Efficient Deterministic Replay through Dynamic Binary Translation

Piyus Kedia

Amarnath and Shashi Khosla School of IT

This dissertation is submitted for the degree of

Doctor of Philosophy

Indian Institute of Technology Delhi

January 2015

To *Nature*

Cerificate

I am satisfied that the thesis presented by `Piyus Kedia` is worthy of consideration for the award of the degree of *Doctor of Philosophy* and is a record of original bonafide research work carried out by him under my guidance and supervision and that the results contained in it have not been submitted in part or full to any other university or institute for award of any degree or diploma.

Dr. Sorav Bansal
Computer Sc. & Engg.

Acknowledgements

I would like to thank my loving parents for their love and affection and for the flexibility to do whatever I like. I would like to thank my Grandfather and Grandmother for teaching me to uphold the values they passed on to me at every stage of my life. I want to thank my teachers in school who introduced me to mathematics and science at a very early stage in my life.

I want to thank my supervisor Dr. Sorav Bansal without whom this thesis would not have been possible. If not for him I would perhaps have not done a PhD at all. Through him I was able to discover my true area of interest. He helped me immensely in writing my papers and running through my practice talks. His spending countless hours discussing my problems at the right times was a big driving force behind this thesis. It is a big privilege to be his student.

I would like to thank my friends Dipanjan and Kinshuk for making my days memorable during my stay at IIT Delhi. Without them it would not have been so easy to complete this PhD. I want to thank all my lab mates for wierd discussions we so often had which made the lab so much more lively and fun to work in. I would also like to thank my committee members for attending my PhD progress talks.

I want to thank MHRD and IBM for supporting me financially during my PhD.

Abstract

We present an efficient software implementation to deterministically record and replay a full multiprocessor virtual machine (VM), including its guest OS kernel and applications. Deterministically replaying a shared-memory monolithic OS kernel (like Linux) presents a significant performance challenge, and we demonstrate the use of dynamic binary translation to achieve this objective.

Dynamic binary translation (DBT) is a powerful technique with several important applications. System-level binary translators have been used for implementing a Virtual Machine Monitor [2] and for instrumentation in the OS kernel [29]. In current designs, the performance overhead of binary translation on kernel-intensive workloads is high. e.g., over 10x slowdowns were reported on the syscall nanobenchmark in [2], 2-5x slowdowns were reported on `lmbench` microbenchmarks in [29]. These overheads are primarily due to the extra work required to correctly handle kernel mechanisms like interrupts, exceptions, and physical CPU concurrency. Since the overhead of DBT is itself very high hence we can not use it for improving determinstic replay performance. We present a kernel-level binary translation mechanism which exhibits near-native performance even on applications with large kernel activity. Our translator relaxes transparency requirements and aggressively takes advantage of kernel invariants to eliminate sources of slowdown. We have implemented our translator as a loadable module in unmodified Linux, and present performance and scalability experiments on multiprocessor hardware. Although our implementation is Linux specific, our mechanisms are quite general; we only take advantage of typical kernel design patterns, not Linux-specific features.

The biggest challenge in deterministically replaying a multiprocessor system is recording the order of shared memory read and writes. The previous comparable approach [27] uses CREW (Concurrent Read Exclusive Write) protocol at the page granularity. Page grained CREW protocol uses hardware page protection technique(EPT/Shadow) to restrict the access privilege of the CPUs such that, multiple CPUs can read from a page by acquiring shared access privilege of the page but for writing to a page they needs to acquire the exclusive access privilege of that page. This scheme suffers from false sharing and huge shuttling between processors for benchmarks having large amount of sharing such as Linux kernel. Every transfer of privilege is recorded in order to reproduce the same transition during the replay. We implement CREW at the byte granularity using DBT to eliminate false sharing. To achieve this we insert reader/writer lock before every shared memory access and then reduce the cost of these locks by choosing the granularity of locks such that number of lock acquisition is minimized at the cost of losing concurrency.

There are two broad approaches to doing this, which we call *data-level mutual-exclusion* and *code-level mutual-exclusion*.

Data-level mutual-exclusion models all code and data as belonging to one shared address space, and synchronizes each memory access by each CPU. In other words, a CPU is modeled as a thread executing in a shared-memory cache-coherent address space. This is identical to the underlying hardware model on shared-memory machines. Synchronization involves CREW-like ownership tracking of memory locations, which involves associating metadata with each memory location to store its ownership status. Code-level mutual-exclusion, on the other hand, divides code regions into disjoint sets called *monitors*, with the property that instructions from two different monitors can never access a memory location concurrently. In this model, ensuring that at most one CPU is active inside a monitor at all times, suffices. We propose a hybrid approach which uses data-level mutual-exclusion for some part of the code and code-level mutual-exclusion for other part of the code for better speedup.

Our implementation exhibits 15-273% recording overhead for several important kernel-intensive benchmarks on a four-processor machine, which is an 11x average improvement over the best existing comparable approach.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

The ability to record and deterministically replay a computer system is a powerful capability with applications in debugging, fault tolerance, intrusion detection, remote attestation, computer forensics, dynamic analysis, testing and verification, and more. While uniprocessor record/replay [15, 24, 26, 30, 58, 61, 66]. exhibits typical runtime overheads of less than 20% and log growth rates of less than 100 KBps (making it practical for use in even production systems), multiprocessor record/replay is significantly more complex and expensive, due to the non-determinism of concurrent accesses to shared memory.

Multiprocessor deterministic replay has been a busy area of research over the past decade. However, the experimentation and evaluation in all previous work has been limited to application-level workloads [4, 27, 38, 42, 55, 62]. In contrast, deterministically replaying a monolithic OS kernel (like Linux) is significantly more challenging for the following reasons:

- Tightly-coupled shared-memory style of programming in the kernel: This results in a large amount of data-sharing among processors, resulting in a high degree of non-determinism due to concurrent memory accesses. Compared to application-level workloads, memory sharing in an OS kernel is denser and much more varied. This results in higher overheads for approaches that involve recording the order of shared memory accesses.

- Lock-free and hidden synchronization: A mature shared-memory monolithic OS kernel usually involves several types of lock-free primitives and hidden synchronization. Many previous approaches to deterministically replay a software system, rely on the ability to identify and interpose on synchronization operations, making them unsuitable for use in this setting.

Approaches involving hardware support for faster tracking of memory sharing among CPUs have also been proposed for deterministic replay [7, 33, 34, 45, 47, 50, 63, 65], and the fundamental performance tradeoffs remain the same.

We demonstrate the use of Dynamic Binary Translation to achieve an efficient software-only implementation of multiprocessor replay.

1.1 Dynamic Binary Translation

Dynamic Binary Translation(DBT) is a technique which transforms the code as it executes. Current system-level binary translators exhibit large performance overheads on kernel-intensive workloads. For example, VMware's binary translator in a Virtual Machine Monitor (VMM) shows 10x slowdowns for the `syscall` nanobenchmark [2]; corresponding overheads are also observed in macrobenchmarks. VMware's DBT performance also includes the overheads of other virtualization mechanisms, like memory virtualization through shadow page tables, etc. Another kernel-level binary translator, DRK [29], reports 2-5x slowdowns on kernel intensive workloads. Applications requiring high kernel activity (like high performance fileservers, databases, webservers, software routers, etc.) exhibit prohibitive DBT slowdowns, and are thus seldom used with DBT frameworks. Since the overhead of DBT is itself very high, we can not use it for any optimization. We implemented a kernel-level dynamic binary translator with near-native performance. Our design differs from VMware and DRK in the following important ways:

Entry Points: We replace kernel entry points (interrupt and exception handlers) directly with their translated counterparts. This is in contrast with VMware and DRK which replace kernel entry points with calls to the DBT's dispatcher (see Figure 3.1 for the component diagram of a dynamic binary translator), which in turn jumps to the translated handlers *after* restoring native state on the kernel stack. This extra work by DBT dispatcher causes significant overhead on each kernel entry. In our design, kernel entries execute at near-native speed. In doing so, we allow the kernel handlers to potentially observe the non-native state generated by the hardware interrupt on stack.

Interrupts and Exceptions: Previous DBT solutions (VMware, DRK) handle exceptions by emulating precise exceptions[1] in software by rolling back execution to the start of the translation of the current native instruction; and handle interrupts by delaying them till the start of the translation of the next native instruction. These mechanisms are complex and expensive, and are primarily needed to ensure that the interrupt handlers observe consistent

[1]A precise exception means that before execution of an exception handler, all instructions up to the executing (emulated) instruction have been executed, and the excepting instruction and everything afterwards have not been executed. Previous DBT implementations have preserved precise exception behaviour for architectures that support precise exceptions (e.g., x86).

state. In our design, we allow imprecise exceptions and interrupts. Relaxing precision greatly simplifies design and improves performance. In our experience, operating systems rarely depend on precise exception and interrupt behaviour.

Reentrancy and Concurrency: The translator's code and data structures need to be reentrant to allow interrupts and exceptions to occur at arbitrary program points. Similarly, physical CPU concurrency needs to be handled carefully. DBT requires maintenance of CPU-private data structures, and migration of a thread from one CPU to another should not cause unsafe concurrent access to common state. In our design, the presence of imprecise exceptions and interrupts introduces more reentrancy and concurrency challenges. We present an efficient mechanism to provide correct translated execution.

Our optimizations result in a very different translator design from both VMware and DRK. We are more aggressive about assumptions on usual kernel behaviour. In doing so, we sometimes relax "transparency"; for us, ensuring *correctness* is enough. Essentially, we show that many transparency requirements are unnecessary and can be relaxed for better performance. Like DRK [29], our translator works for the entire kernel including arbitrary devices and their drivers.

1.2 Deterministic Replay

The biggest challenge in deterministically replaying a multiprocessor system is recording the order of shared memory read and writes. Our implementation records and replays the OS kernel and its applications. We use DBT to achieve an efficient implementation. The use of DBT entails the following advantages over previous work:

1. Works on unmodified binary code: Our scheme can deterministically replay unmodified binary software. i.e., source code is not required.

2. Low overhead recording, without hardware support.

3. Independent of the synchronization primitives used: Our scheme does not rely on the presence or absence of specific synchronization primitives — neither for correctness, nor for performance. This is a significant advantage over previous work that relies on such information [4, 42, 52, 55, 57, 62].

4. Flexibility: We present two primary methods to record the non-determinism, one involving code-level mutual exclusion, and the other involving data-level mutual exclusion. Section 4.2 discusses the two schemes and the tradeoffs involved, and how they map

directly to the classic tradeoffs between message-passing and shared-memory architectures. The ideal method and its associated granularity, depends on the workload. We explore this design space in the thesis. Our system uses a hybrid method which adapts to the workload characteristics to choose the best method/granularity for different regions of the system. This flexibility is unique to our approach, and is crucial to achieving low overheads.

The previous comparable approach [27] uses CREW (Concurrent Read Exclusive Write) protocol at the page granularity. Page grained CREW protocol uses hardware page protection technique(EPT[35]/ Shadow Page Tables[2]) to restrict the access privilege of the CPUs such that, multiple CPUs can read from a page by acquiring shared access privilege of the page but for writing to a page they needs to acquire the exclusive access privilege of that page. This scheme suffers from false sharing and huge shuttling between processors for benchmarks having large amount of sharing such as Linux kernel. Every transfer of privilege is recorded in order to reproduce the same transition during the replay. Furthermore these transitions involve VM exits which have high overheads – each VM exit executes thousands of instructions. We implement CREW at the byte granularity using DBT to eliminate false sharing. To achieve this we insert reader/writer locks before every shared memory access. Lock acquisition failure in this model corresponds to an ownership fault in page-grained CREW-based approach. However, we handle these failures within the guest which is much faster (less than 100 instructions) than page-grained CREW (thousands of instructions).

We have implemented our ideas inside the KVM hypervisor to record and replay a guest VM. We assume a Linux guest, and use a Linux-specific DBT driver (implemented as a loadable module for unmodified Linux). The DBT driver is configured to interpose on guest's memory accesses and communicates with the hypervisor to generate/consume the recorded log of non-deterministic events, during record/replay respectively. We have tested our implementation on several workloads including workloads that extensively exercise kernel code. Our experiments indicate the feasibility of implementing efficient DBT-based deterministic replay for unmodified guest code. Our typical recording overheads range between 15-273% on four processor. This is 11x better on average, than the best known comparable approach.

The thesis is organized as follows. Chapter 3.7 discusses related work. Chapter 3 discusses our DBT algorithm, implementation and results. Chapter 4 describes our two main algorithms, namely code-level and data-level mutual-exclusion. It also discusses our optimizations, implementation and results. Chapter 5 concludes.

Chapter 2

Related Work

Several hardware and software approaches for deterministic replay have been proposed. Deterministic replay can be implemented for a process or for the full system. In a uniprocessor environment, to deterministically replay a process we need to record the process address space at the beginning, any data the kernel writes into process address space, the return value of system calls, the value of special instructions like rdtsc, etc. Whereas for full system uniprocessor deterministic replay we need to record the ioport reads, interrupts and their timings, network packets, the value of special instructions like rdtsc, etc. The overhead of recording these events is typically less than 20% as demonstrated by several works [15, 24, 26, 30, 58, 61, 66]. However, multiprocessor deterministic replay is hard because of need to record the order of shared memory reads and writes. There are many software-only approaches to capture this non determinism and we will discuss them throughout this chapter.

Instant Replay [41] logs the order of all shared-memory accesses during an execution. Instant replay implements a CREW (concurrent read, exclusive write) protocol to record the ordering of shared accesses. The CREW protocol allows concurrent reads to a shared object, but allows write to a shared object only if no other thread is accessing that shared object. To implement the CREW protocol, Instant Replay acquires a reader/writer lock before every read/write access. It maintains a version number corresponding to each shared object. Whenever a process writes to a shared object the version number of the shared object is incremented. For every shared access, the version number of the shared object is recorded such that during replay the process accesses the same version of the object. Instant replay inserts `reader_entry`, `reader_exit` routines around every read access and `writer_entry`, `writer_exit` routines around every write access as shown in Figure 2.1. The instrumentation overhead of Instant Replay is high — the amount of instrumentation code that they insert around every shared-memory access is large, and the frequency of the execution of these shared memory accesses can impair performance. Our scheme is similar to Instant Replay, except that we have drasti-

```
reader_entry(object, process)
{
  if (record) {
    down(object.sema);
    atomic_add(object.active_readers, 1);
    up(object.sema);
    write_log(process, object.version)
  } else if (replay) {
    recorded_version = read_log(process);
    while(object.version != recorded_version);
  }
}

reader_exit(object)
{
  if (record || replay) {
    atomic_add(object.total_readers, 1);
    if (record) {
      atomic_dec(object.active_readers, 1);
    }
  }
}

writer_entry(object, process)
{
  if (record) {
    down(object.sema);
    while(object.active_readers != 0);
    write_log(process, object.version);
    write_log(process, object.total_readers);
  } else if (replay) {
    recorded_version = read_log(process);
    while(object.version != recorded_version);
    total_readers = read_log(process);
    while(object.total_readers < total_readers);
  }
}

writer_exit(object)
{
  if (record || replay)   {
    object.total_readers = 0;
    if (record) {
      object.version++;
      up(object.sema);
    } else {
      atomic_add(object.version, 1);
    }
  }
}
```

Fig. 2.1 Instant replay implementation of reader/writer locks.

cally reduced the amount of instrumentation code required around each memory access (only two extra instructions are executed per shared-memory access in the common case). Also, we present a scheme to decide the granularity and placement of these instrumented routines to optimize performance.

iDNA [12] logs the value returned by every load instruction. Since the frequency of load instructions is very high it is impractical to record every load value. iDNA maintains a per thread cache of load values. Whenever the thread accesses a memory location the cache is also updated. On every load to a memory location iDNA first checks the cached value with the current value, if they are not the same – possible if it is the first load, a kernel mediated control flow or DMA overwrites the memory location, another thread overwrites the memory location – it logs the load value, otherwise it increments the hit counter. Before logging the load value in case of cache conflict it also logs the value of the hit counter to correctly identify the load during replay. Recording the load values is sufficient to correctly replay the recorded sequence, however the ordering of threads may vary during replay. This property is also called *value-determinism*.

SMP-Revirt [27] maintains and logs page-grained read/write ownership for processors. Inspired by the CREW (concurrent read, exclusive write) protocol [22, 41], every page is assigned an owner for exclusive access, or a set of owners for shared read accesses. Through page table manipulations, a page is mapped only in the address space of its owners. Accesses by a processor to a page that it owns, execute at full speed. If a processor tries to access a page that it does not currently own, a page fault is generated; the fault handler transfers ownership, creates the new page table mappings, and records this ownership transfer event. This log of ownership transfers is enough to deterministically reproduce the execution. The SMP-Revirt implements deterministic replay for the virtual machine. It uses hardware page protection techniques on shadow page tables to implement CREW. The SMP-Revirt approach treats the target program as a black-box, and therefore it is possible to use this approach on any target program, including an OS kernel. However, every recorded event involves an expensive page fault and a page table update, making the approach perform very poorly on programs which involve a high degree of sharing, especially false sharing. For example, in our experiments running the Apache webserver, using the SMP-Revirt approach on the Linux kernel slows execution by around 125% on two processors. Further, as also discussed in [27], this scheme scales very poorly with increasing number of processors.

Scribe [38] implements CREW in kernel to do process-level deterministic replay using thread-private page tables. Pages in the thread-private page tables are mapped and unmapped based on the CREW protocol. The Scribe authors evaluate their scheme on a variety of benchmarks like Apache webserver, Linux build, etc. Compared to SPLASH2 benchmarks [64] (used in other work on user-level deterministic replay), the benchmarks used in this work exhibit relatively less sharing at the user-mode. Also, some of these benchmarks spend a large amount of execution time in the kernel, which is not getting recorded. Our work records and replays both process-level and kernel-level execution. While Scribe imposes less than 2.5% overhead for Apache on four processors, our own SMP-Revirt implementation for a virtual machine (full-system) imposes 20x overhead on four processors! Almost all overhead in this benchmark is observed in kernel-mode execution.

Instant replay [41], SMP-Revirt [27], and Scribe [38] are *order-based* replay systems because they record the order of shared memory accesses.

Subsequent work has addressed the performance limitations of SMP-Revirt in several ways. One way is to limit the scope of the programs being recorded; for example, RecPlay [57] assumes data-race free programs and works by recording only explicit synchronization operations. Kendo [52] also assumes data-race free programs but it makes the implementation of locks deterministic such that there is no need to record the order of lock acquisitions. In practice, real programs contain data races and hence these techniques can not be applied to them.

