DYNAMIC BINARY TRANSLATION FOR DETERMINISTIC REPLAY

PIYUS KEDIA

AMARNATH AND SHASHI KHOSLA SCHOOL OF IT

INDIAN INSTITUTE OF TECHNOLOGY DELHI

NOVEMBER 2017

DYNAMIC BINARY TRANSLATION FOR DETERMINISTIC REPLAY

by

PIYUS KEDIA

Amarnath and Shashi Khosla School of IT

Submitted

in fulfilment of the requirements of the *Doctor of Philosophy*

to the

Indian Institute of Technology Delhi

November 2017

To *Nature*

Certificate

This is to certify that the thesis titled **"Dynamic Binary Translation for Deterministic Replay"** being submitted 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.
IIT Delhi

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 also like to thank the anonymous reviewers for their useful comments that made my thesis more comprehensive.

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 weird 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.

Piyus Kedia

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 [28]. 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 [28]. 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, we can not use it for improving deterministic 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 reads and writes. A potential approach is to use the CREW (Concurrent Read Exclusive Write) protocol at page granularity [27] to track the order of shared memory reads and writes. Page-grained CREW protocol uses hardware page protection techniques (Extended Page Tables/Shadow Page Tables) to restrict the access privilege of the CPUs. CREW dictates that multiple CPUs can read from a page by acquiring shared-access privilege (e.g., reader lock) of that page concurrently, but for writing to a page, they need to acquire exclusive-access (e.g., writer lock) privilege. Every transfer of privilege is recorded in order to reproduce the same transition during replay. This page-granular scheme, has been

demonstrated to work on selected user-level applications, but suffers from false sharing and huge shuttling between processors for workloads having a large amount of sharing (e.g., the Linux kernel). In contrast, we demonstrate an implementation of CREW at byte granularity using DBT to eliminate false sharing and achieve lower tracking overheads. To achieve this, we insert reader/writer locks before/after every shared memory access. We implement shadow memory using DBT, to store reader/writer locks (metadata) for each memory byte (data). This involves CREW-like ownership tracking of memory locations, which involves associating metadata (in shadow memory) with each memory location to store its ownership status. Our reader/writer lock implementation is optimized for the common case when one CPU acquires the same locks repeatedly.

Our implementation exhibits 3-9x recording overhead for several important kernel-intensive benchmarks on a four-processor machine, compared to 2-41x overheads of the best existing comparable approach.

सारांश

हम एक पूर्ण बहुप्रोसेसर आभासी मशीन (वीएम) जिसमें इसके अतिथि ओएस कर्नेल शामिल है, को रिकॉर्ड करने और निर्णायक रूप से पुनः चलाने के लिए एक कुशल सॉफ्टवेयर प्रस्तुत करते हैं| साझा स्मृति अखंड ओएस कर्नेल (जैसे लिनक्स) को निश्चित रूप से फिर से चालू करना एक बड़ी चुनौती है और हम इसे प्राप्त करने के लिए डायनामिक बाइनरी ट्रांसलेशन (डीबीटी) का उपयोग करते हैं|

