Relaxed Determinism: Making Redundant Execution on Multiprocessors Practical

Jesse Pool Ian Sin Kwok Wong David Lie
Department of Electrical and Computer Engineering
University of Toronto
$\{$*pool,iansin,lie*$\}$*@eecg.toronto.edu*

Abstract

Given that the majority of future processors will contain an abundance of execution cores, redundant execution can offer a promising method for increasing the availability and resilience against intrusions of computing systems. However, redundant execution systems rely on the premise that when external input is duplicated identically to a set of replicas executing the same program, the replicas will produce identical outputs unless they are compromised or experience an error. Unfortunately, threaded applications exhibit non-determinism that breaks this premise and current redundant execution systems are unable to account for this non-determinism, especially on multiprocessors. In this paper, we introduce a method called *relaxed determinism* that is utilized by our system, called *Replicant*, to support redundant execution with reasonable performance while tolerating non-determinism.

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 [6]. These multiprocessors can offer increased performance through parallel execution, as well as added 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 solely determined by their inputs, such that any divergence in their outputs must indicate a problem. For example, executing identical replicas has been used to detect and mitigate soft-errors [2]. More recently, there have also been several proposals to execute slightly different replicas to detect security compromises [4] and leaks of private information [9]. However, none of these systems have addressed threaded workloads.

Systems that support redundant execution via the OS kernel or a virtual machine monitor cannot account for the non-determinism that occurs among replicas when running threaded workloads on multiprocessors [3, 4, 9]. For similar reasons, systems that support deterministic replay also fall short on multiprocessor systems [5, 8]. This non-determinism undermines redundant execution by causing *spurious divergences* among replicas that are not the result of any failure or attack.

We believe the key insight that will allow efficient support for redundant execution on multiprocessors is that, in many cases, differences in the order that events are delivered to an application will not result in significant differences in application behavior. In light of this, we propose a relaxed deterministic execution model that is analogous to the relaxed memory consistency models used to improve performance on modern processors [1]. Modern processors relax the memory ordering guarantees they provide, and only enforce strict ordering when the software developer uses a memory "fence" or "barrier" instruction. Similarly, our system loosely replicates the order of events among replicas, and only enforces a precise ordering when instructed to do so through *determinism hints* that the developer inserts into the application. With the right relaxed determinism model, redundant execution on multiprocessors can be supported with reasonable overhead on existing processors.

We have implemented a redundant execution system that supports relaxed determinism, called *Replicant*, which is able to both increase the availability of a system, as well as prevent intrusions. In this work, we will begin with a detailed description of the problem faced

```
int counter = 0;
void thread_start(){
  int local;
  lock();
  counter = counter + thread_id();
  local = counter;
  unlock();
  printf(“%d\n”,local);
}
void main(){
  thread_create(thread_start); // thread id = 1
  thread_create(thread_start); // thread id = 2
  thread_create(thread_start); // thread id = 3
}
```

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 1: Code example illustrating non-determinism in a 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.

by redundant execution systems on multiprocessors and follow with our approach to relax deterministic replication. We then give high-level details on annotating applications for Replicant, as well as some preliminary results.

2 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 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 1. 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 locked 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 locked section in order (2, 3, 1). This example demonstrates that 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 between the two replicas. If the redundant execution system ensures that threads enter the locked region in the same order in both replicas, then both replicas will produce the same outputs, though possibly in different orders. If the system further forces the replicas to also execute 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 [5] and Flashback [8] to deterministically replay 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.

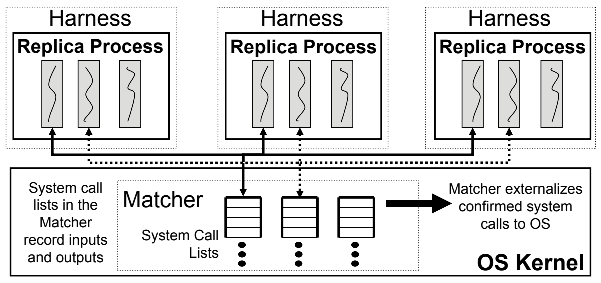
Harness
Replica Process
Harness
Replica Process
Harness
Replica Process
System call lists in the Matcher record inputs and outputs
Matcher
System Call Lists
Matcher externalizes confirmed system calls to OS
OS Kernel

Figure 2: Replicant Architecture.