Another way to improve performance is to relax the definition of replay, so that a replayed run need not mimic the recorded run precisely, but should reproduce its *interesting* behaviour. Probabilistic Replay with Execution Sketching (PRES) [55] and Output-Deterministic Replay (ODR) [4] are examples of such systems, where the behaviour of the recorded run (e.g., failure due to a bug) is reproduced in the replayed run, even though the replayed run may not be identical to the recorded run. Such systems model replay as a "guided search" over the space of possible schedules that match the behaviour of the recorded run. These systems also rely on the ability to identify and interpose on all synchronization operations — for performance, if not for correctness. For example, the search space during replay increases exponentially with the number of data races in both these systems. For systems like OS kernels involving a wide variety of hidden synchronizations (which will appear as data races to these tools), these schemes become impractical.

ODR [4] satisfies the *output-determinism* property in which inputs during the replay may vary from the recorded run but the output remains the same during replay. For example, if the visible outputs are assertion failures, segmentation faults, crashes, etc; then ODR ensures that they must be visible during the replay even if the sequence of inputs or instructions are not the same as the recorded run. This relaxes the need to reproduce the same data race values during the replay, which improves record performance.

In both PRES and ODR there is a trade-off between record and replay performance. The less information you record, the search space during replay will increase accordingly. For example if the PRES only records the order of synchronization operations and system calls, it is not able to successfully replay all benchmarks. However it is able to replay all the benchmarks if it records the order of function calls. The overhead of recording function calls varies between 7%-779%. Similarly if ODR only records the lock order, the overhead lies between 10%-60% for two processors. If it also records the branches the overhead varies between 250%-450% for two processors. Notice that for the low overhead recorded run ODR is not able to replay all benchmarks.

In subsequent work, researchers have combined these ideas. For example, ReSpec [42] combines selective logging with output-deterministic replay in an online manner. Here, execution is sliced into time intervals, and only explicit synchronization operations are recorded and replayed. At the end of a time interval, the execution states of the recorded and replayed executions are compared and a rollback is triggered in case of a mismatch — a mismatch can occur if the replay failed due to non-determinism caused by data-races, for example. On a rollback, the execution of that time interval is serialized, and the serial order recorded. Assuming data-races are rare, ReSpec presents a fast deterministic replay system. Respec supports only “online” replay, i.e., the replayed and recorded processes must execute concurrently.

Subsequent work, DoublePlay [62], extends ReSpec to support offline replaying. DoublePlay assumes the presence of an online replaying session — the recording session executes in a “thread-parallel” fashion while the replaying session executes in an “epoch-parallel” fashion. The epoch parallel execution serializes the thread executions within a time interval, and yet provides throughput comparable to thread-parallel executions. Like ReSpec, DoublePlay compares the execution states at the end of every time interval, and rolls back in case of mismatch. If the states match at the end of a time interval, the serial execution order of the epoch-parallel execution and the recorded order of synchronization operations is sufficient to guarantee correct offline replay. It is possible that the number of rollbacks for an epoch is very high due to large number of data races in that epoch, in that case if the divergence check fails at the end of the epoch the DoublePlay copies the epoch parallel state to the thread parallel state and start execution from the new state. If the divergence is detected in the middle of an epoch — for example the arguments of a system call did not match — then it rolls back to the point of divergence and then copies the state of the epoch-parallel execution to the state of the thread-parallel execution (called *forward recovery* in their paper).

RecPlay is not able to replay `radiosity` and `dbench` because of data races. Notice that RecPlay is able to replay all the SPLASH2 benchmarks ran by DoublePlay; it means these benchmarks don’t have any data races and recording the order of synchronization operations is sufficient to replay them. Due to this DoublePlay didn’t observe any rollbacks for these benchmarks. The performance impact of DoublePlay on `dbench` and `radiosity` will surely give us a clear picture about their performance.

While ReSpec and DoublePlay present low overheads on the application-level benchmarks they evaluated, there are shortcomings to these approaches that make them impractical for use in recording and replaying full systems:

1. They rely on the ability to rollback an execution, which requires maintaining multiple versions of the same state, and extra copying which is quite expensive. Respec and Doubleplay implement page-grained copy-on-write optimizations; while such optimizations may work for applications with relatively small memory footprints, they are impractical for workloads like a full guest kernel, whose memory footprint is much larger.

2. They rely on being able to understand and accordingly interpose on all synchronization operations; as we have discussed before, this is quite hard to do correctly for an OS kernel. Also, this requires the ability to modify the target program.

3. Most importantly, these techniques rely on the rarity of data races. If data races are common, rollbacks are triggered frequently which severely impairs performance. For example, in all the experiments reported in the DoublePlay paper [62], only at most two rollbacks are triggered per program execution; even with the forward recovery, rollbacks were not avoided for some benchmarks; for an OS kernel which deliberately uses data races to implement many types of implicit synchronization, the expected number of rollbacks will be several orders of magnitude higher.

The overhead of ReSpec and DoublePlay varies between 4-100% for SPLASH2 benchmarks on two processors. Interestingly, on the benchmarks used by ReSpec and DoublePlay, SMP-Revirt also exhibits less than 100% overheads on two processors. The two benchmarks on which SMP-Revirt performs badly (7-9x overheads on `dbench` and `radiosity`), have not been evaluated in ReSpec and DoublePlay. Previous work on RecPlay [57] reported that `dbench` and `radiosity` contain data races, and so their scheme could not handle them. We believe that these benchmarks are unlikely to perform well on ReSpec and DoublePlay for this reason.

Neither DoublePlay nor ReSpec record the execution of the OS kernel; they simply log the outputs of the system calls. Thus these systems cannot help in debugging the OS kernel. On the other hand, we record the entire software stack, including the applications and the OS kernel.

Related work on determinizing executions of a non-deterministic program [9, 10, 23, 25, 43], or enforcing determinism at the system level [6] are competing approaches to deterministic replay. Given that current systems are deliberately non-deterministic, deterministic replay is often the most practical and immediate solution.

Another work on scalable deterministic replay in a parallel full-system emulator achieves performance overheads of around 70% for an emulator running on 16 processors [20] — because a full-system emulator already has large overheads due to the use of a software MMU, the overheads of supporting deterministic replay seem small in comparison. However, the ideas presented in this work do not translate directly to supporting efficient deterministic replay on bare-metal executions. While several hardware optimizations have also been proposed for deterministic replay [7, 33, 34, 45, 47, 50, 63, 65], we restrict our attention to software-only approaches in this thesis.

Chapter 3

Fast Dynamic Binary Translation for the kernel

3.1 Introduction

DBT is a technique which transforms the code as it executes. DBT can be implemented both at user-level [17] and at system-level [2, 29]. Current system-level binary translators exhibit large performance overheads on kernel-intensive workloads. For example, VMware's binary translator in a Virtual Machine Monitor (VMM) shows 10x slowdowns for the `syscall` nanobenchmark [2]; corresponding overheads are also observed in macrobenchmarks. VMware's DBT performance also includes the overheads of other virtualization mechanisms, like memory virtualization through shadow page tables, etc. Another kernel-level binary translator, DRK [29], reports 2-5x slowdowns on kernel intensive workloads. Applications requiring high kernel activity (like high performance fileservers, databases, webservers, software routers, etc.) exhibit prohibitive DBT slowdowns, and are thus seldom used with DBT frameworks. Since the overhead of DBT is itself very high it is not used for kernel optimizations.

Ideally, a translated system must run at near-native speed. Low-overhead user-level DBT is well understood [16]; kernel-level translation involves correctly handling interrupts, exceptions, CPU concurrency, device/hardware interfaces, and is thus more complex and expensive.

We implemented a kernel-level dynamic binary translator with near-native performance. Like DRK [29], our translator works for the entire kernel including arbitrary devices and their drivers. This is in contrast with virtual-machine based approaches (e.g., VMware [2], PinOS [19], BitBlaze [60]), where translation is only performed for code that runs in a virtualized guest.

As also discussed by DRK authors [29], making dynamic binary translation work for arbitrary devices and drivers is important because drivers constitute a large fraction of kernel code, and most of this remains unexercised in a virtual machine. Moreover, most interesting program behaviour (e.g., bugs, security issues, etc.) occurs in drivers. Further, many workloads are incapable of running in virtual environments due to device constraints. We evaluate the performance of our DBT framework on a number of workloads, and show significant improvements over previous work.

Our translator is implemented as a loadable kernel module and can attach to a running OS kernel, ensuring that all kernel instructions run translated thereafter. It does not translate user-level code. Our translator exhibits performance *improvements* of up to 17% over native on certain workloads. Similar improvements have previously been observed for user-level binary translators [16], and have been attributed to improved caching behaviour, especially at the instruction cache. Our translator can be detached from a running system at will, to revert to native execution.

Like previous work [2, 29], our translator provides full kernel code coverage, preserves original concurrency and execution interleaving, and is "transparent" to the kernel. i.e., kernel code does not behave differently or break if it observes the state of the instrumented system. While VMware promises complete transparency, both DRK and our translator have transparency limitations, i.e., it is in general possible for kernel code to inspect translated state/data structures. We only ensure that this does not result in incorrect behaviour during regular kernel execution[1]. Our design differs from VMware and DRK in the following important ways:

Entry Points: We replace kernel entry points (interrupt and exception handlers) directly with their translated counterparts. This is in contrast with VMware and DRK which replace kernel entry points with calls to the DBT's dispatcher (see Figure 3.1 for the component diagram of a dynamic binary translator), which in turn jumps to the translated handlers *after* restoring native state on the kernel stack. This extra work by DBT dispatcher causes significant overhead on each kernel entry. In our design, kernel entries execute at near-native speed. In doing so, we allow the kernel handlers to potentially observe the non-native state generated by the hardware interrupt on stack.

Interrupts and Exceptions: Previous DBT solutions (VMware, DRK) handle exceptions by emulating precise exceptions[2] in software by rolling back execution to the start of the translation of the current native instruction; and handle interrupts by delaying them till the

[1] Actually, VMware's binary translator also does not guarantee full transparency. For example, they do not translate user-level code for performance. Most "well-behaved" operating systems work well in this model, but an adversarial guest can expose their transparency limitations.

[2] A precise exception means that before execution of an exception handler, all instructions up to the executing (emulated) instruction have been executed, and the excepting instruction and everything afterwards have not been executed. Previous DBT implementations have preserved precise exception behaviour for architectures that support precise exceptions (e.g., x86).

start of the translation of the next native instruction. These mechanisms are complex and expensive, and are primarily needed to ensure that the interrupt handlers observe consistent state. In our design, we allow imprecise exceptions and interrupts. Relaxing precision greatly simplifies design and improves performance. In our experience, operating systems rarely depend on precise exception and interrupt behaviour.

Reentrancy and Concurrency: The translator's code and data structures need to be reentrant to allow interrupts and exceptions to occur at arbitrary program points. Similarly, physical CPU concurrency needs to be handled carefully. DBT requires maintenance of CPU-private data structures, and migration of a thread from one CPU to another should not cause unsafe concurrent access to common state. In our design, the presence of imprecise exceptions and interrupts introduces more reentrancy and concurrency challenges. We present an efficient mechanism to provide correct translated execution.

Our optimizations result in a very different translator design from both VMware and DRK. We are more aggressive about assumptions on usual kernel behaviour. In doing so, we sometimes relax "transparency"; for us, ensuring *correctness* is enough. Essentially, we show that many transparency requirements are unnecessary and can be relaxed for better performance.

3.2 DBT Background

We first introduce the terminology and provide a basic understanding of how a dynamic binary translator works (also see Figure 3.1). We refer to the terms and concepts described in this section, while discussing our design and optimizations in the rest of the chapter. We call the kernel being translated, the *guest* kernel. Starting at the first instruction, a straight-line native code sequence (*code block*) of the guest is translated by the *dispatcher*. A code block (also called a *trace* in previous work) is a straight-line sequence of instructions which terminates at an unconditional control transfer (branch, call, or return). The instructions in a block are translated using a *translation rulebook*. For quick future access, the translations are stored in a *code cache*. The dispatcher translates one code block at a time and transfer control to it. The dispatcher ensures that it regains control when the block exits by replacing the terminating control flow instruction by a branch back to the dispatcher after appropriately setting the next native PC (called `nextpc`). The dispatcher looks up the code cache to search if a translation for `nextpc` already exists. If so, it jumps to this translation. If not, the native code block beginning at `nextpc` is translated and the translation is stored in the code cache before jumping to it. We call the translated code corresponding to `nextpc`, `tx-nextpc`.

To improve performance, *direct branch chaining* is used, i.e., before the dispatcher jumps to a translation in the code cache, it checks if the previous executed block performed a direct

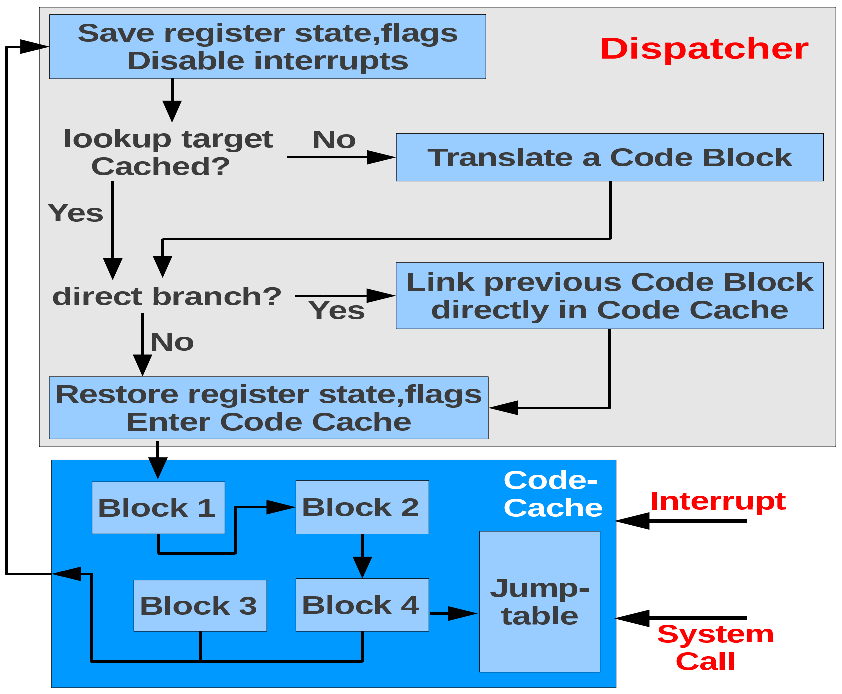

Fig. 3.1 Control Flow of a Dynamic Binary Translator.

```
         jmp .edge0
.edge0:  save_registers_and flags
         clear interrupts
         set nextpc
         jump to dispatcher
```

Fig. 3.2 The translated (pseudo) code generated for a direct unconditional branch. After the first execution of this code, the first “`jmp .edge0`” instruction is replaced with “`jmp tx-nextpc`” to implement direct branch chaining.

branch to this address. If so, the corresponding branch instruction in the previous executed block is replaced with a direct jump to the translation of the current program counter. This allows the translated code to directly jump from one block to another within the code cache (without exits to the dispatcher), thus resulting in near-native performance.

Figure 3.2 shows the translation code for a direct unconditional branch, to illustrate the direct branch chaining mechanism. At the first execution of this translated code, the operand of the first `jmp` instruction is the address of the following instruction (`.edge0`). The code at `.edge0` sets `nextpc` before jumping to the dispatcher. After the first execution, the dispatcher replaces the first instruction with “`jmp tx-nextpc`” to implement direct branch chaining.

If a code block ends with an indirect branch, `nextpc` can only be determined at runtime. As an optimization, a fast lookup table is maintained to convert `nextpc` to `tx-nextpc` without

```
save flags and clear interrupts
save temporary regs %tempreg0 and %tempreg1
mov *MEM, %tempreg0
%tempreg1 := jumptable_hashfn(%tempreg0)
index %per_cpu:jumptable using %tempreg1
if jumptable hit,
    restore flags and jump to tx-nextpc
else,
    jump to dispatcher
```

Fig. 3.3 Translation for the indirect branch instruction "`jmp *MEM`", which looks up the jumptable to convert `nextpc` to `tx-nextpc`. As discussed in Section 3.4.1, a separate per-CPU jumptable is maintained, and "`%percpu:jumptable`" obtains the address of the jumptable of the currently executing CPU. `jumptable_hashfn()` represents the jumptable's hash function. On Linux, the `%fs` segment stores the value of the `%per_cpu` segment and is used to store CPU-private variables (like the jumptable in this case). Section 3.4.1 also discusses the need to clear interrupts before a jumptable lookup.

having to exit into the dispatcher. The lookup table, called *jumptable*, is implemented as a small hashtable. Additions to the jumptable are done in the dispatcher, and lookups are done using assembly code (emitted in the code cache for every indirect branch). Figure 3.3 shows the pseudo-code of the translation of an indirect branch.

3.2.1 Kernel-level DBT Background

Kernel-level DBT requires more mechanisms to correctly handle interrupts, exceptions, reentrancy and concurrency issues. Interposition on kernel execution is ensured by replacing all kernel entry points (interrupt and exception handlers) with custom handlers. In previous work, these entry points have been replaced with a call to the DBT dispatcher. The dispatcher receives as argument, the original PC value at the entry point. Before the dispatcher translates and executes the handler at this PC value, it performs more work as discussed below. (As we discuss in Section 3.3, we avoid this extra work in our design).

PC value pushed on the interrupt stack by hardware is translated to its native counterpart by the dispatcher on every interrupt/exception. The value pushed on stack by hardware is the PC value at the time of the interrupt/exception. This value could be a code cache address or a dispatcher address. In either case, the value is replaced with the address of the *native instruction* that must run after the handler finishes execution. Consequently, the return-from-interrupt instruction (`iret`) is translated to obtain `nextpc` from stack and exit to the dispatcher.

If a synchronous exception has occurred in the middle of the translation of an instruction, **precise exception behaviour is emulated**. The procedure requires *rolling back* machine state to its state at the start of the (translation of the) current native instruction. The code to implement this rollback must be provided in the translation rulebook, and is executed in the context of the dispatcher. After executing the rollback code and putting the native instruction address on stack, the exception handler is executed.

If an asynchronous interrupt was received, the **delivery of this interrupt is delayed** until the (translation of the) next native instruction boundary. This is done to ensure *precise interrupts*, i.e., the interrupted native instruction must never be seen "partially executed" by the handler. This delayed interrupt delivery is implemented by patching the translation of the next native instruction with a software-interrupt instruction (to recover control at that point). After recovering control, the interrupt stack is setup to return to this next instruction before executing the interrupt handler.

These mechanisms are discussed in detail in the DRK paper [29] and also previously in a VMware patent [18]. These mechanisms are expensive, as we discuss next:

First, replacing the PC value pushed by hardware on the interrupt stack, to its native counterpart, on each interrupt, results in significant overhead for an interrupt-intensive application. Similarly, the translation code for the `iret` instruction adds overhead on every return from interrupt.