डीबीटी एक शक्तिशाली तकनीक है, जिसके कई महत्वपूर्ण अनुप्रयोग हैं| सिस्टम-स्तरीय डीबीटी का उपयोग एक आभासी मशीन मॉनिटर [2] को चलाने करने के लिए और ओएस कर्नेल में इंस्ट्रूमेंटेशन [28] के लिए किया गया है। वर्तमान डिजाइनों में, कर्नेल-गहन वर्कलोड पर डीबीटी का प्रदर्शन निम्न स्तर का है। उदाहरण के तौर पर, [2] में सिसकॉल नैनोबेंचमार्क पर 10x से अधिक मंदी की सूचना दी गई थी, [28] एलएमबेंच के 2-5x मंदी के बारे में बताती है। यह मंदी मुख्य रूप से अतिरिक्त कार्य के कारण होती हैं, जो सही ढंग से कर्नेल तंत्र, जैसे कि इंटरप्ट, एक्सेप्षन और भौतिक (सीपीयू) संगामिति को संभालते हैं। चूंकि डीबीटी की लागत बहुत ही उच्च है, इसलिए हम निर्णायक रूप से एक बहुप्रोसेसर प्रणाली को पुनः चलाने के प्रदर्शन को सुधारने के लिए इसका इस्तेमाल नहीं कर सकते। हम एक कर्नेल-स्तरीय डीबीटी प्रस्तुत करते हैं, जो बड़े कर्नेल गतिविधि वाले अनुप्रयोगों पर भी शून्य लागत दर्शाती हैं। हमारे डीबीटी ने पारदर्शिता की आवश्यकताओं को सीमित किया है और मंदी के स्रोतों को खत्म करने के लिए अपरिवर्तनीय कर्नेल का आक्रामक रूप से लाभ उठाया है। हमने मूल लिनक्स में एक लोड करने योग्य मॉड्यूल के रूप में हमारे डीबीटी को निर्मित किया है, और बहुप्रोसेसर हार्डवेयर पर स्केलेबिलिटी प्रयोगों को प्रदर्शित किया है। यद्यपि हमारा कार्यान्वयन लिनक्स विशिष्ट है, हमारे तंत्र काफी सामान्य हैं; हम केवल विशिष्ट कर्नेल डिज़ाइन पैटर्न का लाभ उठाते हैं, लिनक्स-विशिष्ट विशेषताओं नहीं।

निर्णायक रूप से एक बहुप्रोसेसर प्रणाली को फिर से चलाने में सबसे बड़ी चुनौती साझा स्मृति पढ़ने और लिखने का क्रम रिकॉर्ड करना है। साझा स्मृति पढ़ने और लिखने के क्रम को ट्रैक करने के लिए पृष्ठ ग्रैन्युलैरिटी [27] पर सीआरईडब्ल्यू (समवर्ती पढ़ें अनन्य लिखें) प्रोटोकॉल एक संभावित दृष्टिकोण है। सीपीयू के एक्सेस विशेषाधिकार को प्रतिबंधित करने के लिए पृष्ठ ग्रैन्युलैरिटी सीआरईडब्ल्यू प्रोटोकॉल हार्डवेयर पेज सुरक्षा तकनीकों (विस्तारित पृष्ठ सारणी / छाया पृष्ठ सारणी) का उपयोग करता है|

सीआरईडब्ल्यू यह सुझाव देता है कि एकाधिक सीपीयू एक पेज से साझा-एक्सेस विशेषाधिकार प्राप्त कर सकते हैं (उदाहरण के लिए, पाठक ताला), लेकिन किसी पेज पर लिखने के लिए, उन्हें अनन्य प्रवेश (उदाहरण, लेखक ताला) विशेषाधिकार प्राप्त करने की आवश्यकता है| विशेषाधिकार का हर हस्तांतरण, पुनरावृत्ति के दौरान उसी स्थानांतरण को पुन: उत्पन्न करने के लिए दर्ज किया जाता है। यह पृष्ठ-ग्रैन्यूलर स्कीम, चयनित उपयोगकर्ता-स्तरीय अनुप्रयोगों पर कार्य करने के लिए दिखाया गया है, लेकिन यह झूठी साझाकरण से ग्रस्त है, और बड़ी मात्रा में साझाकरण (उदाहरण के लिए, लिनक्स कर्नेल) वाले वर्कलोड के लिए प्रोसेसर के बीच विशेषाधिकार के एक बड़ी संख्या में हस्तांतरण की संभावना है। इसके विपरीत, हम झूठे साझाकरण को खत्म करने और कम रिकॉर्डिंग लागत हासिल करने के लिये डीबीटी का उपयोग करके बाइट ग्रैन्युलैरिटी सीआरईडब्ल्यू प्रोटोकॉल की रचना करते हैं। इसके लिए, हम प्रत्येक साझा स्मृति एक्सेस के पहले/बाद में पाठक/लेखक तालों का उपयोग करते हैं। हम प्रत्येक स्मृति बाइट (डेटा) के लिए पाठक/लेखक तालों (मेटाडेटा) को संग्रह करने के लिए डीबीटी का उपयोग करके छाया मेमोरी की रचना करते हैं। इसमें सीआरईडब्ल्यू की तरह ही स्मृति स्थानों की स्वामित्व ट्रैकिंग शामिल होती है, जिसमें प्रत्येक स्मृति स्थान को उसके स्वामित्व की स्थिति को स्टोर करने के लिए मेटाडेटा (छाया मेमोरी में) के साथ संबद्ध जोड़ना शामिल होता है| हमारे पाठक/लेखक ताले सामान्य परिस्थिति के लिए अनुकूलित हैं, जब एक सीपीयू बार-बार एक ही ताले अधिग्रहित करते हैं|