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

3 Approach

Due to the non-deterministic ordering of communication events, threaded applications can produce outputs in different orders and of different values when given the same inputs over consecutive runs. Unfortunately, forcing deterministic execution for replication by lock-stepping or trapping on shared memory accesses will have a high performance cost. Instead, Replicant allows the ordering of events to diverge between replicas, and requires determinism hints from the developer to indicate when a divergence in ordering can result in a divergence in output values. In this section, we will describe Replicant's system model, enumerate the relaxations to determinism, and the primitives Replicant provides to remove these relaxations.

3.1 System Model

Replicant implements an input replicating and output matching architecture that is tolerant to the reordering of events. Like other redundant execution systems, Replicant manages the inputs and outputs of several replicas of an application to appear as a single process to an external observer. However, unlike other systems, Replicant allows replicas to execute independently and does not assume that divergence in behavior is necessarily indicative of an application problem. Instead, Replicant buffers outputs and only makes them *externally visible* when they are *confirmed* (i.e. independently reproduced) by the majority of replicas. This makes Replicant tolerant to the non-deterministic reordering of outputs that concurrent applications typically exhibit.

Conceptually, Replicant can be viewed as computing the outputs of a correct execution from the system calls that the majority of replicas make. While an adversary may be able to compromise a subset of replicas, a majority is needed to subvert the externally visible behavior of the application. By increasing the number of replicas and introducing differences among replicas, such as address space randomization, 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.

Replicant's architecture is described in Figure 2. Replicant strives to increase performance by removing dependencies between replicas. To do this, Replicant should allow replicas to execute independently and diverge in their behavior. This is achieved by executing each replica in an OS sandbox, called a *harness*, which includes a private copy of the process-specific OS state and a copy-on-write file system. The purpose of the harness is to replicate the underlying OS state with enough fidelity such that the replicas are not aware that their outputs are actually being buffered (e.g. a replica will never notice an unconfirmed write to the file system). Harness state is visible only to the replica itself and is kept up-to-date by applying the outputs and effects of system calls made by the replica to the harness. Replicant also adds a *matcher* component to the OS kernel for each set of replicas. The purpose of the matcher is to fetch and replicate inputs from the external world into the harness, and determine when outputs from the harness should be made externally visible. The matcher is implemented as a set of system call lists that buffer the arguments and results of system calls made by the replicas. System calls are matched based on the thread identifier, the system call name and its arguments.

As summarized in Table 1, Replicant splits the handling of each system call invoked by a replica 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 not. A non-external input is one that can be derived from the harness state, such as a `read` from a file on the copy-on-write file system, while an external input is one that must be derived from the OS, such as a `read` from the network or from a device. Replicant records the inputs from external system calls because they may not yield the same inputs if performed again at a later

	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 matcher 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 by the matcher.

Table 1: Replicant's handling of system calls.

time. Replicant then replays the recorded inputs to other replicas when they make the same system call. System calls that do not require external input do not need to be buffered because each replica is initially provided with identical copies of the OS state in their harness.

Similarly, a non-external output is one that another application or user on the system cannot perceive, such as a `write` to a pipe between two threads in a replica, and an external output is one whose effects are externally visible, such as an `unlink` that deletes a file. Output system calls are executed on the harness, and if they are externally visible, they are committed to the external OS state when confirmed. For example, a `write` to the file system is a system call with external output. As such, its output is applied to the harness and committed when confirmed – which will succeed unless there is a catastrophic failure of the disk. However, a `write` to a socket is a system call with external output but also requires external input derived from the matcher's socket as opposed to the harness. Since the system call cannot be executed until confirmed, Replicant extrapolates the external input from the state of the socket and allows the replica to proceed. The socket `write` is buffered and externalized when confirmed.

3.2 Determinism Hints

Similar to relaxed memory consistency models, Replicant's behavior can be modeled as a set of relaxations that are made to the strict determinism that would be enforced by lock-stepping replica execution. From our system model, we can enumerate those relaxations and provide two determinism hints with which the programmer can temporarily suppress those relaxations when needed.

First, because Replicant does not explicitly force a deterministic ordering of events among replicas, they may produce divergent outputs due to non-determinism as illustrated in Figure 1. Since Replicant will not externalize divergent outputs, we provide the developer with a *sequential region* hint to enforce a deterministic ordering on events that affect external outputs. A sequential region encompasses a section of code, and Replicant ensures that all threads in all replicas will pass through sequential regions in the same order. This concept is similar to the shared object abstraction introduced by LeBlanc et al. [7].