Second, the rollback operation required to ensure precise exceptions is expensive. There is a direct cost of executing the rollback code on each exception. But more significantly, there is an indirect cost of having to structure a translation in a way that it can be rolled back. Typically, this involves making a copy of the old value of any register that is being overwritten. This cost is incurred on the straight-line non-exceptional execution path on every execution of that instruction, and is thus significant.

Third, the delaying of interrupts involves identifying the next native instruction, patching it, incurring an extra software trap, and then patching the interrupt stack. These are expensive operations.

In our work, we show that a guest kernel rarely relies on the PC value being pushed on stack on an interrupt/exception, and is largely indifferent to imprecise exception and interrupt behaviour, and thus these overheads can be avoided for a vast majority of DBT applications.

3.3 A Faster Design

In our design, we do not ensure precise exceptions and interrupts. We also do not guarantee that the PC value on the interrupt stack is a valid native address. We simply allow the PC value pushed by hardware to get exposed to the interrupt handler. We also allow the interrupt handler to inspect intermediate machine state if an interrupt/exception occurred in the middle of the translation of a single instruction.

The design is simple. We disallow interrupts and exceptions in the dispatcher (see Section 3.4.1 for details). Thus, interrupts and exceptions only occur while executing in the code cache, or while executing in user mode.

We replace a kernel entry point with the translation of the code block at the original entry point. This causes an interrupt or exception to directly jump into the code cache (see Figure 3.1). Consequently, we use the identity translation for the `iret` instruction (i.e., `iret` in native code is translated to `iret` in translated code) to return back directly to the code cache. The system thus executes at full speed. But we need more mechanisms to maintain correctness.

The first correctness concern is whether an interrupt or exception handler could behave incorrectly if it observes an unexpected PC value on the interrupt stack. Fortunately, in practice, the answer is no, barring a few exceptions. For example, on Linux, only the page fault handler depends on the value of the faulting PC. The Linux page fault handler uses the faulting PC value to check if the fault is due to a permissible operation (like one of the special `copy_from_user(), copy_to_user()` functions) or a kernel bug. To implement this check, the kernel compiler generates an "exception table" representing the PCs that are allowed to fault and the faulting PC is searched against this table at runtime. With DBT, because the code cache addresses will not belong to this table, the page fault handler could incorrectly panic.

Similar patterns, where certain exception handlers are sensitive to the excepting PC value, are also found in other kernels. For example, on some architectures (e.g., MIPS), *restartable atomic sequences* (RAS) [11] are implemented to support fast mutual exclusion on uniprocessors. RAS code regions, indicating critical sections, can be registered with the kernel using PC start and end values. If a thread was context-switched out in the middle of the execution of a RAS region (determined by checking the interrupted PC against the RAS registry), the RAS region is "restarted" by the kernel by overwriting the interrupt return address by the start address of the RAS region. With DBT, this mutual-exclusion mechanism could get violated because the code cache addresses will not belong to the RAS registry. Also, kernels implementing RAS can cause execution of native code as they could potentially overwrite the interrupt's return address with a native value. A similar pattern involving overwriting of the interrupt return address by the handler is also present in the BSD kernels, namely FreeBSD, NetBSD, and OpenBSD. The pattern is shown in Figure 3.4. As explained in the figure, this is done to allow kernel subsystems to install custom page fault handlers for themselves. As another example of a similar pattern, Microsoft Windows NT Structured Exception Handling model supports a `__try/__except` construct which registers the exception handler specified by the `__except` keyword with the code in the `__try` block. These constructs are implemented by maintaining per-thread stacks of exception frames; on entry to a `__try/__except`

```
void function_that_can_cause_page_fault()
{
  /* by default, pcb_onfault = 0. */
  push pcb_onfault;
  pcb_onfault = custom_page_fault_handler_pc;

  /* code that could page fault. */

  pop pcb_onfault;
}

void kernel_page_fault_handler()
{
  /* handler invoked on every page fault. */
  if (pcb_onfault) {
    intr_stack[RETADDR_INDEX] = pcb_onfault;
  }
}
```

Fig. 3.4 Pseudo-code showing registry of custom page fault handlers by kernel subsystems in BSD kernels. The `pcb_onfault` variable is set to the PC of the custom page fault handler before execution of potentially faulting code. On a page fault, the kernel's page fault handler overwrites the interrupt return address on stack with `pcb_onfault`.

block, an exception frame containing the exception handler pointer is pushed to this stack and on function return, this exception frame is popped off the stack. If an exception occurs, the kernel's exception handler (e.g., page fault handler) traverses this exception stack top-to-bottom to find and execute the appropriate `__except` handler [3]. Because on an exception inside the `__try` block, the kernel's exception handler overwrites the excepting PC, our DBT design can incorrectly cause execution of native untranslated code.

Fortunately, such patterns are few, and can be usually handled as special cases. On Linux for example, the kernel allows loadable modules to register custom exception tables at load time, to extend similar functionality to loadable modules. On a page fault, the faulting PC is also checked against the modules' exception tables. For our DBT implementation, we ensure that the code cache addresses corresponding to the functions that already existed in kernel's exception table belong to our module's exception table. This ensures correct behaviour on kernel page faults. Similarly, DBT for kernels implementing RAS can be handled by manip-

[3] On non-x86 architectures (e.g., ARM, AMD64, IA64), a somewhat different implementation for `__try`/`__except` is used. A static exception directory in the binary executable contains information about the functions and their `__try`/`__except` blocks. On an exception, the call stack is unwound and the exception directory is consulted for each unwound frame to check if a handler has been registered for the excepting PC.

ulating the RAS registry to also include the translated RAS regions. The exception directory in Microsoft Windows for non-x86 architectures can be handled similarly. Further, to avoid execution of native code after interrupt return, due to overwriting of return address by a handler (e.g., custom page fault handler installation in BSD kernels), the `iret` instruction can be translated to also check the return address; if the return address does not belong to the code cache, indicating overwriting by the handler, the translator should jump to the dispatcher to perform the appropriate conversion to its corresponding translated code cache address[4].

In general, we believe that for a well-designed kernel, any interrupt or exception handler whose behaviour depends on the value of the interrupted PC value, should ideally also allow a loadable module to influence the handler's behaviour, because the PC values of the module code are only determined at module load time. For example, Linux provides the module exception table for page fault handling. This allows a DBT module to interpose without violating kernel invariants. In cases where such interposition is not possible, our DBT design will fail.

In some kernels, we also found instances where an excepting PC address is compared for equality with a kernel function address in the exception handler. These checks against hardcoded addresses (as opposed to a table of addresses as in Linux), pose a special problem, as it is no longer possible for the DBT module to manipulate these checks. Fortunately, such patterns are rare, and are primarily used for debugging purposes. If such patterns are known to exist, special checks can be inserted at interrupt entry (by appropriately translating the first basic block pointed to by the interrupt descriptor table) to compare the interrupted PC pushed on stack against translations of these hardcoded addresses. If found equal, the PC pushed on stack should be replaced by their corresponding native code address. Similar checks should be added on interrupt return with appropriate conversion from native address to its translated counterpart, if needed. Notice that these special-case checks are much cheaper than translations from native addresses to code cache addresses and vice-versa on every interrupt entry and return respectively, as done in previous designs.

Table 3.1 summarizes our survey findings regarding the use of the interrupted PC address on stack in various kernels. In summary, we allow fast execution of the common case (where interrupted PC value is not read or written), and use special-case handling for the few design patterns where the PC value is known to be read/written in unconventional ways.

The second correctness concern has to do with the presence of code cache addresses in the kernel's data structures. For example, if an interrupt occurs while the translated kernel is executing in the code cache, the code cache address would be pushed on the kernel stack. If the executing thread then gets context-switched out, the code cache address would continue to

[4] If the code cache is allocated in a contiguous address range, this translation of `iret` to check the return address is cheap (4-8 instructions). This is much faster than converting native addresses to translated addresses on every interrupt return, as done in previous DBT designs.

OS	**Unconventional uses of the interrupted/excepting PC value pushed on stack by hardware**
Linux	Found one check against a table of addresses (exception table) in page fault handler.
MS Windows	`__try()/__catch()` blocks implemented by maintaining per-thread stacks of exception frames.
FreeBSD	Found three equality checks against hardcoded function addresses. Found two more uses for debugging purposes. Implements RAS. Overwrites return address to implement custom page fault handlers.
OpenBSD	Implements RAS. Overwrites return address to implement custom page fault handlers.
NetBSD	Found two uses for debugging purposes. Implements RAS. Overwrites return address to implement custom page fault handlers.
BarrelFish	Found no such use.
L4	Found two equality checks against hardcoded function addresses in page fault handler.

Table 3.1 Unconventional uses of the interrupt return address (in ways that need special handling in our DBT design) found in the kernels we studied.

live in the kernel data structures. If the code cache address becomes invalid in future (due to cache replacement, for example), this can cause a failure.

To solve this problem, we ensure that code cache addresses do not become invalid until they have been removed from all kernel data structures. Firstly, we disallow cache replacement; we assume that the space available for code cache is sufficient to store translations of *all* kernel code. This is not an unreasonable assumption; for example, we use a code cache of 10MB which is sufficient for the Linux kernel, whose entire code section (including code of loadable modules) is typically less than 8MB in size. There may be corner cases, where the size of the code cache may exceed the available space (for example, due to repeated loading and unloading of modules); we discuss how to handle such situations in Section 3.4.4. Secondly, once a code block is created in the code cache, we do not move or modify it (except the one-time patch for direct branch chaining). This ensures that a code cache address, once created, remains valid for the lifetime of the translated execution. Further, translator switchoff needs to be handled carefully — all code cache addresses should be removed from kernel data structures before effecting a switchoff (see Section 3.4.4).

The third correctness concern is regarding violation of precise exception and interrupt behaviour. Interestingly, none of the kernel exception handlers we encountered, depend on precise exception behaviour. In practice, the kernel exception handlers at most examine the contents (opcode and operands) of the instruction at the faulting PC to make control flow decisions. As long as the translated code does not cause any extra exceptions or does not suppress any exception that would have occurred in native code, the system behaves correctly. Similarly, the kernel never depends on the PC value for interrupt handling and is thus indifferent to violation of precise interrupts. The handlers are also indifferent to the values of other registers (that are not used by the faulting instruction) at interrupt/exception time[5].

3.4 Design Subtleties

3.4.1 Reentrancy and Concurrency

Consider a CPU executing inside the dispatcher. An interrupt or exception at this point could result in a fresh call to the dispatcher, to translate the code in the interrupt/exception handler. Similarly, consider a CPU executing in the code cache. The translated code for an instruction could have multiple instructions and could potentially be using temporary memory locations (scratch space) to store intermediate values (e.g., our translation of the indirect branch instruction uses two scratch space locations). An interrupt/exception in the middle of the translated code could result in race conditions on accesses to this scratch space. We call these reentrancy problems.

The other reentrancy problems we faced were the same as those encountered in previous work [29]. As also noted by DRK, it is incorrect to call kernel routines from within the dispatcher. It is possible for the dispatcher to be in the middle of translating a kernel routine, and another call to the same routine from within the dispatcher could result in incorrect behaviour. The most common bug of this type is a deadlock where a lock inside the routine is attempted to be acquired twice by the same thread (first by the kernel's call to it, and second by the

[5]System calls depend on register values, but they are implemented as software exceptions in user mode, and we are discussing hardware exceptions/interrupts received in kernel mode; interrupts/exceptions received in user mode are irrelevant to our design as DBT does not influence user-mode behaviour.

```
restore guest registers/stack
restore guest flags (interrupts may get enabled)
jmp *%per_cpu:tx-nextpc-loc
```

Fig. 3.5 The code to exit the dispatcher and enter the code cache at `tx-nextpc` (which is the value stored in `tx-nextpc-loc`). As discussed later, `tx-nextpc-loc` is a per-cpu location accessed using the per-cpu segment.

dispatcher's call). Hence, to avoid having to call `malloc()` from within the dispatcher, we preallocate memory for the dispatcher's data structures (including code cache). We also do not call any kernel I/O functions from the dispatcher (to debug our translator, we ran the OS in a virtual machine and used a hypercall into the VMM to print debugging information to the console).

In previous work, reentrancy problems were simplified because their designs ensured precise exceptions and interrupts. An interrupt or exception was serviced only after the state of the system reached a native instruction boundary; this meant that a handler (or a dispatcher call made by it) never observed intermediate state. Our design is different, and we discuss the resulting challenges and their solutions.

Firstly, we disallow interrupts and exceptions inside dispatcher execution. Exceptions are disallowed by design; none of the dispatcher instructions are expected to generate exceptions, and page faults are absent because all kernel code pages are expected to be mapped. We also never interpret kernel code within the dispatcher. Interrupts are disallowed by clearing the interrupt flag (using `cli` instruction) before entering the dispatcher. To avoid clobber, the kernel's interrupt flag is saved on dispatcher entry and restored on dispatcher exit. (Notice the clearing of interrupt flag in Figures 3.2 and 3.3). Notice that the NMI interrupt cannot be masked and hence it can occur in the middle of the dispatcher. NMI interrupts are used for debugging purposes and our current implementation doesn't support it. We have tested our implementation with NMI handler running natively. In our experiments we completely disabled NMI for the full system.

At dispatcher exit (code cache entry), the kernel's flags need to be restored inside the dispatcher before branching to the code cache. This presents a catch-22 situation: restoring kernel flags could cause interrupts to be enabled and thus to preserve reentrancy, there should not be any accesses to a dispatcher memory location after that; and yet we need some space to store `tx-nextpc` (the code cache address to jump to). Figure 3.5 shows the code at dispatcher exit. Notice that at the last indirect branch in this code, `tx-nextpc` cannot be stored in a register (because the registers are supposed to hold the kernel's values at this point), and cannot be stored on stack (because the stack should not be any different from what the kernel expects it to be). `tx-nextpc` is instead stored at a per-CPU location called `tx-nextpc-loc` (a

```
restore guest registers/stack
jmp *%per_cpu:tx-nextpc-loc
```

Fig. 3.6 The code to exit the dispatcher and enter the code cache at `tx-nextpc` irrespective of kernel design.

```
restore guest flags
```

Fig. 3.7 Prefix code at `tx-nextpc` the beginning of each code block. Only executes if called from dispatcher/ indirect branch lookup code.

per-CPU location in the kernel is a location that has separate values for each CPU; we discuss the need for `tx-nextpc-loc` to be per-CPU later in our discussion on concurrency). This code at dispatcher exit is non-reentrant because an interrupt after guest flags are restored and before the last indirect branch to `*tx-nextpc-loc` executes, could clobber `tx-nextpc-loc`.

To solve this problem, we save and restore `tx-nextpc-loc` at interrupt entry and exit respectively. Thus, this one dispatcher memory location has special status. The translation of the first block of all interrupt/exception handlers is augmented to save `tx-nextpc-loc` to stack, and the translation of the last code block before returning from an interrupt (identified by the presence of the `iret` instruction) is augmented to restore `tx-nextpc-loc` from stack. Because some of the interrupt state on stack is pushed by hardware (e.g., code segment, program counter, and flags), simply adding another `push` instruction at interrupt entry (to save `tx-nextpc-loc`) will not work, as that will destroy the interrupt frame layout on stack. On Linux, we identified a redundant location in the stack's interrupt frame structure, and used it to save and restore `tx-nextpc-loc` on interrupt entry and return respectively[6].

If a redundant stack location cannot be identified, saving the flags after jumping to the `tx-nextpc` will solve the issue. To do this instead of saving flags before jumping to the `tx-nextpc-loc`, in the dispatcher exit code, as shown in Figure 3.5, the dispatcher jumps directly to the `tx-nextpc-loc` as shown in Figure 3.6. A prefix code as shown in Figure 3.7 is prepended before the code block corresponding to `tx-nextpc`, which restores the guest flags. Since the interrupts are now being enabled after jumping to the target location there will be no reentrancy issue.

Indirect branch lookup code as shown in Figure 3.8, jumps to the `tx-nextpc` in the similar fashion as dispatcher. Here also, instead of restoring flags at line 8 in Figure 3.8, the lookup code jumps directly to the prefix code, which is prepended to the target basic block, as we did in case of dispatcher-to-guest entry. Since the indirect target always jumps to the prefix code,

[6]On Linux, the interrupt frame field to save and restore the `%ds` segment selector is redundant, because the value in `%ds` register is never overwritten by an interrupt/exception handler. Thus, we translate the instructions that save and restore `%ds` to instructions that save and restore `tx-nextpc-loc` instead.

```
save flags and clear interrupts
save temporary regs %tempreg0 and %tempreg1
mov *MEM, %tempreg0
%tempreg1 := jumptable_hashfn(%tempreg0)
index %per_cpu:jumptable using %tempreg1
if jumptable hit,
   restore temporary regs %tempreg0 and %tempreg1
   restore flags
   jmp *per_cpu:tx-nextpc-loc
else,
   jump to dispatcher
```

Fig. 3.8 Indirect branch lookup code. If the `tx-nextpc` is found in jumptable we store it into `tx-nextpc-loc`. The lookup code jumps to the target basic block after storing temporary registers and flags.

```
restore guest flags
jmp tx-nextpc
```

Fig. 3.9 Prefix stub code for every jump from dispatcher/ indirect branch lookup code.

the dispatcher needs to add only the addresses of prefix code in the per-CPU jumptable.

Notice that this design can handle the reentrancy problem successfully but it creates a new problem. In our design, we allow jump within the code block. Because the dispatcher can only jump to the prefix code, and the prefix code can only be added at the beginning of a code block, it can not jump to the middle of a code block. To handle this, the translator emits prefix stubs which branches to the destination code after restoring flags as shown in Figure 3.9. The dispatcher and indirect branch lookup code jump to these stubs, to jump at arbitrary location in a code block.

As we discuss later in Section 3.4.2, allocating memory for these prefix codes from a separate memory pool exhibits better instruction cache locality as they execute less frequently.

Next, we consider reentrancy problems due to interruption in the middle of a translation in the code cache. To address this, we need to ensure that accesses to scratch space (used in translated code) are reentrant. We mandate that any extra scratch space required by a translation rule should be allocated on the kernel's thread stack. The `push` and `pop` instructions are used to create and reclaim space on stack. Because a kernel follows the stack abstraction (i.e., no value above the stack pointer is clobbered on an interrupt), this ensures reentrant scratch space behaviour. Because typical space allocation for kernel stacks (8KB on Linux) is comfortably more than its utilization, there is no danger of stack overflow due to the small extra space used by our translator.

```
        cmp %reg1, %reg2
        jcc .edge0
        cmp %reg3, %reg4
        jcc .edge1
        mov %reg3, %reg2
        jmp .edge2
.edge0: save_registers_and flags
        clear interrupts
        set nextpc
        jump to dispatcher
.edge1: ... (similar to .edge0)
.edge2: ... (similar to .edge0)
```

Fig. 3.10 The translated (pseudo) code generated for a code block involving multiple conditional branches (`jcc`).