हमारी रचना, चार-प्रोसेसर मशीन पर कई महत्वपूर्ण कर्नेल-गहन मानक के लिए 3-9x रिकॉर्डिंग लागत प्रदर्शित करती है, जबकि सबसे अच्छा मौजूदा तुलनीय दृष्टिकोण की लागत 2-41x है।

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Deterministic execution is an important property that simplifies the task of debugging and enables instruction-by-instruction debugging. Deterministic execution requires identical behavior of multiple executions of an application for the same set of inputs. This can be achieved in two ways:

- Logging the non-determinism during the original execution to reproduce the execution during the debug run.
- Enforcing determinism in a parallel application.

For this thesis, I focus on the first approach as it is more generally applicable and can be used to introduce determinism in existing software stacks. In this setting, the original execution is *recorded* by logging all potential non-determinism (also called non-deterministic inputs) in the system, such that these executions can be *replayed* by reading the non-deterministic inputs from the log file to reproduce the recorded execution. Non-deterministic inputs of interest include inputs that may change the logical behavior of the program. For example, interrupts, inter processor communication, shared memory interaction, device interaction, special instructions (e.g., `rdtsc`, `rdscp`), etc., can cause the application to behave differently during the replayed execution. We call the above set of events, non-deterministic, because they are external to the program's execution state, and need to be recorded for faithful replay. The non-deterministic events, which need to be logged varies, depending on the architecture (e.g., *uniprocessor* vs. *multiprocessor*), and on the granularity of record/replay (e.g., *full system* vs. *process*).

The machine state is *snapshotted* at the beginning of the execution, to ensure that the record and replay start with the same memory and register states. During the recorded execution, the non-deterministic events are logged for replay. Some non-deterministic events are

asynchronous, e.g., interrupts. Interrupts can occur at any time, so the *timing* of interrupts also needs to be recorded to enforce same behavior during replay. Other non-deterministic events are synchronous, e.g., device I/O reads (`in`, `ins`) and special non-deterministic instructions (`rdtsc`, `rdscp`). In these cases, only recording the input value is sufficient to reproduce the same execution during replay.

The timing of asynchronous non-deterministic events needs to be based on some logical behavior of the machine that can be reproduced identically during replay. For example, the number of retired instructions before an interrupt can be used to decide when to inject interrupts during replay.

For a uniprocessor full system record/replay system, the non-deterministic events are interrupts, device I/O, and special instructions. For a process level record/replay, recording the order of system calls and their return values are also needed for deterministic replay. These events can be logged very cheaply as demonstrated in previous work [16, 24, 26, 29, 54, 58, 63] and exhibit typical runtime overheads of less than 20% and log growth rates of less than 100 KBps (making it practical for use in production systems). However, deterministically replaying a multiprocessor system is harder because multiple cores can access the shared memory concurrently, which adds another major source of non-determinism. To replay a multiprocessor application, we also need to record the order of shared memory accesses. If all shared memory accesses are protected by a lock, then recording the order of lock acquisitions is sufficient to reproduce the order of shared memory accesses. But, real programs contain data races, which makes this problem challenging and interesting. Several ideas have been proposed to record multiprocessor applications efficiently. Some of the hardware based approaches [6, 31, 32, 43, 44, 46, 60, 62] require significant changes to the hardware. Another category of previous work assumes rarity of data races: PRES[51] and ODR[3] do not record all the data races and try to reproduce the data race in multiple replays. Some other approaches like DoublePlay[59] and ReSpec[41] detect the data races during record by executing and checkpointing another replica of the application concurrently, and rollback to a previous checkpoint, if the states of the two executions differ due to a data race. The above systems only support application level record/replay. System level record/replay is more challenging than application level record/replay because of 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 more 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.