Sections of code that contain inter-thread communication that can affect external outputs must be executed within sequential regions. Inserting sequential regions is straightforward in a conservatively written application where all accesses to shared memory are protected by locks. Because threads communicate through the locked memory, a simple solution is to place a sequential region around each critical section. By ensuring that memory accesses in sequential regions occur in the same order on all replicas, Replicant provides the same guarantees as lock-stepping but at a lower cost since deterministic execution is only enforced when threads are accessing shared data structures.

The second source of non-determinism is caused by instances where Replicant is unable to extrapolate the results of a system call that has an external output and requires an external input. A concrete example of this occurs when an application writes to a network socket. Replicant will return a result based on the state of the socket at the time the replica makes the system call. Later, when the system call is confirmed and is executed on the OS, the remote side may have closed the connection causing the socket write to fail. Other replicas who made the system call earlier may have been led to believe that the write succeeded because the remote client was still connected at the time. This leads to divergent results being returned to the replicas. Under these circumstances, the developer may suppress the relaxed determinism with a *synchronize syscall* hint. This hint instructs Replicant to cause the thread to block when it executes the next system call until it is confirmed, thus relieving Replicant of the need to extrapolate the return value.

Replicant also relaxes consistency between inputs and external outputs by delaying outputs until they are confirmed. Thus, Replicant may reorder inputs and outputs with respect to an earlier output in a way that is conceptually similar to the memory consistency guarantees provided by Partial Store Order [1]. The developer can also use the synchronize syscall hint to suppress this relaxation.

4 Annotating Applications

In the previous section, we enumerated the sources of non-determinism that may result in divergent outputs among replicas. Since Replicant will not externalize outputs with divergent values, the application developer must use determinism hints to annotate the events that can affect the outputs required for an external observer to perceive a correct application execution. Replicant will ensure that annotated events are deterministically replayed in all replicas.

Replicant introduces non-determinism among replicas in two ways: through non-deterministic ordering of inter-thread communication events and through non-deterministic input values extrapolated from system calls that have external output. Figure 1 illustrates inter-thread communication occurring through a shared counter variable. In this example, the developer can use a sequential region hint to ensure that the threads in both replicas access the shared variable in the same order, thus causing the outputs between the two replicas to be identical.

Figure 1 illustrates a key insight that makes it easier to place sequential region hints – accesses to shared variables are typically protected by critical sections defined by `lock` and `unlock` pairs. This has motivated us to create a sequential region programming interface which reflects that of locks. Replicant extends the Linux kernel with a `begin_seq` system call and an `end_seq` system call, which informs Replicant when a thread is entering and leaving a sequential region respectively. The developer uses these system calls by placing a `begin_seq` whenever a critical section begins, such as before the `lock` statement in Figure 1, and an `end_seq` whenever the critical section ends, such as right after the `unlock`. Sequential regions need only be inserted if the ordering of events can affect externally visible outputs. For example, if the program in Figure 1 did not print the intermediate values of the shared counter variable on line 8, but instead only printed the final value after all threads had updated it, then no sequential regions would be needed. This is because the thread ordering no longer has any effect on the application output. Since sequential regions enforce ordering across threads, they can reduce opportunities for concurrency, and should be used only when necessary.

In annotating an application, the developer may need to annotate several critical sections with sequential regions. To avoid adding unnecessary dependencies between critical sections protected by unrelated locks, Replicant allows the developer to define an arbitrary number of sequential region *domains*. Replicant enforces the order in which threads cross sequential regions that belong to the same domain, but does not enforce any order on sequential regions in different domains. As a result, there is a one-to-one mapping between locks in an application and sequential region domains, and each critical section that is protected by a certain lock maps to a sequential region in the corresponding domain. Sequential region domains are initialized through the `init_seq` system call, which takes a word-length domain identifier as an argument. This identifier is also passed as an argument to `begin_seq` and `end_seq` calls to identify which domain the sequential region belongs to.