Finally, accesses to the jumptable need to be reentrant. In Figure 3.3 which shows the translated code of an indirect branch, consider a situation where the thread executing this translation gets interrupted after it has determined that `nextpc` exists in the jumptable and before it reads the value of `tx-nextpc` from its location. If the interrupt handler gets to run in between, its translation could cause addition of new entries to the jumptable, potentially replacing the mapping between `nextpc` and `tx-nextpc` (that has already been read). Now, when the interrupted thread resumes, it would read an incorrect `tx-nextpc` (because it had previously determined that `nextpc` exists in the table although it has been replaced now), causing a failure. We fix this problem by clearing the interrupt flag before executing the jumptable lookup logic, and restoring it before branching to `tx-nextpc`, as shown in Figure 3.3. This was the most subtle issue we encountered in our design.

To avoid concurrency issues arising from execution by multiple CPUs simultaneously, we maintain CPU-private data structures: the dispatcher executes on a CPU-private stack, all temporary variables are stored on the stack, and per-CPU jumptables for indirect branches are used. The dispatcher code is also reentrant and thread-safe (no global variables). The special `tx-nextpc-loc` variable is also maintained per-CPU. The only inter-CPU synchronization required is for mutual exclusion during addition of blocks to the shared code cache[7].

3.4.2 Code Cache Layout

Figure 3.10 shows an example of a code block with multiple conditional branches. Notice that `jcc` instructions initially point to the corresponding "edge" code (code which sets `nextpc` and branches to the dispatcher). On the first execution of this edge code, the target of the `jcc` instruction is replaced to point to a code cache address (direct branch chaining). After direct branch chaining, the code cache layout looks very similar to the native code layout, differing only at block termination points.

We experimentally found that the extra edge code introduced for each block results in poorer spatial locality for the instruction cache. This edge code is executed only once at the first execution of the corresponding branch, but shares the same cache lines as frequently executed code. We fix this situation by allocating space for the edge code from a separate memory pool. This allows better icache locality for frequently executed code in the code cache. In our experiments, we observed a noticeable performance improvement after this optimization. Allocating prefix code, as discussed in Section 3.4.1 from a separate memory pool is likely to provide similar benefits (Recall that we does not use prefix code for our experiments – we simply use the redundant location in the Linux kernel interrupt frame to achieve the same purpose).

We also found that multiple code copies resulting from CPU-private code caches result in poor icache behaviour. For this reason, we use a shared code cache among CPUs. This does not result in concurrency issues because instructions in code cache are read-only, except the one-time patching of branch instructions for direct branch chaining.

3.4.3 Function call/return optimization

Our design eliminates most DBT overheads; the biggest remaining overhead is that of indirect branch handling. Each indirect branch is translated into code to first generate `nextpc`, then lookup the jumptable in assembly, and finally, if the jumptable misses, branch to the dispatcher. Even if the jumptable always hits, 2-3x slowdowns are still observed on code containing a high percentage of indirect branches. The most common type of indirect branches are function returns (`ret` instruction on x86). We optimize by using identity translations for `call` and `ret` instructions. In doing so, we let a function call push a code cache address to the stack; at function return, the thread simply returns to the pushed code cache address. This optimization works because after the kernel has fully booted, the return address on stack is

[7]The cost of this synchronization is small because additions to the code cache are relatively rare in steady state. This synchronization could have been avoided by using multiple per-CPU code caches but that results in poor icache performance as also discussed in Section 3.4.2.

only accessed using bracketed call/return instructions. We find that this optimization yields significant improvements.

Because this optimization uses the identity translation for `ret`, all calls *must* push only code cache addresses. This poses a special challenge for calls with indirect operands. Indirect calls of the type "`call *REG`" and "`call *MEM`" are supported on the x86 architecture. Without the call/ret optimization, handling of these instructions is straightforward: the target address (`nextpc`) is obtained at runtime, the jumptable is searched, and if the jumptable hits, the address of the *native* return address is pushed to the stack, and a branch to `tx-nextpc` is executed. If the jumptable misses, the code still pushes the native return address to stack, sets `nextpc` and then exits to the dispatcher; the dispatcher converts `nextpc` to `tx-nextpc` and jumps to it.

With our call/ret optimization, this translation of indirect calls becomes more difficult. First, `nextpc` is obtained at runtime from the operands of the indirect call instruction, and the jumptable is indexed to try and obtain `tx-nextpc`. If the jumptable hits, the address of the *code cache address* corresponding to the *native return address* (let's call this `tx-retaddr`) needs to be pushed to stack. The code at the native return address may not have been translated yet, and so `tx-retaddr` may not even be known at this point. To handle this, on the jumptable hit path, we emit a "`call tx-nextpc`" instruction immediately followed by an extra unconditional direct branch to `tx-retaddr` (see Figure 3.11). This extra unconditional direct branch to `tx-retaddr` is supplemented by code to branch to the dispatcher if `tx-retaddr` is not known (similar to how it is done for any other direct branch through "direct branch chaining"). A "`«jmp tx-retaddr»`" line in Figure 3.11 represents the full direct branch chaining code (as shown in Figure 3.2) for branching to `tx-retaddr`.

If the jumptable misses for `nextpc`, the dispatcher is burdened with having to push `tx-retaddr` to stack before branching to `tx-nextpc`. We handle this case by using a `call` instruction to exit to the dispatcher (instead of using the `jmp` instruction), thus pushing the address of the code cache instruction following the `call` instruction to stack. The dispatcher proceeds as before, converting `nextpc` to `tx-nextpc` and then jumping to it. A future execution of the `ret` instruction will return control to the instruction following the call-into-dispatcher instruction. At this location, we emit a direct unconditional branch to `tx-retaddr` using the same direct branch chaining paradigm, as used for a jumptable hit (see Figure 3.11).

Note that this call/ret optimization allows code cache addresses to live on globally visible kernel stacks. This global visibility of code cache addresses is acceptable in our design, but will fail if used with previous designs which allow code cache replacement.

```
       set nextpc
       obtain tx-nextpc from jumptable
       if not found, jump to .miss
       call tx-nextpc
       <<jmp tx-retaddr>>
.miss: call dispatcher-entry
       <<jmp tx-retaddr>>
```

Fig. 3.11 The translation code for an indirect call instruction of the type "`call *MEM`" or "`call *REG`", with call/ret optimization. The "`«jmp tx-retaddr»`" line represents the full direct branch chaining code (as shown in Figure 3.2), replacing `tx-retaddr` for `tx-nextpc` (and `retaddr` for `nextpc`).

3.4.4 Translator Switchoff and Cache Replacement

Our design creates more complications at switchoff. Because we store code cache addresses in kernel stacks, we must wait for all such addresses to be removed before overwriting the code cache. To ensure this, we iterate over the kernel's list of threads, replacing PC values on each thread's stack to their translated/native values at switchon/switchoff respectively. At switchon, if the translated value of a PC does not already exist, the translation is generated before replacing the value. The PC values are identified by following the stack's frame pointers.

Finally, we discuss code cache replacement. As discussed previously, we do not allow code cache blocks to get replaced in normal operation. It is possible to hit the code cache space limit if translation blocks are frequently created and later invalidated (e.g., due to module loading and unloading). If we hit the code cache space limit, we switchoff the translator and switch it back on to wipeout the code cache to create fresh space. We only need to ensure that no kernel code is executed between the switchoff and switchon; this is done by pausing all CPUs at a kernel entry (except the CPU on which the switchoff/switchon routine is running) till the new cache is operational. We expect such translator reboots to be rare in practice.

3.5 Implementation and Results

For evaluation, we discuss our implementation, experimental setup, single-core performance, scalability with number of cores, and DBT applications. We finish with a design discussion.

3.5.1 Implementation

Our translator is implemented as a loadable kernel module in Linux. The module exports DBT functionality by exposing `switchon()` and `switchoff()` `ioctl` calls to the user. A `switchon()` call on a CPU replaces the current interrupt descriptor table (IDT) with its translated counterpart. Similarly, the `switchoff()` call reverts to the original IDT. We also provide `init()` and `finalize()` calls. The `init()` call preallocates code cache memory and initializes the translator's data structures, and the `finalize()` call deallocates memory after ensuring that there are no code cache addresses in kernel data structures.

A user level program is used to start and stop the translator on all CPUs. To start, the program calls `init()` in the beginning. To stop, the program calls `finalize()` at the end. In both cases, the program spawns *n* threads (where *n* is the number of CPUs on the system), pins each thread to its respective CPU (using `setaffinity()` calls), and finally each thread executes `switchon()/switchoff()` (for start/stop respectively).

For efficiency, we use a two-level jumptable. Lookup to the first level jumptable does not involve hash collision handling and is thus faster. The second level jumptable is indexed only if the first level jumptable misses. The second level uses linear probing for collision handling and allows up to 4 collisions for a hash location. The most recent access at a location is moved to the front of the collision chain for faster future accesses.

Our code generator is efficient and configurable. It takes as input a set of translation rules. The translation rules are pattern matching rules; patterns can involve multiple native instructions. Our code generator allows codification of all well-known instrumentation applications. Our implementation is stable and we have used it to translate a Linux machine over several weeks without error. Our implementation is freely available for download as a tool called BTKERNEL [1].

3.5.2 Experimental Setup and Benchmarks

We ran our experiments on a server with 2x6 Intel Xeon X5650 2.67 GHz SMP processor cores, 4GB memory, and 300GB 15K RPM disk. For experiments involving network activity, our client ran on a machine with identical configuration connected through 10Gbps ethernet. We compare DBT slowdowns of our implementation with the slowdowns reported in DRK and VMware's VMM. We could not make direct comparisons as we did not have access to DRK; and VMware's VMM uses more virtualization mechanisms like shadow page tables, which make direct comparisons impossible. Hence, to compare, we use the same workloads as used in the DRK paper [29] (with identical configurations).

All our benchmarks are kernel-intensive; the performance overhead of our system on user-level compute-intensive benchmarks is negligible, as we only interpose on kernel-level execution. We evaluate on both compute-intensive and I/O-intensive applications. I/O-intensive applications result in a large number of interrupts, and are thus expected to expose the gap between our design and previous approaches. Some of our workloads also involve a large

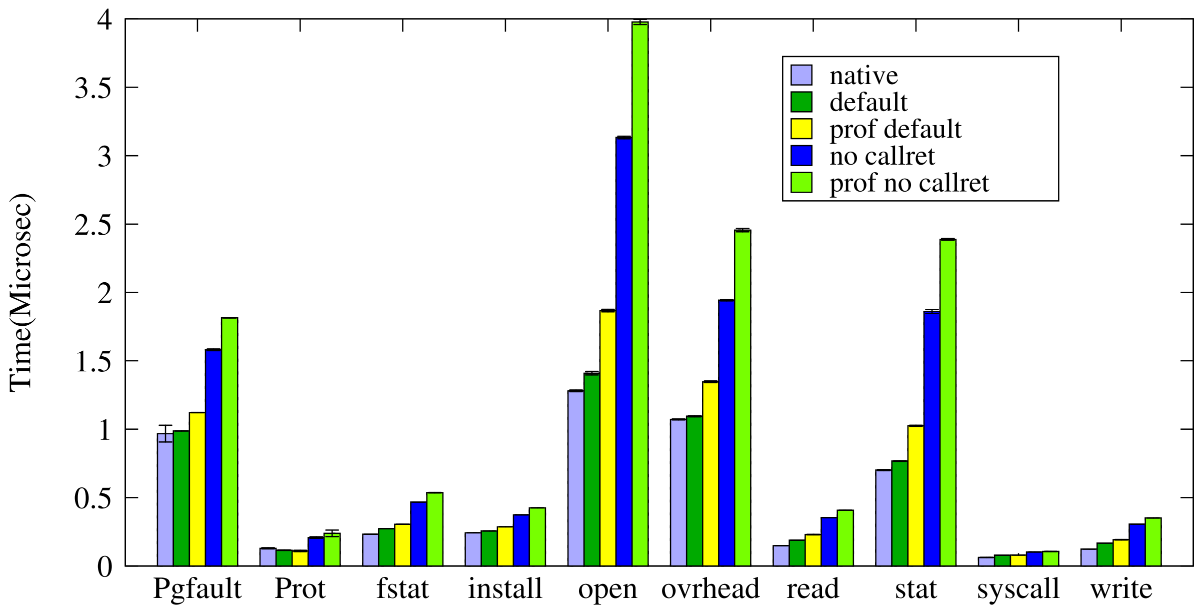

Fig. 3.12 `lmbench` fast operations

number of exceptions/page faults.

We use programs in `lmbench-3.0` and `filebench-1.4.9` benchmark suites as workloads. We also measure performance for `apache-2.2.17` web server with `apachebench-2.3` client, using 500K requests and a concurrency level of 200. We also compare performance overheads during the compilation of a Linux kernel source tree; an example of a desktop-like application with both compute and I/O activity.

We plot performance for two variants of our translator: `default` (all optimizations are enabled), `no-callret` (all except call/ret optimization are enabled). We also implement a profiling client (`prof`) to count the number of instructions executed, the number of indirect branches, the number of hits to the jumptables (first and second level), and the number of dispatcher entries. The corresponding results are labeled `prof-default` (all optimizations enabled) and `prof-no-callret` (all except call/ret optimization enabled) in our figures. Table 3.3 lists the profiling statistics obtained using the `prof` client.

3.5.3 Performance

We first discuss the performance overhead on a single core. Figures 3.12, 3.13, and 3.16 plot our performance results. All these workloads intensely exercise the interrupt and exception subsystem of the kernel. The "fast" kernel operations in Figure 3.12 exhibit less than 20% overhead, except `write` (35% overhead) and `read` (25% overhead). We find 11% improvement in `Protection(Prot)`. Figure 3.13 plots the performance of fork operations in

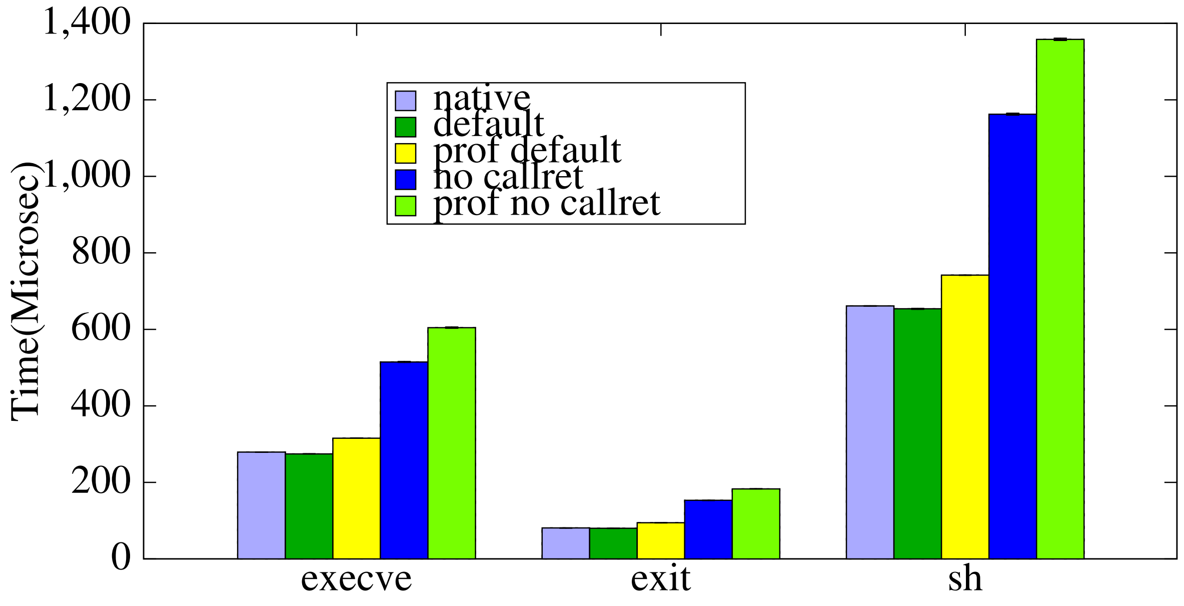

Fig. 3.13 `lmbench` fork operations

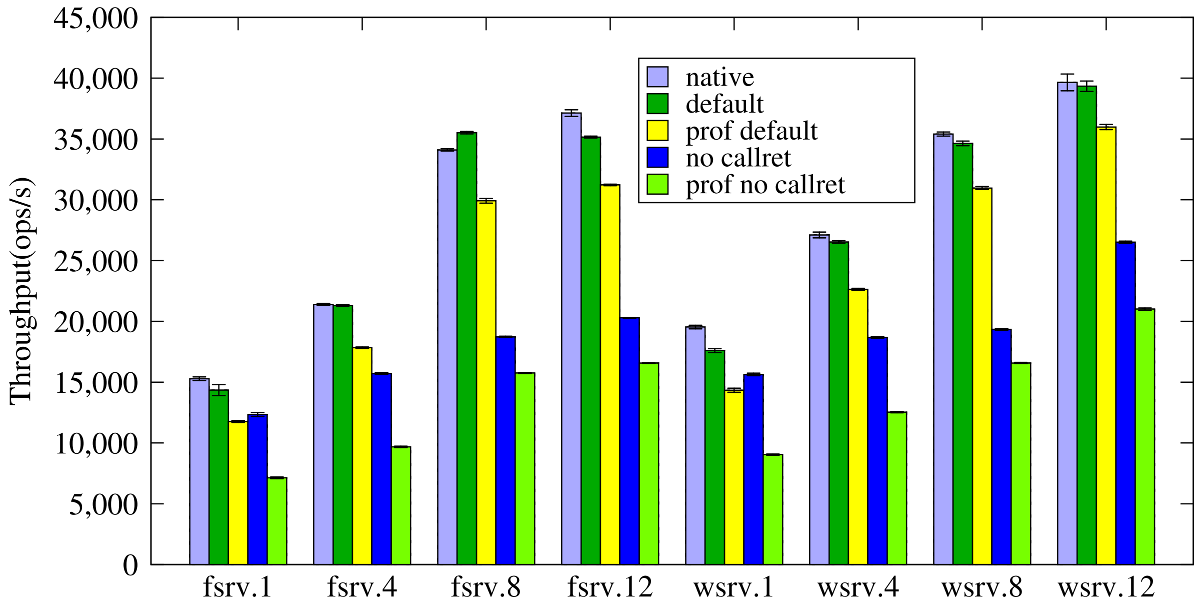

Fig. 3.14 `filebench` on 1, 4, 8, and 12 processors: `fileserver(fsrv)` and `webserver(wsrv)`