SMP-Revirt [27] is perhaps the first system to support full system virtual machine deterministic replay. SMP-Revirt does not assume anything about the synchronization primitives and works for unmodified virtual machines. The authors of SMP-Revirt rely on hardware-based page protection techniques to capture the order of shared memory accesses. SMP-Revirt implements a CREW (concurrent reads exclusive write) protocol using a combination of page-protection techniques and per-processor *shadow page tables* [2]. The CREW protocol assigns an owner to every virtual page that gets accessed during runtime. Multiple CPUs can own a page in read-only mode, whereas for write access, a CPU needs exclusive access to the page. If the CPU does not have adequate permission to access a page, the record/replay subsystem logs an ownership transfer (also involves modification of page table entries) event before transferring the requested permissions to the current CPU. SMP-Revirt works well for applications having little sharing but suffers from huge performance degradation for applications having more sharing, due to a large number of ownership conflicts. Our work is based on the following observations:

- Tightly-coupled shared-memory style of programming in the kernel causes a lot of ownership conflicts and is unavoidable.
- A large number of ownership conflicts are due to false sharing because of the page-granularity limitation. If multiple cores try to access the same page at different indexes and at least one of them is a write operation, then the page-grained CREW protocol causes an ownership conflict even though there is no actual conflict in the real program.
- The cost of ownership transfer is very high. An ownership conflict causes a trap inside the hypervisor followed by more traps by other conflicting cores. A hypervisor trap is much more expensive than a system call, as it causes a *world switch*[18].

To address these issues, we propose a different recording scheme for the kernel. To eliminate false sharing, we maintain the ownership information for each byte, instead of each page (i.e., page granularity). Instead of relying on the hardware page protection technique, we take an instrumentation based approach. Every instruction, which accesses a shared memory location, is instrumented to check the ownership of the current access. If the current CPU does not

own the memory byte, then it executes a *slowpath* to acquire the ownership from the owner of that byte. Our instrumentation does not require any trap to the hypervisor, and thus the ownership conflict is also much faster than the page-protection based CREW.

To instrument unmodified binary code, we rely on Dynamic Binary Translation (DBT). DBT is an effective way to dynamically instrument an unmodified binary with lower overheads [17]. DBT is a technique which transforms the code as it executes. Thus the code can be dynamically instrumented before it executes. The main component of DBT is a *dispatcher*. The dispatcher translates one basic block at a time and transfers control to it. It then instruments the basic block to regain control after termination of the basic block. It then computes the address of the next basic block and jumps to it. The translate and execute loop continues until the program terminates. Several optimization techniques are introduced to do this efficiently (discussed in Chapter 3).

DBT requires different mechanisms for user-level applications and OS kernels. The existing system level DBT exhibits higher overheads, 2-5x slowdown for 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 low performance also includes the costs of other virtualization mechanisms, like memory virtualization through shadow page tables, etc. Another kernel-level binary translator, DRK [28], 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 cannot use it for any optimization. We implemented a kernel-level dynamic binary translator with near-native performance. Our optimizations result in a very different translator design from both VMware and DRK. We are more aggressive about assumptions on typical kernel behavior. In doing so, we sometimes relax "transparency" [1]; for us, ensuring *correctness* is enough. Essentially, we show that many transparency requirements are unnecessary and can be relaxed for better performance. Like DRK [28], our translator works for the entire kernel including arbitrary devices and their drivers.

Our design differs from VMware and DRK in the following important ways:

- On an interrupt or exception, the current program counter (PC) value is pushed on the stack by the hardware. To maintain transparency, the translated code should store the original PC on the stack. VMware and DRK replace the kernel entry points with a call to the dispatcher, which jumps to the translated handler after restoring the original PC on

[1] Transparency means that a translated code should never observe a different state from its native execution.

the stack. In our design, we replace the kernel entry points with the translated address of their handlers. In doing so, we allow the translated code to observe non-native state.

- Both VMware and DRK emulate 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 beginning of the translation of the next native instruction. In our design, we allow imprecise exceptions and interrupts.

- 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.

