joe Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 235–248, Oct 2005. 8.2, 9.1.2

[29] 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. 8.1.1

[30] 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*, pages 144–155, May 2001. 8.4.2

[31] 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. 1, 2.2, 9.1.2

[32] 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. 8.1.3

[33] Daniel P. Siewiorek and Priya Narasimhan. Fault Tolerant Computing and Architectures for Space and Avionics Applications. In *Proceedings of the 1st International Forum on Integrated System Health Engineering and Management in Aerospace*, November 2005. 2.1, 2.2, 9.1.2

[34] 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. 8.1.1

[35] 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. 2.3, 8.2

[36] Mark Stamp. Risks of Monoculture. *Communications of the ACM*, 47(3), March 2004. 2.2

[37] Karthik Sundaramoorthy, Zachary Purser, and Eric Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the 9th International Conference Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 257–268, November 2000. 8.1.1

[38] The Linux Kernel Archives. Linux Kernel Source, 2007. `http://kernel.org` (Last accessed: 06/20/2007). 5

[39] Wilfredo Torres-Pomales. Software Fault Tolerance: A Tutorial. Technical Report NASA/TM-2000-210616, National Aeronautics and Space Administration (NASA), 2000. Available at `http://techreports.larc.nasa.gov/ltrs/-PDF/2000/tm/NASA-2000-tm210616.pdf`. 2.1

[40] Joseph Tucek, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Song. Sweeper: A Lightweight End-to-End System for Defending Against Fast Worms. In *Proceedings of the 2nd ACM SIGOPS EuroSys*, April 2007. 9.1.2

[41] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22th International Symposium on Computer Architecture (ISCA)*, pages 392–403, June 1995. 8.1.1

[42] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–168, May 2001. 8.4.1

[43] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the 2004 ACM SIGCOMM Conference*, pages 193–204, August 2004. 9.1.2

[44] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22th International Symposium on Computer Architecture (ISCA)*, pages 24–36, June 1995. 1.1, 3.3, 6.1

[45] Aydan Yumerefendi, Benjamin Mickle, and Landon Cox. TightLip: Keeping Applications from Spilling the Beans. In *4th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 159–172, April 2007. 1, 2.1, 4.1, 4.2.2, 5.3.3, 7, 8.1.3, 9.1.2