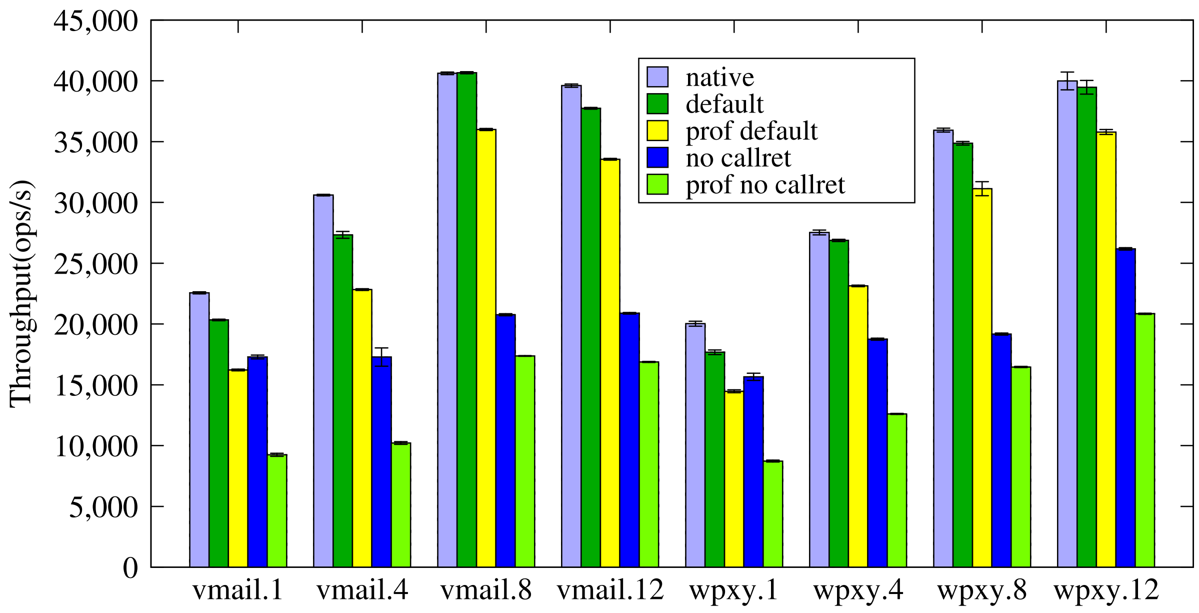

Fig. 3.15 `filebench` on 1, 4, 8, and 12 processors: `varmail(vmail)` and `webproxy(wpxy)`

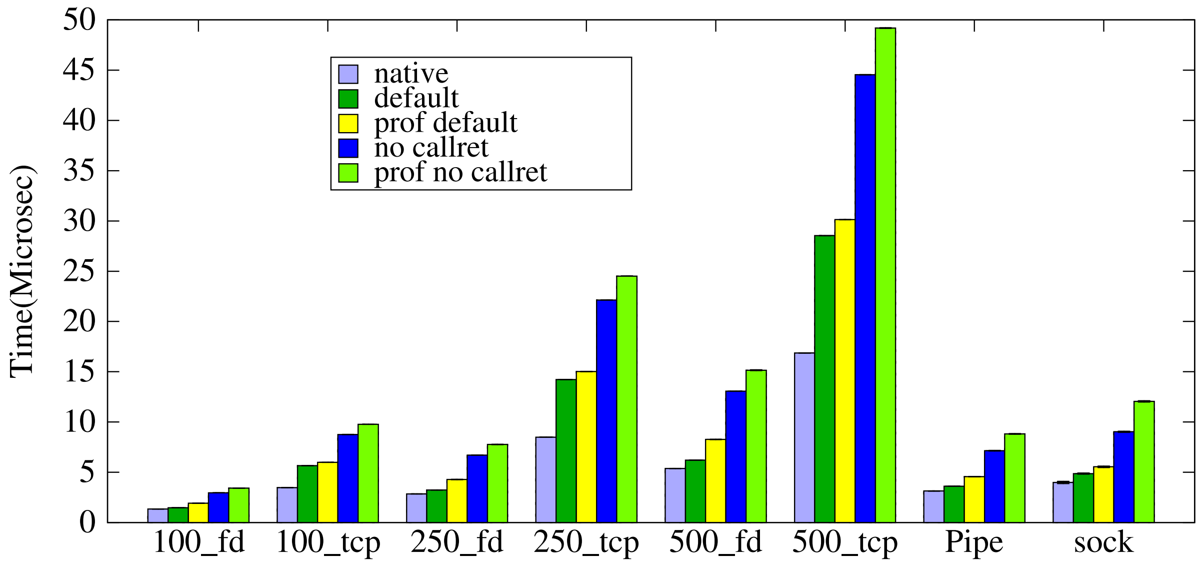

Fig. 3.16 `lmbench` communication related operations

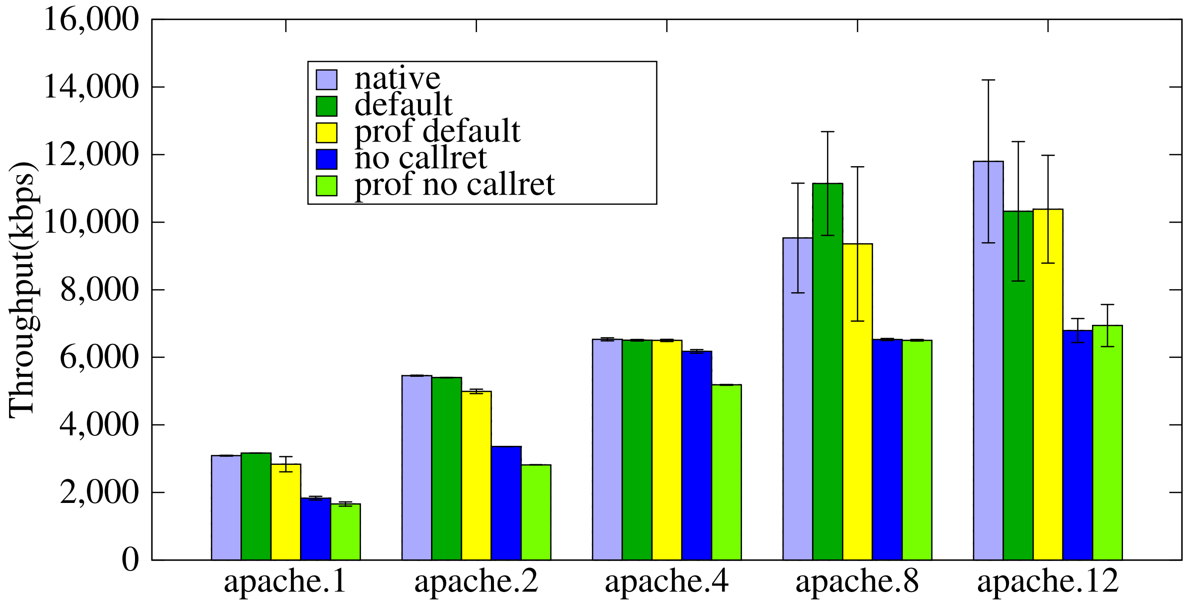

Fig. 3.17 Apache on 1, 2, 4, 8, and 12 processors

	System(s)	User(s)	Wall(s)
native.1	249	3633	4280
default.1	235	3625	4257
prof-default.1	263	3631	4295
no-callret.1	417	3647	4565
prof-no-callret.1	504	3670	4666
native.12	275	3704	573
default.12	273	3702	555
prof-default.12	304	3698	560
no-callret.12	491	3726	590
prof-no-callret.12	573	3740	594

Table 3.2 Linux build time for 1 and 12 CPUs

`lmbench`. Here, we observe 1-1.5% performance improvement with DBT. Similarly, Figure 3.16 plots the performance on communication-related microbenchmarks. DBT overhead is higher for `tcp` (69%) and `sock` (22%); for others, overhead is less than 15%. DRK exhibited 2-3x slowdowns on all these programs. These experiments confirm the high performance of our design on workloads with high interrupt and exception rates.

3.5.4 Scalability

To further study the scalability and performance of our translator, we plot performance of different programs with increasing number of processors. Figures 3.14 and 3.15 plot the

	Without Call Optimization					With Call Optimization				
	Total (x1B)	Indirect (x1M)	Jtable1 (x10K)	Jtable2 (x1K)	Dispatcher Entries	Total (x1B)	Indirect (x1M)	Jtable1 (x10K)	Jtable2 (x1K)	Dispatcher Entries
fileserver	56.54	1285.55	1234.1	51205	238907	94.51	337.49	330.9	6562	17
webserver	62.25	1335.91	1289.1	46674	94351	98.50	401.15	393.9	7179	12
webproxy	62.19	1337.71	1287.1	50389	169203	100.1	406.47	398.9	7485	4
varmail	65.07	1395.25	1337.5	57503	224263	109.7	448.73	439.5	9170	8
linux build	569.1	16038.0	15622.9	342962	72153153	589.9	626.30	613.9	12302	33059
apache	55.65	1650.14	1469.9	173057	7158220	59.10	202.18	171.7	30445	125
tcp500	0.142	3.316	3.3	4	1344	0.268	1.934	1.9	1	0
pgfault	5.294	158.631	158.6	12	2835	5.836	6.915	6.9	1	2

Table 3.3 Statistics on the total number of instructions executed, number of indirect instructions executed, number of first-level and second-level jumptable hits, and the number of dispatcher entries with and without call/ret optimization (obtained by `prof` client). Values in columns labeled `(x1B)` are to be multiplied by one billion, labeled `(x1M)` are to be multiplied by one million, labeled `(x10K)` are to be multiplied by ten thousand and labeled `(x1K)` are to be multiplied by one thousand.

throughput of `filebench` programs with increasing number of cores. To eliminate disk bottlenecks, we used RAMdisk for these experiments. As expected, the throughput increases with more cores, but our translation overheads remain constant. This confirms the scalability of our design (CPU-private structures, minimal synchronization). Interestingly, our translator results in performance improvements of up to 5% for `fileserver` on 8 processors. For other `filebench` workloads, DBT overhead is between 0-10%.

Figure 3.17 shows the throughput of `apache` webserver, when used with `apachebench` client over network. DBT overheads are always less than 12%. We observe performance improvement of 17% (for 8 processors) and 2.5% (for 1 processor) on `apache`. DRK reported 3x overhead for this workload. Table 3.2 shows the time taken to build the Linux source tree using "`make -j`" with and without translation. The time spent in kernel while building Linux improves by 5.6% on one processor, and exhibits near-zero overhead on 12 processors.

Fair comparisons with VMware's VMM are harder, because VMware's VMM also implements many other virtualization mechanisms, namely shadow page tables, device virtualization, etc. However, we qualitatively compare our results with those presented in the VMware paper [2]. The VMware paper reported roughly 36% overheads for Linux build (compared with -5.6% using our tool) and 58% overhead for `apache` (compared with 12% using our tool).

All our performance results confirm that call/ret optimizations result in significant runtime improvements. Table 3.3 reports statistics on the number of indirect branches (that needed jumptable lookups) with/without the call/ret optimization on a single core. Clearly, the majority of indirect branches are function returns. We also present jumptable hit rates (for both levels) and the number of dispatcher entries for different benchmarks in the table. These statistics were generated in steady state configuration, when the code cache has already warmed up. Without call/ret optimization, the jumptable hit rates for `apache` were 99.56% (89.07% first level, 10.48% second level). With call/ret optimization, the jumptable hit rates were always above 99.99% (84.94% first level, 15.05% second level). In all our experiments, the number of dispatcher entries was roughly equal to the number of jumptable misses.

3.6 Discussion

In summary, our fast DBT design has the following salient features:

- We avoid back-and-forth translation of interrupted/excepting PCs between native and translated values, on interrupt entry and return.
- We assume a large enough code cache, so it can fit all kernel code and does not need cache replacement during normal operation.
- We relax precision requirements on exceptions and interrupts.
- We maintain temporary DBT state on kernel thread stacks and use a reentrant dispatcher.
- We use a cache aware layout for the code cache.
- We use identity translations for function call and return instructions.

Evidently, our DBT design requires knowledge about guest OS internals, to handle special cases appropriately. We also require the guest to obey certain invariants:

- The guest should read the interrupted/excepting PC value (pushed on stack by hardware) mostly through the return-from-interrupt instruction and should be otherwise indifferent to it, except special cases that can be handled specially.
- The guest should not depend on precise exceptions and interrupts.
- The guest should allow a module to access the kernel's list of threads and their call stacks, to allow translation of return address PCs to translated and native values at switchon and switchoff times respectively.
- The guest must obey the stack discipline.
- After it has booted, the guest must use function return addresses only through bracketed call/return instructions, to allow call/ret optimization.

For these reasons, our design is inappropriate for use in VMMs expected to run *any* guest OS. Our scheme can be used however to improve performance for *specific* guest operating systems, using a custom guest-side kernel module in VMMs.

As an alternative though, it is possible to write static or dynamic analysis tools to determine if an optimization is legal. Verification for the call/return optimization would involve checks that all code uses the return address only through bracketed call/return instructions. Notice that any dynamic analysis can be implemented by using or DBT implementation (without optimizations).

Our scheme improves performance for several other DBT applications like instrumentation, testing, architecture compatibility, profiling, sandboxing, and dynamic code optimization. On the other hand, some applications which anticipate unconventional guest OS behaviour may not work with our DBT design. For example, it may not be desirable to use our framework for certain security-related applications (e.g., rootkit analysis); such applications may require full transparency to hide from a malicious program, and our framework may violate this requirement.

3.7 Related Work

User-level DBT frameworks are well understood, with many different systems built on similar techniques: DynamoRio [16], Pin [44], Valgrind [49], vx32 [32], etc. User-level DBT requires stricter transparency requirements, as few assumptions can be made on user program behaviour. In contrast, we show that it is possible to rely on typical kernel behaviour to provide design simplicity and performance for kernel-level DBT.

JIFL [53] is a kernel-level DBT framework that provides an API to instrument system calls. JIFL does not instrument interrupt handlers and kernel threads, making it less comprehensive than our work. Similarly, PinOS is a whole-system instrumentation framework to instrument a guest running paravirtualized in a Xen hypervisor [19], based on the Pin [44] instrumentation framework. Firstly, running in a virtual machine limits execution coverage, as only device drivers for virtual devices get executed. An instrumentation framework for a bare-metal OS (such as ours) can execute drivers for any device, provided the appropriate hardware is available. Secondly, Pin uses a call-based model of instrumentation and so is much slower. PinOS uses similar mechanisms as DRK and VMware to ensure precise exceptions and interrupts. With already high DBT overheads (of Pin), the small overhead of extra mechanisms at interrupts and exceptions (of PinOS) is relatively insignificant.

We compare the differences and similarities between our work and VMware's DBT-based VMM [2, 3] throughout this chapter. Unlike VMware, our approach can instrument all device drivers (and not just drivers that get exercised in VM environments), and provides better interrupt/exception performance. However, our design requires guest-specific knowledge. We believe (though do not show) that our techniques can be used in VMMs to improve performance for specific guests, through custom guest kernel modules ("guest tools"). Device passthrough configurations on VMware's DBT-based VMMs can also benefit from our techniques to efficiently interpose on device interrupts.

Chapter 4

Deterministic Replay

The goal of deterministic replay is to recreate the execution using a trace of non-deterministic events. Deterministic replay has applications in debugging [4, 41, 48, 54, 55, 61], fault tolerance [8, 13–15], intrusion detection [26], remote attestation [37], computer forensics [26], dynamic analysis [21, 51, 56], profiling [5], testing and verification [48, 56], trace generation [12, 56, 66], and more. We study the problem of efficient whole system multiprocessor record/replay. The biggest challenge in deterministic replay is to record the order of shared memory reads and writes. In this chapter we discuss our methods to efficiently record shared memory non-determinism using Dynamic Binary Translation.

4.1 Some Illustrative Examples

Consider the following example of a function `foo()` to contrast the different approaches to deterministic replay, and to motivate our approach.

```
void foo(int *ptr) {
  (*ptr)++;
}
```

`foo()` dereferences the `ptr` argument passed to it as a function argument. If multiple threads can call `foo()` on the same `ptr`, the order of accesses by these threads to this shared variable needs to be recorded for deterministic replay.

To deterministically replay this program, the SMP-Revirt approach [27] incurs expensive, and sometimes redundant page faults on accesses to this location by multiple threads. RecPlay [57] fails if accesses to the shared `ptr` are not protected by an explicit lock. PRES [55] only records an “execution sketch” (e.g., the order of explicit synchronization operations) which represents only a subset of all the execution non-determinism, and during replay, searches for an execution schedule that reproduces the recorded sketch. In this example, assuming that the access to `ptr` was not protected by explicit synchronization, replay may have to execute a few different schedules before being able to reproduce the recorded sketch. This search space during replay increases exponentially with the number of data races in the program. Similarly, ODR [4] records the "observable" behaviour of the program, and uses it to guide the search for a matching data-race schedule during replay. Some examples of observable behaviour that can be recorded are order of synchronization operations, order of memory reads, sequence of taken branches, etc. The choice of the recorded behaviour dictates the tradeoff between high recording overheads and large search spaces during replay. If the recorded information is too sparse (e.g., synchronization order), replay involves searching through an exponential space of all possible data-race schedules. On the other hand, if the recorded information is too dense (e.g., sequence of branches), it results in high recording overheads (e.g., 4.5x overheads reported in [4]).

Respec [42] and DoublePlay [62] execute the replayed run in an online fashion for the execution time slice containing the access to `ptr`; if the replayed run happens to have the same access interleaving as the recorded run, the replay is successful, else the replay is retried through rollbacks. Again, the expected number of rollbacks increases swiftly with the number of unsynchronized (or implicitly synchronized) memory races executed in one time-slice of the program. A rollback is very expensive, and increases recording overhead significantly.

We avoid the limitations of previous work by inferring the sharing behaviour of the program instructions, through its execution over several test inputs in a *training phase*. We use the inferred information to appropriately instrument our target program. In this example, executions of the function `foo()` in a training phase infers that the instruction "`(*ptr)++;`" can potentially access data that is shared across multiple threads. We call such instructions that are known to potentially access thread-shared data, *sharing instructions*. During record, our tool inserts calls to `acquire(lock)` and `release(lock)` routines before and after all sharing instructions respectively, through DBT. These `acquire(lock)` and `release(lock)` routines simulate the acquisition and release of an imaginary `lock`, created by us. Through these routines, we serialize the execution of all potentially sharing instructions with respect to each other. Further, the `acquire(lock)` routine records the acquisition order of our imaginary `lock`, and this recorded order captures the non-determinism due to concurrent access to shared memory by multiple threads. The instrumented program looks like the following:

```
void foo(int *ptr) {
  acquire(lock);
  (*ptr)++;
  release(lock);
}
```

There are four advantages of instrumenting dynamically-inferred sharing instructions like this, over previous work:

First, our instrumentation overhead is significantly lower than the page-fault based mechanism of SMP-Revirt.