We used our DBT framework to instrument the Linux kernel for efficient deterministic replay. We assign an owner to every byte in the main memory. All instructions that can potentially access a shared memory are instrumented to check ownership before accessing the memory. If the current CPU does not own the memory location, it automatically acquires the ownership from remote CPU(s) and logs this event to reproduce the same effect during replay. The ownership information corresponding to a memory byte is stored in the shadow byte, which is kept at a constant offset from the original byte. We achieve significant performance improvement over our SMP-Revirt[26]-like implementation of page-grained CREW, through this DBT-based scheme.

The thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 discusses our DBT algorithm, implementation and results. Chapter 4 describes our byte-grained CREW algorithm. It also discusses our implementation and results. Finally, we conclude this thesis in Chapter 5.

[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 behavior for architectures that support precise exceptions (e.g., x86).

Chapter 2

Related Work

Several hardware and software approaches for deterministic replay have been proposed in the past. Deterministic replay can be implemented for a process, or for the full system (an entire operating system with applications). In a uniprocessor environment, to deterministically replay a process we need to snapshot the process's address space at the beginning, the return values of system calls (including kernel writes to application memory), and the values returned by special instructions like `rdtsc`, `rdscp`, etc. For a full system uniprocessor deterministic replay, we need to additionally record the I/O-port reads, interrupts and their timings, and DMA-based I/O activity (e.g., network packets). The overhead of recording these events is typically less than 20% as demonstrated by several previous studies [16, 24, 26, 29, 54, 58, 63]. However, multiprocessor deterministic replay is much harder due to the need to record the order of shared memory reads and writes. Many software-only approaches have been proposed to capture this type of shared-memory non-determinism, and we discuss them in this chapter.

Instant Replay [40] logs the order of all shared memory accessed during the program execution. Instant replay implements a CREW-like (concurrent read, exclusive write) protocol to record the order of shared accesses. The CREW protocol allows concurrent reads to a shared object, but writes to a shared object are restricted to a single thread at any time. To implement CREW, Instant Replay acquires a reader/writer lock before every read/write access. Instant Replay 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, so that during replay, the same version number of the shared object gets accessed. 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. Every object contains additional fields: `version_number`, `total_readers`, `active_readers`, and `sema (semaphore)`. During record, `semaphore` acts as a writer lock. Before writing to an object,

```
reader_entry(object, process):
1  if (record) {
2    down(object.sema);
3    atomic_add(object.active_readers, 1); /* writer waits if readers are active */
4    up(object.sema);
5    write_log(process, object.version); /* record the object version */
6  }
7  else if (replay) {
8    recorded_version = read_log(process);
9    /* during replay wait for writers to update this version to the logged version */
10   while(object.version != recorded_version);
11 }

reader_exit(object):
1  if (record || replay) {
2    /* update the total number of readers for current version of object */
3    atomic_add(object.total_readers, 1);
4    if (record)
5      atomic_dec(object.active_readers, 1);
6  }

writer_entry(object, process):
1   if (record) {
2     down(object.sema);
3     while (object.active_readers != 0);  /* wait until all the readers finish */
4     write_log(process, object.version);  /* record the current object version */
5     write_log(process, object.total_readers); /* record total reads for this version */
6   }
7   else if (replay) {
8     recorded_version = read_log(process);
9     /* during replay wait for writers to update this version to the logged version */
10   while(object.version != recorded_version);
11   total_readers = read_log(process);
12   /* during replay wait for all the readers to read this version of object */
13   while(object.total_readers < total_readers);
14  }

writer_exit(object):
1 if (record || replay) {
2   object.total_readers = 0; /* reset the total reads on version update */
3   if (record) {
4     object.version++;  /* update the object version */
5     up(object.sema);
6   }
7   else
8     atomic_add(object.version, 1);
9 }
```