While one can infer most of the inter-thread communication in an application from its use of locks, developers frequently find application-specific opportunities to increase performance by avoiding the use locks when accessing shared variables. As a result, when porting applications, we have found that while using information gleamed from the locks to automatically add sequential regions saves a great deal of time, some amount of manual analysis is usually required to discover the communication that does not occur in a critical section, but can still affect external output values. Qualitatively, we have found that inserting sequential regions is as difficult, and very similar in process, to inserting locks to parallelize an application. While the proper use of locks is certainly not trivial, they are in common use in concurrent applications today. Therefore, we feel that, if done at the time of development, the addition of sequential regions will not be an overly heavy burden on the application developer.

The other circumstance where Replicant may introduce non-determinism among replicas is by returning different extrapolated input values in response to system calls. In these cases, a synchronize syscall hint can be used to eliminate non-deterministic inputs at the developer's request. Similar to the sequential region, the developer need only insert this hint if the extrapolated input of the system call will affect the application's output values. We have added the `make_sync` system call for use in application annotations. This annotation is only required before instances of system calls with external inputs and extrapolated outputs where the application or a library checks the return value of the system call and takes some output action based on the return value.

5 Preliminary Results

We study the effectiveness of a 2-replica implementation of Replicant on six representative threaded appli-

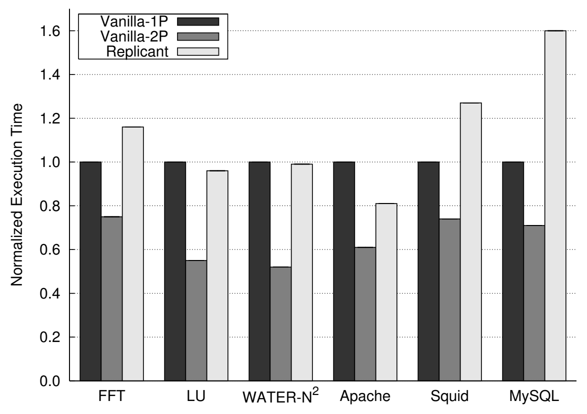

Figure 3: Performance of Replicant on six representative threaded applications as compared to 1P and 2P.

cations. We have ported three SPLASH-2 benchmarks (FFT, LU and WATER-N^2), Apache HTTP Server 2.2.3 (with worker Multi-Process Module), Squid Web Proxy Cache 2.3.STABLE9 (with asynchronous I/O enabled) and MySQL 5.0.25. All benchmarks were performed on an Intel Core 2 Duo 2.13GHz machine with 1GB of memory running Fedora Core 5 on a Gigabit network. The working set of all benchmarks fit in memory and the number of threads was increased until the dual processor vanilla benchmark could no longer utilize any more CPU time. We note that this does not mean that applications were necessarily able to utilize both CPUs to their maximum.

Figure 3 illustrates the performance of Replicant as compared to unmodified (vanilla) application performance on single processor and dual processor configurations. The comparison against the single processor performance is indicative of the case where the vanilla application is unable to make use of all processors available due to lack of sufficient parallelism. This is a reasonable scenario considering that future processors are projected to have many cores.

We find that there are three major application-dependent factors in Replicant performance. The first is how well the application balances load among threads. Squid has poor load balance and exhibits poor performance, while Apache, a very similar application, has good load balance and enjoys good performance. The second is the number of determinism hints that need to be invoked. MySQL has many sequential regions due to its frequent use of locks. Since sequential regions reduce opportunities for concurrency, MySQL experiences higher overhead. Finally, the ratio of user-space to kernel-space execution will affect application performance. Applications that spend much of their time in the kernel will experience less overhead because many kernel operations are only performed once, where as user space execution must always be duplicated.

6 Conclusion

By relaxing the determinism requirement among replicas in a redundant execution system, Replicant is able to provide better security and reliability at a lower cost to performance than systems that enforce strict deterministic replication of execution.

Acknowledgments

We would like to acknowledge Lionel Litty, Tom Hart, Ashvin Goel and the anonymous reviewers for their helpful comments. This work was funded in part by an NSERC Discovery Grant and a MITACS Seed Grant.

References

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

[2] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop advanced architecture. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN)*, pages 12–21, June 2005.

[3] A. L. Cox, K. Mohanram, and S. Rixner. Dependable $\neq$ unaffordable. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability*, pages 58–62, Oct. 2006.

[4] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-Variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*, pages 105–120, Aug. 2006.

[5] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. 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, Dec. 2002.

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

[7] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, Apr. 1987.

[8] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. 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.

[9] A. Yumerefendi, B. Mickle, and L. Cox. TightLip: Keeping applications from spilling the beans. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2007.