Second, we do not rely on the presence/absence/infrequency of data races. If data sharing exists in the program and if it was noticed during the training phase, it will get appropriately instrumented during the recording phase. Conversely, it is possible for a data-sharing instruction to remain unidentified during the training phase, but result in unrecorded non-determinism during the recording phase. In this case, we use a similar search procedure as PRES and ODR during replay, to find a schedule that matches the trace of our recorded run. Because such unidentified data sharing instructions are expected to be very rare (assuming the training phase runs the program several times over several inputs), our replay-time search space is likely to be much smaller than PRES and ODR (which relied on properties like rarity of data-races). In our experiments on the Linux kernel, we ran the training phase by running many applications over 30-40 minutes; with this modest amount of training, we were able to replay all our recorded runs in the first trial — this indicates that no non-determinism due to unidentified sharing instructions was present in our several recorded runs, perhaps because all sharing instructions were correctly identified during our training phase.

Finally, we do not rely on being able to identify and interpose on explicit synchronization operations. We treat synchronization instructions like all other instructions accessing program data. In our method, these synchronization instructions are just highly likely to be identified as sharing instructions in the training phase, given that they are likely to access thread-shared synchronization objects (e.g., lock variables). Thus, we avoid the very complex and hard task of identifying, understanding, and manually instrumenting explicit synchronization operations, which was necessary in many previous works [4, 42, 55, 57, 62].

4.2 Data-level and Code-level Mutual-Exclusion

The act of recording the non-determinism involves synchronizing accesses to shared state. There are two broad approaches to doing this, which we call *data-level mutual-exclusion* and *code-level mutual-exclusion*.

Data-level mutual-exclusion models all code and data as belonging to one shared address space, and synchronizes each memory access by each CPU. In other words, a CPU is modeled as a thread executing in a shared-memory cache-coherent address space. This is identical to the underlying hardware model on shared-memory machines. Synchronization involves CREW-like ownership tracking of memory locations, which involves associating metadata with each memory location to store its ownership status. Just like SMP-Revirt, a CPU must ensure that it holds the required privileges before accessing a memory location, and all ownership transfers must be logged to enable offline replay. All previous approaches to deterministic replay, also use this model. When compared with previous work, our DBT-based approach allows us to implement data-level mutual-exclusion at byte-granularity with lower overheads (Section 4.4).

Code-level mutual-exclusion, on the other hand, divides code regions into disjoint sets called *monitors*, with the property that instructions from two different monitors can never access a memory location concurrently. In this model, ensuring that at most one CPU is active inside a monitor at all times, suffices. All monitor-entry events are logged to enable offline replay. If we put all code in a single monitor, the system degenerates to a single-threaded system. The more monitors we identify, the better concurrency we enable. Precisely identifying monitors for black-box binary code is undecidable in general. We use an execution-driven approach through an offline training phase to *imprecisely* identify the monitors. This imprecision can both under-approximate and over-approximate the monitors. In practice, the approach works well however. We discuss our algorithm to identify the monitors, and the resulting imprecision and its implications in Section 4.2.1.

Data-level mutual-exclusion works especially well if sharing is sparse and data-dependent. On the other hand, code-level mutual-exclusion works better if there is a large amount of sharing that happens in relatively small critical sections. In this case, it is better to synchronize the critical sections themselves, instead of synchronizing on the data that they access.

This tradeoff between data-level and code-level mutual-exclusion is somewhat similar to the classic tradeoff between shared-memory and message-passing architectures. Code-level mutual-exclusion divides code regions into monitors. Using the duality argument [40], these monitors can be mapped to processes in a message-passing architecture, and CPU threads can be mapped to messages being exchanged between the different monitors. In the dual setting, deterministic replay then involves recording the order of message arrival at the process ports (or monitor entry points). Similarly, the data-level mutual-exclusion model is identical to the “procedure-oriented system" in [40].

As pointed out in the duality paper [40], the choice of model (code-level or data-level) then depends only on the nature of the machine architecture. In our case, the underlying hardware model is cache-coherent shared-memory, arguing for the procedure-oriented approach (i.e., data-level mutual-exclusion). However the recording subsystem alters the effective cache hit and miss costs of the underlying hardware. With recording enabled, a memory access is a cache hit if the location was owned (in read-shared or write-exclusive mode, depending on the nature of access) by the accessing CPU at the time of access; otherwise, it is a cache miss. Thus, the effective cost of a cache hit increases slightly (over that of the underlying hardware), as it involves an extra ownership check. The effective cache miss-cost however increases significantly as it now involves transferring ownership from the other CPU and also logging this event. This makes the effective miss cost significantly higher than the effective hit-cost, with the miss-cost:hit-cost ratio being significantly higher than that of the underlying architecture. Thus, message-oriented synchronization approach (i.e., code-level mutual-exclusion) often performs better for recording many workloads, even if the original software program and the underlying hardware were written and optimized for the procedure-oriented (shared-memory) style.

4.2.1 Monitor Identification

Identifying monitors involves identifying instruction pairs that can potentially be involved in a data race, so that they can be added to the same monitor. Several previous work on static and dynamic data-race identification [28, 31, 39, 46, 59, 67] can be leveraged here. For this work, we implemented a dynamic and relatively imprecise data-race detector.

For monitor identification, we run the program in a separate offline training phase with several test inputs. During the training phase, the program is instrumented (using our DBT driver) to maintain shadow metadata state with every memory byte. In particular, we store the IDs of the kernel threads that have accessed that byte so far. If a byte is seen to be accessed by multiple threads, we categorize it as a shared location. We then identify all code instructions that access that byte, and infer that all these instructions conflict with each other, i.e., they can potentially be involved in a data-race. To guard against memory reuse, we instrument the memory-management routines (malloc and free) to reinitialize the metadata state associated with the bytes being allocated. We identify these routines by searching for common memory-management routine names in the target program's symbol table.

Our monitor-identification algorithm is imprecise on both sides, i.e., it is possible that two instructions that can never access a memory location concurrently, are still identified as conflicting, and put in the same monitor (over-approximation); conversely, it is also possible that two instructions that potentially conflict are put in separate monitors (under-approximation).

Two common reasons for over-approximation are: (1) our algorithm ignores any happens-before relationships (due to synchronization operations, for example) between two memory accesses; (2) a memory management routine may remain unidentified, resulting in false conflicts due to memory reuse. Over-approximation of monitors, causes unnecessary serialization, but does not affect the correctness of our algorithm.

The most common reason for under-approximation is incomplete test coverage during training phase, i.e., a memory-sharing pattern may remain unexercised during the training phase, but may occur during the recording phase. In this case, it is possible that the non-deterministic order of shared-memory accesses by the two instructions remains unrecorded, as they belong to different monitors. This can cause a divergence during replay, and hence is a correctness concern.

We handle errors due to under-approximation by modeling replay as a "guided-search" procedure, exactly as done in previous work on output-deterministic replay and probabilistic replay using execution sketching [4, 55]. However, the probability of these errors in our work, is significantly lower than that of previous work, as we do not rely on the correct identification of all data-synchronization operations. In our experiments on recording and replaying the Linux kernel, we identified 1774 monitors, at instruction granularity, in the Linux kernel code in the training phase. These exclude instructions that were never seen to be involved in any sharing.

For us, monitors are equivalence classes. i.e., we consider the conflict-relation between instructions to be reflexive, symmetric, and transitive. Our training-phase algorithm represents monitors as sets of instructions. Initially, each instruction is in a separate singleton monitor. On noticing a conflict between two instructions, we take a union of the two corresponding monitors, to yield one unified monitor. The set of monitors at the end of the training phase, is the final set of monitors to be used for record/replay.

We implement monitors at function granularity[1]. i.e., monitors are represented as sets of functions (and not as sets of instructions). In theory, this can lead to over-approximation, i.e., two instructions belonging to the same function, which should ideally belong to different monitors, could unnecessarily get merged into the same monitor. In practice though, the performance effect of this over-approximation is insignificant. On the other hand, using function granularity greatly simplifies our instrumentation logic and data structures. At function granularity, the number of monitors identified in the training phase reduce to 168.

Finally, we also mark instructions that are never seen to access shared data during the training phase. These instructions do not need to be instrumented for record — neither for code-level mutual-exclusion, nor for data-level mutual-exclusion. Once again, imprecision can result due to incomplete test coverage of the training phase. In practice, such imprecision is small and is effectively handled by modeling replay as a guided-search. We call all instructions that are seen to access shared data even once during the training phase, *shared memory access instructions*. Only these instructions are instrumented during record and replay.

[1]Our Linux-based DBT driver identifies functions by looking at targets of the function call instructions, and also by consulting the Linux kernel's symbol table.

4.2.2 Categorizing Code-Regions for Data-Level and Code-Level Mutual Exclusion

We also categorize the code instructions of the target program depending on their sharing behaviour, during the training phase. Ideally, we would want to use data-level mutual exclusion for code that is involved in sparse and data-dependent sharing; and code-level mutual exclusion for code that is involved in dense sharing. Again, we perform this categorization at function granularity. i.e., a function is either marked a *data-level mutual-exclusion (DME) function*, or a *code-level mutual-exclusion (CME) function*. This categorization depends on the sharing behaviour seen during the training phase: a function that is seen to be involved in relatively sparse sharing (or no sharing) is marked DME, while a function that is seen to be involved in dense sharing is marked CME.

In our implementation, we perform this CME/DME categorization initially (before the record phase); however, it may be best to adapt this categorization dynamically during the record phase depending on the characteristics of the current workload. The statistics collection overhead for this categorization during record is negligible, as ownership-transfers due to sharing are getting logged anyways. While we have not implemented dynamic categorization during record, we show results with different initially-chosen workload-specific categorizations in Section 4.4, to simulate a similar effect.

Thus, through our analysis, each function is associated with a monitor, and is categorized as either CME/DME. A DME function is instrumented with ownership-tracking instructions before each shared memory access instruction. A CME function uses identity translation. i.e., no instrumentation code/overhead. We ensure that a CME function cannot execute concurrently with another function (CME or DME) belonging to the same monitor. On the other hand, DME functions can execute concurrently with other DME functions, even if they belong to the same monitor.

4.3 Implementation

We implement mutual-exclusion using *instrumented read/write locks*. For data-level mutual-exclusion, we associate a lock with each memory byte. At each shared memory access instruction, we prepend and append code to acquire and release the associated lock respectively. For instructions performing read operations, we prepend with the `read_acquire()` routine and for write instructions, we prepend with the `write_acquire()` routine. The lock state also maintains the IDs of the CPU threads that last held the lock (multiple threads could hold the lock in read-mode simultaneously). If the current CPU thread's ID does not appear in the list of thread-IDs that last held this lock, a log entry is generated to record the ownership-update event.

For code-level mutual-exclusion, we associate a read/write lock with each monitor, and add instructions implementing monitor-lock acquire and release at function entries and exits respectively. Entries to DME functions acquire the monitor-lock in read mode, to allow multiple DME functions to execute concurrently within the monitor. Entries to CME functions acquire the monitor-lock in write mode, to prevent any other function (CME/DME) to execute concurrently. All function exits (through branches or function calls) are instrumented with the monitor-lock release instructions. As with data-level locks, monitor-locks maintain the thread ownership information, and log ownership transfer events for deterministic replay. We also call locks implementing data-level mutual exclusion, *data-locks*; and locks implementing code-level mutual exclusion, *code-locks*.

This scheme implements mutual exclusion at the desired granularity, and together with logging, ensures deterministic replay. If done simply, the instrumentation overhead of this scheme is very high, and the following optimizations were necessary for good performance:

1. Identifying shared-memory access instructions during training phase, and only instrumenting these instructions (as already discussed).

2. Tuning the lock implementation, for the common case of a lock being repeatedly acquired and released by the same thread. Our current instrumentation overhead is two instructions (from ten originally) per shared-memory access instruction.

3. Rate-limiting ownership transfers.

4. Optimizing control flow transfers between CME/DME functions, to minimize instrumentation overhead.

5. Unifying monitors to reduce instrumentation overhead.

We next discuss the implementation of our instrumented locks, and the optimizations involved.

4.3.1 Lock Implementation

Figure 4.1 presents the pseudo-code for our lock acquire and release routines. These routines are optimized for our common case where a lock is acquired several times repeatedly by the same thread, and the critical sections are small.

```
struct rwlock {
  /* bitmap indicating thread-ids */
  word_t owners;
};

struct thread_t {
  int id;  /* thread id */
  bool inside_critical_section;
      /* whether this thread is currently
         within a critical section */
  long sequence_number;
      /* stored with the record log entries */
  unsigned char read_mask;  /* (1 << id) */
  unsigned char write_mask; /* ~(1 << id) */
};

read_acquire(struct rwlock l):
  cur_thread.inside_critical_section = true;
  if ((l.owners & l.read_mask) == 0) {
    /* slow path : update ownership
       and log event; unmark critical section
       and mark at the end */
    read_acquire_slowpath(l, cur_thread);
  }
  cur_thread.sequence_number++;

write_acquire(struct rwlock l):
  cur_thread.inside_critical_section = true;
  if ((l.owners & l.write_mask) != 0) {
    /* slow path : update ownership
       and log event; unmark critical section
       and mark at the end */
    write_acquire_slowpath(l, cur_thread);
  }
  cur_thread.sequence_number++;

release(struct lock l):
  cur_thread.inside_critical_section = false;
```

Fig. 4.1 Pseudo-code of our instrumented reader-writer lock routines. The `thread_t` structure stores per-CPU state. The `owners` field stores a bitmap of the CPUs that currently own this location. At any time, a lock must have at-least one owner. If it has multiple owners, it must be in a read-shared state. The `read_acquire()` function checks if the current thread is one of the owners. The `write_acquire()` function checks if the current thread is the only owner (by checking that no other thread owns this lock). The slowpath functions are called if these checks fail, and they update the ownership information. The `write_acquire_slowpath()` function must wait for the current owner CPUs to leave the critical sections (i.e., wait for their `inside_critical_section` flags to become false) before updating ownership. This implementation of reader-writer locks is tuned for very-small critical sections and frequent acquisition of a lock by the same thread repeatedly. The sequence number is logged with every ownership update event, to record the order of events.

Because our instrumentation code causes execution of extra instructions (especially branches), it interferes with the branch-counting logic used for uniprocessor record/replay[2]. To avoid this, we implemented the following logic in the hypervisor:

1. We enable branch-counting only for user-mode execution in the guest CPU. Kernel-mode execution does not modify the hardware performance counters.

2. Before delivering an interrupt to the guest, the hypervisor checks if the guest CPU is executing in user-mode or kernel-mode. If executing in user-mode, it delivers the interrupt as before, and records the branch count. If executing in kernel-mode, the hypervisor delays the delivery of the interrupt till the next VM exit by the guest due to a deterministic reason (such as execution of an I/O instruction). Because interrupts in kernel-mode are now only delivered at deterministic exits from the guest, branch-counting in kernel-mode is not required for deterministic replay; only recording the deterministic exit number suffices. To ensure bounded waiting times, we also inject a guest interrupt at *poll points* (see Section 4.3.2), which are also deterministic, and can be efficiently counted.

Thus, because we disable branch-counting in kernel-mode, our instrumentation code does not interfere with record/replay determinism. If implemented as such, the acquire and release routines require 10 assembly instructions on x86 (ignoring the slowpath). Most of these instructions implement reading the current thread-id from memory (using the per-CPU segment `%fs`) into a temporary register, as extra instructions are required to save and restore the temporary register and flags.

We optimized the instrumentation code by using per-CPU code caches within our DBT driver, and specializing this instrumentation code for each CPU. Thus multiple copies of each code block (one per CPU) exist in our DBT code cache. The instrumentation code in each copy is specialized for that CPU. In particular, instead of reading the current thread ID from memory, we hard-code the number inside our instrumentation code, as an immediate value. This saves us two instructions required to save and restore the temporary register, and one instruction to read the current thread ID from memory. Further, we perform a liveness analysis on the code block being instrumented to avoid having to save and restore the flags, which saves another two instructions, bringing down our instrumentation code tally to 5 instructions (from 10).

[2]Branch-counting is required to ensure deterministic delivery of asynchronous interrupts to the CPU [26].

Interplay with Call/Return optimization

Recall the call/return optimizations in Section 3.4.3 – we allow code cache addresses to be live in the kernel thread stacks. If all the CPUs share the same code-cache, this approach works fine but if the CPUs have different code-caches, it creates a new problem. Consider a scenario when thread A is scheduled on CPU2 that was previously scheduled on CPU1. When the thread A was preempted on CPU1 its stack contained the code cache addresses corresponding to CPU1. When the scheduler picks thread A to run on CPU2, its stack still contains addresses corresponding to CPU1. Since with call/return optimization enabled return instruction directly jumps to the code-cache address corresponding to return address, when the thread A will unwind its stack it will incorrectly jump to the code corresponding to CPU1. This can violate an invariant, which can potentially end up with crashing the recorded execution. To handle it correctly we insert a callback function in the schedule routine which replaces code cache addresses in the stack with the code cache address corresponding to the current CPU. Notice that the overhead of callback routine is negligible if the CPU on which the current task was scheduled previously is the same as the current CPU; it just requires a check on the return address corresponding to the schedule routine. Since we also allow interrupts to push code cache addresses on the stack (Section 3.3), we also insert a callback function before every iret to replace the return addresses as we did in the case of schedule.

To replace the return address on stack, we traverse the function frame pointers (`ebp` on x86) pushed by function headers on the stack. In doing so, we make assumptions about the compiler following certain conventions. In our experiments on Linux (compiled using `gcc`), we did not face any issues in the compiled code. However, we had to manually modify four assembly functions in Linux to ensure that our design and implementation remains correct. We augmented our training phase to identify such functions, and passed this information to our DBT dispatcher, which replaced these functions with our manually-modified implementation. In two of these functions (`call_rwsem_down_read_failed`, `call_rwsem_down_write_failed`), the manual modification only required saving and restoring the frame pointer in the function's header and footer respectively. In the other two (`ptregs_clone`, `PTREGSCALL3`), we had to add one extra location to stack. These last two functions are executed at system-call entries of `clone` and `execve` respectively.