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

the writer acquires the semaphore (line-2 in writer_entry) and releases it after the completion of the write (line-5 writer_exit). The `active_readers` variable acts as a reader lock. Every reader increments the `active_reader` count when it enters the read *critical section*. The writer waits until the `active_readers` count is zero before writing to the object. Before incrementing `active_readers`, the readers wait for all the writers to finish. This is ensured by acquiring `semaphore` at line-2 in readers entry. The above mechanism guarantees mutual exclusion between the readers and writers and also prevents the starvation of the writer (if some readers are active all the time). The reader releases its reader lock at line-5 in `reader_exit` by decreasing the `active_reader` count. `total_readers` counts the total number of reads of an object for a particular version number. During record, in the read *critical section*, the readers log the current version number of the object. During replay, the readers wait until the version number of the object is same as it was during record. During record, in addition to the version number, the writer also logs the current value of `total_readers`. During replay, before writing to the object the writer first waits for the version number of the object to reach its recorded version, and then it waits for all the readers to access the current version of the object as it was during record. After writing to the object, the `writer_exit` routine updates the version number of the object. The above mechanism ensures that all readers and writers access the same version of the object during record and replay. The instrumentation overhead of Instant Replay is high — the amount of instrumentation code inserted 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 drastically reduced the amount of instrumentation code required around each memory access.

iDNA [12] logs the value returned by every load instruction. Because 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. In the case of cache conflict, before logging the load value, it also logs the value of the hit counter to correctly identify the load during replay. Recording the load values is sufficient to accurately replay the recorded sequence, even if 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, 40], every page is assigned an owner for exclusive access, or a set of owners for shared read accesses. SMP-Revirt

records CREW conflicts by using hardware page protection support. Through page table manipulations, a page is mapped only in the address space of its owners. The 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. SMP-Revirt implements deterministic replay for the virtual machine. It uses hardware page protection techniques on shadow page tables to implement CREW. For a virtual machine, CPUs are virtualized (called virtual CPUs). The hypervisor maintains one shadow page table corresponding to each page table inside the guest OS. SMP-Revirt modifies the shadow page table implementation, to create one copy of shadow page table corresponding to every virtual CPU. To implement CREW, the hypervisor maps a physical page with different privileges in per-CPU shadow page tables. The SMP-Revirt approach treats the target program as a black-box, and therefore it is possible to use this scheme 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 more 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 19x on four processors. Further, as also discussed in [27], this scheme scales very poorly with increasing number of processors.

Scribe [38] implements CREW in the kernel to do process-level deterministic replay using thread-private page tables. In UNIX-like kernels, one `page-table` is created for every process. To record/replay a process, Scribe creates one copy of page tables corresponding to every thread of the process. Pages in the thread-private page tables are mapped and unmapped based on the CREW protocol. The difference between Scribe and SMP-Revirt is: SMP-Revirt does the whole system record/replay, whereas, Scribe does record/replay for a process. To record/replay, a process kernel-mode execution is not recorded; instead, recording the order of system calls and process interaction between kernel through memory is enough to reproduce the same execution. This means that, unlike SMP-Revirt, overheads of recording shared memory access are not included in Scribe overheads. The Scribe authors evaluate their scheme on a variety of benchmarks like Apache webserver, Linux build, etc. Compared to SPLASH2 benchmarks [61] (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 SMP-Revirt implementation for a virtual machine (full-system) imposes 19x overhead on four processors! Almost all overhead in

this benchmark is observed in kernel-mode execution. We present a more efficient scheme to record and replay such workloads.

Instant replay [40], 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 [53] assumes data-race free programs and works by recording only explicit synchronization operations. Kendo [49] 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* behavior. Probabilistic Replay with Execution Sketching (PRES) [51] and Output-Deterministic Replay (ODR) [3] are examples of such systems, where the behavior 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 behavior 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 [3] 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 can 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 [41] 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 is 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 [59], 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 a 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 a large number of data races in that epoch; in that case, if the divergence check fails at the end of the epoch, 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). One problem with this Respec [41] and DoublePlay [59] is they don't record any arbitrary kind of execution. DoublePlay can record only those sequences within an epoch which can be achieved by running those threads sequentially. ReSpec can omit some record sequences, where a data-race may result in a divergence of state in the recorded and replayed execution during a time interval. Due to this limitation, some bugs may remain hidden during record that were otherwise possible in the native run.

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 [59], 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 [53] 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 records 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, 42], or enforcing determinism at the system level [5] 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 [19] — 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 [19] do not translate directly to supporting efficient deterministic replay on bare-metal executions. While several hardware optimizations have also been proposed for deterministic replay [6, 31, 32, 43, 44, 46, 60, 62], we restrict our attention to software-only approaches in this thesis.