4.3.2 Disabling Lock Preemption

Our next trick to reduce the instrumentation code involves disabling lock preemption. In the scheme outlined so far, we allow one thread to remove the ownership of another thread from a memory location, by allowing it to update the ownership bitmap in the slowpath code (provided that the other thread does not have the `inside_critical_section` flag set). This capability requires us to maintain a per-thread sequence number and also requires us to set and unset the `inside_critical_section` flag on every acquire and release respectively, so that another thread may be allowed to acquire the lock immediately, if it needs it.

Because ownership transfer events are very expensive (they involve generation of a log entry), minimizing ownership transfers is an important optimization. Consequently, it is not as important to allow one thread (say, thread A) to immediately acquire a lock when it needs it, if it means transferring ownership from another thread (say, thread B) — it's quite likely that thread B may again access the same memory location in future, and not transferring the ownership immediately will avoid the overheads of ownership shuttling.

Given this background, we disable lock preemption completely. A thread can only request the lock owner to yield the lock; and each thread must periodically poll for all outstanding lock-yield requests, and satisfy them (by voluntarily yielding the locks held by it). In other words, if thread A requires a lock that is currently held by thread B, it must wait for thread B to voluntarily yield that lock. Thread B will periodically poll for all outstanding lock requests and satisfy A's request. This polling-based implementation further reduces the instrumentation code on the fast path. Firstly, the `inside_critical_section` flag does not need to be set and unset on every acquire and release, which saves another two instructions. Secondly, the sequence number does not need to be updated on every lock acquisition; it now suffices to increment the sequence number only at the periodic poll points. The optimized two-instruction assembly implementation of our `acquire()` and `release()` routines is given in Figure 4.2. Notice that the `release()` function is now empty.

The execution points where the thread polls for outstanding requests should be locations where the overhead of polling is relatively small, i.e., these locations should be executed relatively infrequently. We call these locations, *poll points*. Using poll points, we also rate-limit the ownership transfers, which we have found to be an important optimization to reduce the execution overhead and log-size growth rate. We use the following poll points in our hypervisor-level deterministic replay implementation:

1. VM exits: this includes exceptions, I/O instructions, and other instructions that cause a VM exit.

2. Execution of the `pause` instruction by the CPU: Almost all spin-waiting loops execute the pause instruction, and identifying them as poll points ensures that we do not cause deadlocks by introducing cyclic waiting dependencies[3].

[3]We found some spinloops in the Linux kernel that did not use the `pause` instruction. We instrumented these loops with the `pause` instruction ourselves, through DBT.

```
write_acquire:
 testb $CPU_ID_WRITE_MASK, shadow_offset(MEM)
 je 1f
 call write_acquire_slowpath
1:

read_acquire:
 testb $CPU_ID_READ_MASK, shadow_offset(MEM)
 jne 1f
 call read_acquire_slowpath
1:

release:
  //empty (nothing do be done on release)
```

Fig. 4.2 The optimized assembly implementations of acquire and release routines. `CPU_ID_READ_MASK` and `CPU_ID_WRITE_MASK` are per-CPU mask values used to compare against the shadow byte. `shadow_offset` is the offset at which the lock associated with memory location `MEM` resides. For word/doubleword instructions, we use `testw`/`testl` (in place of `testb`) instructions with the appropriate read/write masks, respectively.

3. Slowpath functions: Before a thread starts waiting for another thread to yield a lock inside a slowpath function, it polls its outstanding lock-yield requests. This is also required to avoid deadlocks resulting from cyclic lock-acquisition dependencies.

4. Other statically-identified code locations, to ensure bounded waiting times, without unduly increasing single-threaded overhead: The DBT driver is used to insert these poll points. For this work, we manually identified 2-3 such locations in the Linux kernel.

Together, these optimizations reduce the average single-threaded instrumentation overhead from 6x to 3x for a sharing-intensive benchmark like `forkwait`, which spawns and waits for several processes created in the Linux kernel (Section 4.4). Notice that we only disable preemption for data-locks; code-locks (for code-level mutual-exclusion) are still preemptive as the instrumentation overhead due to code-locks is relatively small.

4.3.3 Optimizing Control Flow Transfers between Functions

Control flow transfers between two functions, where at-least one function is a CME function, result in execution of the instrumented lock routines for acquiring and/or releasing the monitor lock. In many cases, this overhead can be avoided. For example, if a CME function calls another CME function in the same monitor, it would be wasteful to release the monitor-lock at the caller function's exit, only to re-acquire it back at the callee function's entry.

To facilitate such optimizations, we allow multiple translations of each function to exist simultaneously in our DBT driver's per-CPU code cache. For example, a CME function can exist in two translated implementations: first, where the function entries and exits are instrumented with lock routines; and second, where the function is translated identically (no instrumentation). If the caller is a CME function and belongs to the same monitor as the CME callee, then it transfers control to the second implementation, thus avoiding instrumentation overheads. All other functions transfer control to the first implementation. (We discuss indirect control-flow transfers later in this section).

As another example, if a CME function calls a DME function, it needs to release the monitor-lock before the call, and re-acquire it after the function return. If the callee is a short function, this instrumentation overhead on every call and return is significant. If the two functions belong to the same monitor, it may be more efficient to use a CME implementation of the callee for this call. We allow two translated implementations of a DME function: first, where all its shared-memory access instructions are instrumented with appropriate lock instructions; and second, where the function is translated identically (no instrumentation). If the caller is a CME function and belongs to the same monitor as the DME callee, then it transfers control to the second implementation, avoiding instrumentation overheads. All other functions transfer control to the first implementation. Similarly, DME-to-DME control flow transfers can also avoid `release` and `read-acquire` instrumentation overhead on the monitor-lock.

So far, we have discussed only direct control-flow transfers where the callee is known at translation time, and hence the callee implementation (to branch to) can be chosen at translation time. For indirect control-flow transfers, our DBT driver uses a map from the target program PCs to the code-cache addresses, called *jumptable*, to resolve the destination address at runtime Figure 3.3. The PC address generated at runtime is translated to its corresponding code-cache address using the jumptable, and control is transferred to it. In our case, we use caller-specific jumptables at the call-site. For example, the jumptable at a CME function will contain mappings to the code-cache addresses of uninstrumented implementations of the callee CME functions (belonging to the same monitor); on the other hand, the jumptable at a DME function will contain mappings to the code-cache addresses of instrumented implementations of the callee functions.

4.3.4 Unifying Monitors to Reduce Instrumentation Overhead

Identifying a large number of monitors helps in better overall concurrency. However, crossing monitor boundaries also results in instrumentation overhead, as these crossings cannot benefit from the optimizations discussed in Section 4.3.3. It is often more efficient to unify two monitors into one, if there are a large number of crossings observed between them.

Using this idea, we unified monitors to reduce instrumentation overhead. In fact, for all our experiments on the Linux kernel, we found that using only one monitor for the whole kernel, produced best results. While this was our experience with the Linux kernel, it is quite possible that for another system, it may be better to use more than one monitors. Based on our experience however, we suspect that most systems will yield best results with only a small number of monitors. (Recall that even after merging many monitors into one, the distinction between CME and DME functions, and their associated synchronization provides significant concurrency).

4.4 Experiments

Our multiprocessor VM deterministic replay system is implemented inside KVM/Qemu on an x86/Linux host, and all our experiments record and replay a full Linux guest. To capture uniprocessor non-determinism, we logged all interrupts with their timing, all input I/O (e.g., network packets) and the results of all non-deterministic instructions (e.g., `rdtsc`). We used hardware performance counters to count the number of branches executed in user mode, and that together with the current value of the program counter (represented by `rip` and `rcx` registers on x86), uniquely determined an event's timestamp. During replay, we used the somewhat imprecise performance-monitoring interrupt (PMI) mechanism to inject a logged event at a given time epoch (as also done in [27, 66]).

For comparison, we implemented a page-grained CREW mechanism similar to SMP-Revirt [27] inside KVM. We used the extended page table (EPT) hardware [35] and manipulated the present and read/write bits in the EPT page tables, to implement page-grained read/write CPU-level ownership of pages. This mechanism was sufficient to ensure a complete and robust deterministic replay of a full Linux multiprocessor VM, with its applications. The performance of this system however severely suffered for any application that involves a large amount of OS kernel activity (see results later), and also degraded significantly with increasing number of virtual CPUs.

We next implemented our DBT-based approach to record and replay the OS kernel. In all our experiments, the user-mode programs run as separate processes and do not share data. i.e., we demonstrate our ideas only for the Linux kernel (extending this to also record/replay application-level workloads using DBT will involve changing the OS loader to start in DBT mode, and to disable branch counting for user-mode execution). For completeness, we have also separately implemented a KVM-based tool that uses page-grained CREW-based record/replay

[27] for user-mode execution and our DBT-based record/replay for kernel-mode execution simultaneously. We have tested replay in both modes (with and without user-mode page-grained CREW) for hundreds of workloads running over a large period of time. While using user-mode page-grained CREW, we also tested multi-threaded user-mode applications which exhibit non-determinism due to sharing. All results in this thesis were generated for only the kernel (by turning off user-mode page-grained CREW).

Name	Description
`ppid`	Repeatedly calls `getppid()` system call.
`time`	Repeatedly calls `gettimeofday()` system call.
`creat`	Repeatedly calls `creat()` system call to create new files within one directory followed by `unlink()` system call to remove them.
`read`	Repeatedly calls `read()` system call on a single file.
`readdir`	Repeatedly calls `readdir()` system call on a single directory.
`ipc`	Creates two processes, producer and consumer. Producer sends a large amount of data to the consumer through a `pipe()`.
`socket`	Same as `ipc`, except uses TCP `socket()` instead of pipe.
`forkwait`	One process repeatedly creates a process and waits for it to exit.
`apache`	Uses the `ab` tool to send 80K requests at concurrency-level 50 to the Apache webserver running locally.

Table 4.1 Description of Benchmarks

We used BTKernel [36] as our DBT driver, which provides near-zero overhead DBT for the kernel. BTKernel was configured to instrument for training, record, and replay phases appropriately. The data-level locks were implemented by using a shadow-byte per memory-byte, to store the lock state. Our current implementation allows recording of up to seven concurrent CPUs (one bit per CPU in the shadow byte is used for ownership tracking, one bit is needed to ensure mutual exclusion among the slowpath routines). We have tested our implementation for up to four CPUs. All our experiments were run on an four-processor Intel Core i7 machine with 3GB RAM. We run a Linux VM over our KVM hypervisor with 1GB RAM, of which 512MB is used for shadow memory to store the data-locks. Also, all our experiments involved pinning kernel threads to their respective CPUs, as migration of threads between CPUs may incorrectly appear as sharing to our hypervisor, and would distort our results. Extending our implementation to support thread migration would involve instrumenting the context-switch logic in the kernel to appropriately manage the thread-to-CPU mappings.

Our training phase consisted of running several different kernel-intensive programs over a long period of time, to identify instructions that can share data, instructions that can potentially be involved in a data race (to identify monitors), and execution frequencies (for unifying

CPU0	CPU1	same/ diff file	KVM time (s)	Page-CREW		BTHybrid		BTData		BTCode	
				time (s)	log (MB)	time (s)	log (MB)	time (s)	log (MB)	time (s)	log (MB)
ppid	ppid		4.38	4.61	1.68	4.11	0.64	6.59	0.16	9.49	288.11
time	time		2.60	4.32	1.67	2.34	0.31	4.11	0.08	5.07	148.05
ppid	read		4.24	4.48	2.04	6.26	0.20	6.27	0.19	8.28	90.21
creat	creat	same	5.15	10.51	28.24	6.47	13.01	17.97	57.14	6.70	10.01
creat	creat	diff	3.66	8.46	12.82	6.87	13.01	14.95	56.05	6.18	11.01
read	read	same	2.73	6.40	2.82	7.10	94.04	6.90	87.04	5.69	94.04
read	read	diff	2.39	2.82	0.53	3.82	0.06	4.08	0.09	4.74	94.03
readdir	readdir	same	3.24	4.51	1.58	4.83	1.25	4.33	0.11	3.41	0.06
creat	readdir	same	2.97	5.32	4.43	5.93	0.19	6.17	0.20	4.58	1.14
ipc			17.93	75.64	55.81	19.64	314.71	35.87	348.25	28.80	150.36
socket			7.06	64.84	268.08	9.22	77.08	18.81	91.45	14.68	150.13
forkwait			9.60	114.10	637.82	13.53	54.23	23.20	69.02	16.45	47.79
apache			10.93	24.63	78.83	19.40	39.79	18.72	35.72	12.46	14.19

Table 4.2 Results when benchmarks are run in parallel on two processors. For the first few rows, different benchmarks were run on different CPUs; for the last few rows, the same benchmark used both processors. `KVM` column represents the time taken on unmodified KVM, `Page-CREW` represents page-grained CREW-based implementation of record/replay, `BTHybrid` represents our hybrid scheme that categorizes functions into CME/DME, `BTData` represents a scheme where all functions are marked DME, and `BTCode` represents a scheme where all functions are marked CME. The `time` column lists the execution time of the benchmark, and the `log` column lists the size of the record log produced. The `same/diff` column says if the benchmarks were run on the same or different files or directories, where relevant.

Benchmark running on all 4 procs	KVM time (s)	Page-CREW		BTHybrid		BTData		BTCode	
		time (s)	log (MB)	time (s)	log (MB)	time (s)	log (MB)	time (s)	log (MB)
read (same)	1.93	8.81	2.11	12.12	266.20	14.79	174.23	7.20	118.73
readdir (same)	3.70	8.57	31.15	4.74	1.21	5.01	1.51	4.28	1.34
ipc	10.92	71.11	94.37	21.52	331.31	22.56	339.14	32.27	200.01
socket	4.65	28.75	115.47	10.79	91.24	11.15	97.89	16.13	190.03
forkwait	6.51	270.01	2672.04	18.99	179.21	33.61	309.41	26.37	154.11
apache	10.73	212.54	1919.76	25.12	156.75	24.11	138.83	14.15	52.72

Table 4.3 Results when benchmarks are run on four processors. Column and cell labels have the same meaning as in Table 4.2.

monitors). While we do identify monitors in the Linux kernel, all our performance results in this thesis are with a single monitor (all code belongs to the same monitor). We found this to be a simple-yet-performant configuration in all our experiments (we show results with multiple monitors in Table 4.5). With one monitor, we do not need to instrument DME functions with `read_acquire()` calls on the monitor-lock; it suffices to acquire the monitor lock in read-mode at monitor entry (i.e., kernel entry) and release it at monitor exit (i.e., kernel

Benchmark 4 procs	Before		After	
	time (s)	log (MB)	time (s)	log (MB)
read (same)	17.89	559.97	12.12	266.20
readdir (same)	11.63	35.01	4.74	1.21
ipc	58.57	2877.80	21.52	331.31
socket	24.15	761.08	10.79	91.24
forkwait	39.41	1669.28	18.99	179.21
apache	40.46	1644.37	25.12	156.75

Table 4.4 Runtime overheads and logsize-growth before and after the optimizations involving lock implementation and disabling lock preemption on four processors.

Benchmark 4 procs	Before (s)	After (s)
read (same)	15.13	12.12
readdir (same)	4.86	4.74
ipc	20.58	21.52
socket	7.68	10.79
forkwait	140.51	18.99
apache	27.95	25.12

Table 4.5 Runtime overheads before and after optimization of control flow transfers. The logsize growth remains largely unaffected by this optimization on four processors.

Config	Description	Best-performing benchmarks
`Config1`	All functions marked DME, except `schedule()`, `do_page_fault()`, `release_pages()`, and `__mutex_lock_common()`.	`forkwait`, `ppid`, `time`
`Config2`	All functions marked DME, except `schedule()`, `do_page_fault()`, `release_pages()`.	`ipc`, `socket`
`Config3`	All functions marked DME, except `schedule()`, `do_page_fault()`, `release_pages()`, and all filesystem functions (e.g., `sys_read`, `sys_open`).	`read`, `readdir`, `creat`

Table 4.6 `BTHybrid` configurations used in our experiments

exit), allowing uninstrumented control flow transfers between DME functions. We tested our replay implementation over several workloads; in all our experiments, the replay succeeded in first attempt, probably because our training phase was exhaustive enough to identify all potential instruction conflicts. Our experiments on the Linux kernel demonstrate the robustness of our approach against several types of hidden synchronization patterns present in the target program.

We performed experiments to answer the following questions:

- What is the recording overhead and log-growth rate, and how does it compare with previous work?

- What is the comparison of code-level, data-level, and hybrid mutual exclusion approaches on the different workloads?

- How much do the optimizations discussed in Section 4.3 help?

Table 4.1 lists our benchmarks. We chose our benchmarks to exercise different subsystems of the kernel, and to exercise different types of sharing behaviour between the different subsystems. The `ppid` and `time` benchmarks primarily exercise the system call execution logic in the kernel. The `creat`, `read`, `readdir` benchmarks exercise the filesystem logic. The `ipc` benchmark exercises the inter-process communication subsystem using pipes, and the `socket` benchmark exercises the networking subsystem. `forkwait` exercises the process creation and destruction logic, and the virtual memory subsystem of the kernel. `apache` demonstrates performance on a larger benchmark. Most experiments involved running these benchmarks simultaneously on multiple CPUs, or running a pair of these benchmarks together, to observe their interference behaviour.

Table 4.2 shows runtime and record-log size results when these benchmarks are run on two processors. We ensured that while running simultaneously, each benchmark runs for around the same amount of time on unmodified KVM. Also, for filesystem-related benchmarks, we have two versions: one where the benchmarks operate on the same file/directory, and other where they operate on different files/directories. We compare the performance of our hybrid scheme (`BTHybrid`) involving both code and data locks, with unmodified KVM (`KVM`), page-grained CREW (`Page-CREW`), only data-based mutual exclusion (`BTData`), and only code-based mutual exclusion (`BTCode`). We manually categorized functions in the kernel as CME/DME for these results. We used three different configurations for CME/DME categorization. We show the best result among all three configurations for each benchmark in the `BTHybrid` column — note that the best configuration is often different for different benchmarks. As we discuss in Section 4.2.2, it may be best to dynamically perform this categorization at record time. Later in this section, we discuss our categorization procedure, and the sensitivity of our results to this categorization.

Most of our benchmarks show an improvement over page-grained CREW in both runtime and logsize. For `ppid-ppid`, `time-time`, `ppid-read`, and `read-read-diff`, `BTData` performs better than `BTCode`. On the other hand, `BTCode` performs better than `BTData` for `creat-creat`, `read-read-same`, `readdir-readdir-same`, and `creat-readdir-same`. Overall, benchmarks that share more data (e.g., `read-read-same`) perform better in `BTCode`, and benchmarks that share less data (e.g., `read-read-diff`) perform better in `BTData`. `BTHybrid` often performs better than *both* `BTCode` and `BTData`, as it uses a finer-grained choice of synchronization strategy. However, for some workloads, `BTCode/BTData` perform better than `BTHybrid`. Clearly, `BTHybrid` is more powerful than both `BTCode` and `BTData`, as the latter just represent two points in the configuration space of `BTHybrid`. However, all three configurations that we used for these experiments were worse than both `BTCode` and `BTData` for some benchmarks. Our manual identification of configurations is clearly not exhaustive, and a better approach would be to algorithmically and dynamically choose the configuration. We leave this for future work.