SMP-Revirt [27] comes closest to our goals of recording a full multi-core virtual machine (including the OS and its applications), towards a faithful replay, without making stronger assumptions on the absence or rarity of data-races in software. We re-implemented SMP-Revirt [27], using nested-page-tables [13] (the original implementation used shadow page tables), and found that one of the main reasons for the slowdown is false-sharing. Because SMP-Revirt works at page-granularity, it is possible that two threads that are accessing disjoint locations on the same page, result in unnecessary ownership transfers. We employ an instrumentation-based approach like [40] in an attempt to minimize recording overheads. We use dynamic rewriting techniques on unmodified binary code, to be able to instrument the code running inside a VM, for efficient recording. However, existing binary rewriting tools for instrumentation impose very high overheads for kernel-intensive workloads. Towards this, we propose a novel system-level binary instrumentation technique that significantly outperforms existing methods. In the rest of this section, we discuss existing dynamic binary translation (DBT) frameworks.

A DBT framework transforms the code as it executes. Both, user-mode and kernel-mode DBT framework exist today. DBT has several applications, including, but not limited to, profiling, debugging, cross-architecture translation, security, etc. QEMU [8] is a full system emulator that can emulate an entire operating system with applications and uses DBT. QEMU provides a hardware abstraction to the target operating system (guest OS). The interactions between guest device drivers and hardware are emulated in software. From the guest's perspective, it talks to the real device but in reality, QEMU proxies for the device. Similarly, the memory management unit (MMU) is also emulated in software to provide portability across all architectures. The high-level goal of QEMU is to provide a common platform for cross architecture translation. QEMU generates intermediate code (in its intermediate representation) from the input instruction set and then generates the target code for the host instruction set from this intermediate code. The intermediate code is optimized through custom optimization passes implemented within QEMU. However, this multi-layer translation imposes significant overheads, even if the input and the target architectures are the same. For example, QEMU imposes 5-20x slowdown for regular applications for x86-to-x86 translation.

VMWare's paper on comparing hardware and software techniques for virtualization [2] shows that full system emulation is possible at a small cost if the input and target instruction sets are largely same. For example, VMWare's virtual machine monitor (VMM) reports

only 2.9 % overheads for compute intensive SPECInt benchmarks. The primary insight between VMWare's software-based virtualization is that if the input and target instruction sets are largely same, most instructions can be run natively, without any extra DBT overhead. In fact, VMWare runs the entire user-mode code natively without any instrumentation. The only part that needs to be instrumented is the guest kernel. The guest kernel interacts with the hardware through privileged instructions and memory mapped I/O. These privileged instructions need to be emulated. Typically, the DBT *dispatcher* (the component of the DBT software that orchestrates the whole translation) maintains *shadow state* corresponding to the guest OS's privileged state (e.g., interrupt flags), in VMM's memory. Instructions that read-/write the privileged state in the guest OS are translated to instead emulate the functionality by reading/writing to the shadow state. Another important component of an OS is its memory management unit (MMU), which also needs to be virtualized. VMWare's VMM creates a shadow page table corresponding to each page table in the guest. The VMM identifies a page table load, by tracking the `mov_to_cr3` instruction. Modifications to the page-table are tracked through write-protecting the pages corresponding to guest page tables. The VMM also interposes on the kernel entries to emulate the corresponding hardware behavior, enabling access to the kernel pages, and running the system in translated mode afterward. The VMM also ensures precise exceptions and interrupts. Precise interrupts and exceptions are the property of the hardware that ensures the delivery of exceptions and interrupts at a valid instruction boundary. For an exception, the valid instruction boundary is the start of the excepting (and partially executed) instruction, whereas, in the case of interrupts, it is the next instruction boundary. In a typical DBT environment, an instruction can be translated into multiple instructions – to ensure the precise exception property, the changes made by previous instructions in the translated set must be rolled-back before injecting the exception. In the case of precise interrupts, the delivery of interrupts must be delayed until all the instructions in the translation set have finished their execution. These mechanisms are costly and discussed in details in VMWare's paper [2] that compares the software virtualization approach based on DBT, with hardware-assisted virtualization.