Table 4.3 shows runtime and record-log size results on four processors. We omit discussing benchmarks that do not provide interesting information over Table 4.2. The results show similar trends as on two processors, with our BT-based approach outperforming page-grained CREW on all benchmarks by up to 38x (11x average). The average slowdowns over unmodified KVM are 34% on two processors and 123% on four processors. The average decrease in logsize is 84% for four processors and 47% for two processors. `BTHybrid` often performs better than both `BTCode` and `BTData`; however, it also often performs worse (due to our limited set of configurations).

The log-growth size is a telling indicator of the sharing behaviour in the program. Larger logsize in `BTData` implies larger true sharing (in SMP-Revirt, the log size also includes false sharing). On the other hand, if the amount of sharing is small, `BTData` will generate a smaller log than `BTCode`, as `BTCode` generates a log entry each time another CPU enters the critical section, irrespective of whether sharing happened or not. `read-read-same` and `read-read-diff` in Table 4.2 illustrate this phenomenon. `read-read-same` has large true sharing because both the CPUs are accessing the same file that must be protected through some lock, resulting in large logsize for `BTData`. On the other hand, `read-read-diff` has little or no sharing because different files are protected using different locks, resulting in low log-growth rates for `BTData`. Notice that in both cases (`read-read-same` and `read-read-diff`), the critical section for `BTCode` is the same and hence both exhibit similar log-growth rates with `BTCode`.

Notice that the `readdir` benchmark has small log-growth rates. This is because the number of iterations of `readdir-readir-same` was significantly less (around 4K iterations) than the `read-read-same` benchmark (around 1500K iterations), for example, for a similar amount of time.

We next discuss the effect of our optimizations. Table 4.4 shows the runtime overhead before and after the optimizations involving lock implementation (Section 4.3.1) and disabling lock-preemption (Section 4.3.2) in hybrid mode. Recall that our instrumentation code reduced from ten instructions to two instructions, per shared-memory access, due to these optimizations. Evidently, the performance impact of these optimizations is significant, both due to the reduction in instrumentation code, and due to rate-limiting of ownership transfers (as indicated by the reduction in logsize).

Table 4.5 shows the runtime overhead before and after our optimizations involving optimizing control-flow transfers between functions (Section 4.3.3) and unifying monitors (Section 4.3.4). The extra overhead before these optimizations is due to the execution of extra instrumentation code during control-flow transfers between CME functions (control-flow transfers between DME functions were still uninstrumented). The most interesting result is that of `forkwait` which spends a large amount of time in CME functions (others spend most time in DME functions). For `forkwait`, the runtime improvement is primarily due to elimination of instrumentation code at function call boundaries. Benchmarks which spend a large amount of time in DME functions do not benefit from this optimization.

We next discuss CME/DME categorizations used for our experiments, and we discuss the procedure we used to arrive at them. Table 4.6 lists the configurations, and the benchmarks that performed best in that configuration. Initially, we marked all kernel functions DME. We then identified functions that show dense sharing patterns, and marked them CME. Also, we ensured that all functions in the callee tree of this function are executed in CME mode to avoid instrumentation overheads (as discussed in Section 4.3.3). If we found two functions that exhibit dense-sharing being called by a parent function, we mark the parent function CME, thus automatically causing all its children to execute in CME mode when executed through the parent. The first few functions we thus identified in the Linux kernel were: `schedule()`, `do_page_fault()`, `release_pages()`, and `__mutex_lock_common()`. Marking these functions CME improved the recording overhead of `forkwait` by 225% (`Config1`). Similarly, we found that `Config2` resulted in best performance for `ipc` and `socket`; while `Config3` resulted in best performance for all filesystem-related benchmarks. We have not explored the best configuration for `apache` yet, as it is a large benchmark and requires extensive analysis. We currently use `Config1` for the results on `apache`; we notice that in this configuration, the maximum data-lock faults for `apache` occur in the TCP stack of the Linux kernel. As a preliminary study, we marked all kernel functions starting with the strings "tcp_" and "ext3_" as CME, and that improved the `BTHybrid` performance on two processors for `apache` from 19.40 seconds (with around 40MB log) to 15.01 seconds (with around 27MB log).

4.5 Discussion

The flexibility of choosing the granularity and placement of locks is crucial to performance. While we present a DBT-based method to do this, it may be possible to extend other approaches to allow this flexibility. For example, SMP-Revirt could be extended such that it notices code regions that exhibit a large number of page faults, and uses code-level mutual exclusion for these regions. Using code-level mutual exclusion in this setting would mean stopping all other CPUs, and letting one CPU proceed at full speed (all pages mapped). There are two caveats to such extensions to page-grained CREW:

1. The cost of a "miss" still remains high (as it involves a page fault) and the number of misses due to false-sharing add to the problem. Also, false sharing can result in imprecise and confusing identification of regions which need to be protected through code-level mutual exclusion.

2. Identifying the termination of a region which needs to be protected through code-level mutual exclusion, is completely unclear. After a CPU is running at full speed (all pages mapped), we need to identify when to interrupt it to wrest control back and revert to page-grained CREW.

   One approach is to wait till the next system event (e.g., interrupt, page-fault, etc.). However, this approach can lead to long periods in time where only one CPU is executing, causing loss in performance.

   Another approach is to use hardware code-breakpoints to interrupt the CPU at desired code locations. Breakpoints and associated VM exits are expensive, and hence the practicality of this approach remains to be seen.

4.5.1 Fidelity

Deterministic replay systems potentially cause the target system to deviate from its native behaviour. We ignore deviations due to timing changes, e.g., page faults making certain interleavings less probable than others. Other deviations involve certain interleavings being completely prevented due to the deterministic replay system. All systems we have discussed so far, suffer from such fidelity limitations. We discuss them in turn below. We introduce a new metric, called *coarsening*-factor, to characterize the deviation caused by these systems. The higher the coarsening-factor, the more interleavings they thwart, and the more they deviate from native behaviour. High coarsening-factors have negative implications for applications like debugging and testing.

SMP-Revirt [27] is perhaps the least deviant from native behaviour — the only interleavings thwarted are the ones involving multiple operations within a single architectural instruction. For example, consider the increment instruction `inc MEM` on x86 (`MEM` represents a memory address). If two threads execute this instruction simultaneously, it is possible for the final value in MEM to be different for different executions — one instruction is sub-divided into two memory accesses (read and write), and a race is possible among these memory accesses. However, under SMP-Revirt, if two threads simultaneously execute this instruction, at-least one of them will trigger a page-fault. If the architecture implements precise exceptions (which the x86 architecture does), any interleavings involving multiple memory operations within a single instruction get thwarted. We say that the coarsening factor of SMP-Revirt is

$$\frac{one-machine-instruction}{one-memory-access}$$

In other words, page-grained CREW coarsens the atomic unit from one memory access to the granularity of one instruction.

Other approaches like Instant Replay [41] and Scribe [38] also have a coarsening-factor of one instruction. Approaches which rely on recording synchronization operations (e.g., RecPlay [57], Kendo [52]) coarsen the critical sections to the distance between two consecutive synchronization operations. If the program has no data-races and only lock-based synchronization, the coarsening factor is one. For general programs with data-races, the coarsening factor is > 1. Similarly, output-deterministic replay (e.g., PRES [55], ODR [4]) schemes do not coarsen the critical sections, under the assumption that the data-races are absent. These schemes are unique however, in that, even in the presence of data-races, they do not coarsen the recorded execution; however, data-races cause an explosion in search space during replay.

Approaches which involve recording the values returned by load instructions (e.g., iDNA [12]) can coarsen the critical sections depending on how they are implemented. In theory, they have a coarsening factor of 1.

ReSpec [42] and DoublePlay [62] divide execution into epochs. Each epoch is considered atomic, in that only interleavings among epochs are effectively allowed. Hence, the coarsening-factor for these systems is:

$$\frac{epoch-size}{one-memory-access}$$

With forward recovery [62], the coarsening-factor reduces to:

$$\frac{distance-between-two-observable-events}{one-memory-access}$$

The observable events in these prototype systems are system calls, so the numerator becomes “distance between two system calls”.

Our approach has two components: namely data-level and code-level mutual exclusion. Further, we implement data-level mutual exclusion in two ways: with preemptive locking, and with non-preemptive locking (Section 4.3.2). Data-level preemptive locking has a coarsening factor of

$$\frac{one-machine-instruction}{one-memory-access}$$

as a lock can only be acquired and released at the granularity of instructions — sub-instruction operations are treated atomically. Data-level locking without preemption has a coarsening factor dictated by the choice of poll points. Recall that we poll for outstanding lock-yield requests at VM exits, execution of the `pause` instruction, slowpath functions, and other statically-identified code locations. The average distance between poll-points depends on the average distance between these events, resulting in a coarsening factor of

$$\frac{distance-between-poll-points}{one-memory-access}$$

Code-level mutual exclusion on the other hand, coarsens the critical section to the size of the CME function (and its descendants). Contrasting with ReSpec and DoublePlay, while their approach coarsens to system call granularity, our approach coarsens to poll-point granularity. Depending on the target system, one may be better than the other.

Chapter 5

Conclusion

In this thesis, we explore the problem of deterministic replay from a software systems standpoint, exploring the different approaches possible and the tradeoffs involved. We show two broad approaches, namely code-level and data-level mutual exclusion. The tradeoffs involved in these approaches, are not very different from the classic tradeoffs between shared-memory and message-passing machines. We present a method based on dynamic binary translation, to explore this design space. Our method outperforms previous comparable approaches to deterministic replay. We also present a method to efficiently implement dynamic binary translation for the OS kernel, which involves handling of asynchronous interrupts and exceptions.

Data-level mutual exclusion is essentially a CREW (concurrent read, exclusive write) [22, 41] implementation at byte granularity. To implement CREW, we insert a reader/writer lock before every instruction that reads/writes shared memory. We use DBT-based shadow-memory implementation to implement byte-grained reader/writer locks. If a lock acquisition fails, we record this event so that during replay, we can acquire the lock in the same way. This scheme works well when there is relatively less true sharing. However, a monolithic OS kernel like Linux has code regions where several instructions are involved in dense true sharing. To deal with this situation, we introduce code-level mutual exclusion for such regions. Code-level mutual exclusion protects a large critical section. To support code-level mutual exclusion we identify regions of code (called monitors) which don't share memory, in an offline phase. Each monitor has its own reader/writer lock, which CPUs acquire and release on entry and exit to the monitor. For regions protected using code-level mutual exclusion, the monitor lock is acquired in write mode, and for regions using data-level mutual exclusion, the monitor lock is acquired in read mode. We categorize code-regions for code-level and data-level mutual exclusion at function granularity, called CME functions and DME functions respectively. We use DBT to achieve an efficient software-only implementation of this scheme.

The use of dynamic binary translation entails advantages over higher-level approaches that assume properties about the target program (e.g., knowledge of synchronization operations). DBT also alleviates the false-sharing problems of page-grained techniques. Previous DBT solutions (VMware, DRK) [2, 29] for the kernel has 3-5x overhead for many kernel intensive benchmarks. We implemented a kernel-level dynamic binary translator with near-native performance. Our design differs from VMware and DRK in its more efficient handling of exceptions and interrupts. Our design also allows dynamic switchon and switchoff of DBT for a running system.

Many hardware-based approaches propose hardware extensions for byte-grained mutual-exclusion. However, hardware-based approaches are not as flexible; for example, the flexibility of dynamic categorization of CME/DME functions, and appropriate instrumentation with code or data locks is not available. As we show, DBT provides an efficient method to achieve this. Also, DBT based approach can run on commodity hardware.

This work an be extended in at least two respects. First, we do not present a method to decide the monitors – we currently use a single monitor, and that works fine for the Linux kernel. A more complete approach would involve a profile-guided method to identifying monitors, keeping in mind the instrumentation overheads involved in crossing monitor boundaries. Second, we do not present an algorithm to categorize CME/DME functions. Our current configuration space for this categorization is hand-chosen for our benchmarks; it would be much better to have a dynamic and automatic method to achieve the same (and perhaps much better) results.

In summary, we present a method to efficiently record the shared-memory non-determinism present in a tightly-coupled shared-memory software system like a monolithic OS kernel. This is perhaps the first study of its type, as almost all published work on deterministic replay so far, has focused on application-level programs.

References

[1] BTKERNEL: Fast Dynamic Binary Translation for the Kernel. `https://github.com/piyus/btkernel`, as on September 15, 2013. pages 31

[2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS '06*. pages ix, 2, 4, 13, 14, 36, 38, 66

[3] Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, and Pratap Subrahmanyam. The evolution of an x86 virtual machine monitor. *SIGOPS Oper. Syst. Rev.*, 44(4), December 2010. pages 38

[4] Gautam Altekar and Ion Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP '09*. pages 1, 3, 8, 41, 42, 43, 46, 63

[5] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, pages 307–320, 2012. pages 41

[6] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *OSDI '10*. pages 10

[7] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *PADD '91*. pages 2, 11

[8] Joel F Bartlett. A nonstop kernel. *ACM SIGOPS Operating Systems Review*, 15(5):22–29, 1981. pages 41

[9] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS '10*. pages 10

[10] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for c/c++. In *OOPSLA '09*. pages 10

[11] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS '92*. pages 19

[12] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06*. pages 7, 41, 63

[13] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 17, pages 90–99. ACM, 1983. pages 41

[14] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under unix. *ACM Transactions on Computer Systems (TOCS)*, 7(1):1–24, 1989. pages

[15] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *SOSP '95*. pages 1, 5, 41

[16] Derek Bruening. *Efficient, Transparent and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004. pages 13, 14, 38

[17] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *VEE '12*. pages 13

[18] Edouard Bugnion. Binary translator with precise exception synchronization mechanism. US Patent 7516453, filed June 2000. pages 18

[19] Prashanth P. Bungale and Chi-Keung Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *VEE '07*. pages 13, 38

[20] Yufei Chen and Haibo Chen. Scalable deterministic replay in a parallel full-system emulator. In *PPoPP '13*. pages 11

[21] Jim Chow, Tal Garfinkel, and Peter M Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 1–14, 2008. pages 41

[22] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10), October 1971. pages 7, 65

[23] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *OSDI'10*. pages 10

[24] Jong deok Choi, Ton Ngo, John Vlissides, Bowen Alpern, Bowen Alpern, Manu Sridharan, and Manu Sridharan. A perturbation-free replay platform for cross-optimized multithreaded applications. In *In Int. Parallel and Distributed Processing Symp*, page 10, 2001. pages 1, 5

[25] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: Deterministic shared memory multiprocessing. In *ASPLOS '09*. pages 10

[26] 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 *OSDI '02*. pages 1, 5, 41, 50

[27] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *VEE '08*. pages ix, 1, 4, 7, 8, 41, 55, 56, 62

[28] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI '07*. pages 45

[29] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *ASPLOS '12*. pages ix, 2, 3, 13, 14, 18, 23, 31, 66

[30] Stuart I. Feldman and Channing B. Brown. Igor: A system for program debugging via reversible execution. In *PADD '88*. pages 1, 5

[31] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *PLDI '09*. pages 45

[32] Bryan Ford and Russ Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX ATC'08*. pages 38

[33] Nima Honarmand and Josep Torrellas. Relaxreplay: Record and replay for relaxed-consistency multiprocessors. In *ASPLOS '14*. pages 2, 11

[34] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA '08*. pages 2, 11

[35] Intel 64 and IA-32 architectures software developer's manual volume 3B: System programming guide part 2.
`http://www.intel.com/products/processor/manuals/`. pages 4, 55

[36] Piyus Kedia and Sorav Bansal. Fast dynamic binary translation for the kernel. In *SOSP '13*. pages 56

[37] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Intrusion recovery using selective re-execution. 2010. pages 41

[38] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *SIGMETRICS '10*. pages 1, 7, 8, 63

[39] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), July 1978. pages 45

[40] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2), April 1979. pages 44

[41] T.J. Leblanc and J.M. Mellor-Crummey. Debugging parallel programs with instant replay. *Computers, IEEE Transactions on*, C-36(4):471 –482, april 1987. pages 5, 7, 8, 41, 63, 65

[42] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient online multiprocessor replayvia speculation and external determinism. In *ASPLOS '10*. pages 1, 3, 9, 42, 43, 63

[43] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *SOSP '11*. pages 10

[44] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*. pages 38

[45] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA '08*. pages 2, 11

[46] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI '06*. pages 45

[47] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *ASPLOS '06*. pages 2, 11

[48] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05*. pages 41

[49] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007. pages 38

[50] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *PADD '93*. pages 2, 11

[51] Edmund B Nightingale, Daniel Peek, Peter M Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *ACM Sigplan Notices*, volume 43, pages 308–318. ACM, 2008. pages 41

[52] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS '09*. pages 3, 8, 63

[53] Marek Olszewski, Keir Mierle, Adam Czajkowski, and Angela Demke Brown. Jit instrumentation: a novel approach to dynamically instrument operating systems. In *EuroSys '07*. pages 38

[54] Douglas Z Pan and Mark A Linton. Supporting reverse execution for parallel programs. In *ACM SIGPLAN Notices*, volume 24, pages 124–129. ACM, 1988. pages 41

[55] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *SOSP '09*. pages 1, 3, 8, 41, 43, 46, 63

[56] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010. pages 41

[57] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2), May 1999. pages 3, 8, 10, 41, 43, 63

[58] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *PLDI '96*. pages 1, 5

[59] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15, November 1997. pages 45

[60] Dawn Song et. al. Bitblaze: A new approach to computer security via binary analysis. In *ICISS '08*. pages 13

[61] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *ATEC '04*. pages 1, 5, 41

[62] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: parallelizing sequential logging and replay. In *ASPLOS '11*, pages 15–26, 2011. pages 1, 3, 9, 10, 42, 43, 63

[63] Evangelos Vlachos, Michelle L. Goodstein, Michael A. Kozuch, Shimin Chen, Babak Falsafi, Phillip B. Gibbons, and Todd C. Mowry. Paralog: Enabling and accelerating online parallel monitoring of multithreaded applications. In *ASPLOS '10*. pages 2, 11

[64] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95*. pages 7

[65] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03*. pages 2, 11

[66] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*, volume 3, 2007. pages 1, 5, 41, 55

[67] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05*. pages 45