Figure 2.2 shows the data from the VMWare's paper[2]. The nano-benchmarks repeatedly execute the same opcode in a loop. The two nano-benchmarks, `divzero` and `syscall` incur the overheads of `262%` and `853%`. These overheads confirm that the kernel entries and exits are very expensive. Other micro-benchmarks that do a lot of paging activities – incur high overheads due to MMU virtualization. On the other hand, `SPEC-Int` and `kernel-compile` spend most of the time in user-mode and has smaller overheads. The webserver, `apache` is slow due to network activities that involve a lot of interrupts. These results confirm that the kernel-level DBT has very high overheads and not suitable for kernel intensive workloads.

Data from “Comparison of Software and Hardware Techniques for x86 Virtualization”
K. Adams, O. Agesen, VMware, ASPLOS 2006.

VMware’s Software Virtualization Overheads

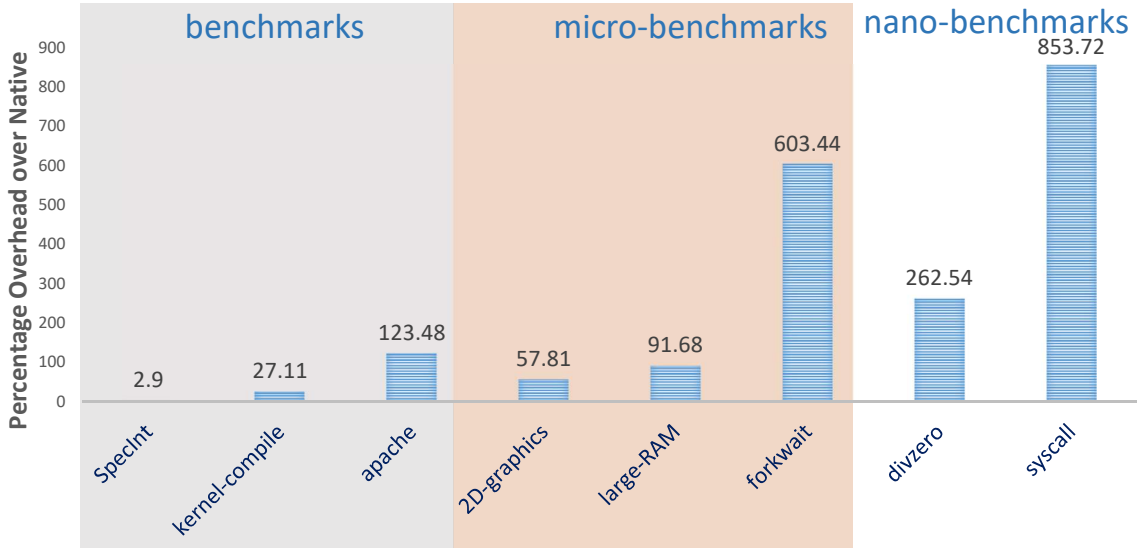


Fig. 2.2 VMware’s software virtualization overheads.

However, VMware VMM does a lot of extra work to virtualize a guest OS, that is not needed, if we only want to translate the kernel.

Another work, DRK[28], implements the DBT framework for the kernel as a loadable module inside the guest OS. Unlike VMWare, the goal of DRK is not virtualization but instrumentation. DRK also emulates precise exceptions and interrupts similar to VMWare, but does not emulate privileged instructions. Similarly, DRK does not implement MMU virtualization (i.e., no shadow page tables). Figure 2.3 shows overheads of DRK for several kernel intensive benchmarks. DRK authors report overheads up to 350% for workloads like fileserver, webserver, webproxy, varmail, and apache. These overheads are primarily due to the additional mechanism needed to ensure transparency. Recall that, by transparency, we mean that a translated code should never observe a different state from what it would have observed during the native run.

Ideally, a translated system must run at near-native speed. Low-overhead user-level DBT is well understood (e.g., [17]). For this thesis, we focus on kernel-level DBT. We find that some of the transparency requirements, enforced by previous DBT systems, are unnecessary and the Linux kernel, for example, does not depend on such guarantees. With slightly-more relaxed transparency, we achieve near-native performance for kernel-intensive benchmarks. We discuss our optimizations for efficient dynamic binary translation in Chapter 3. Our de-

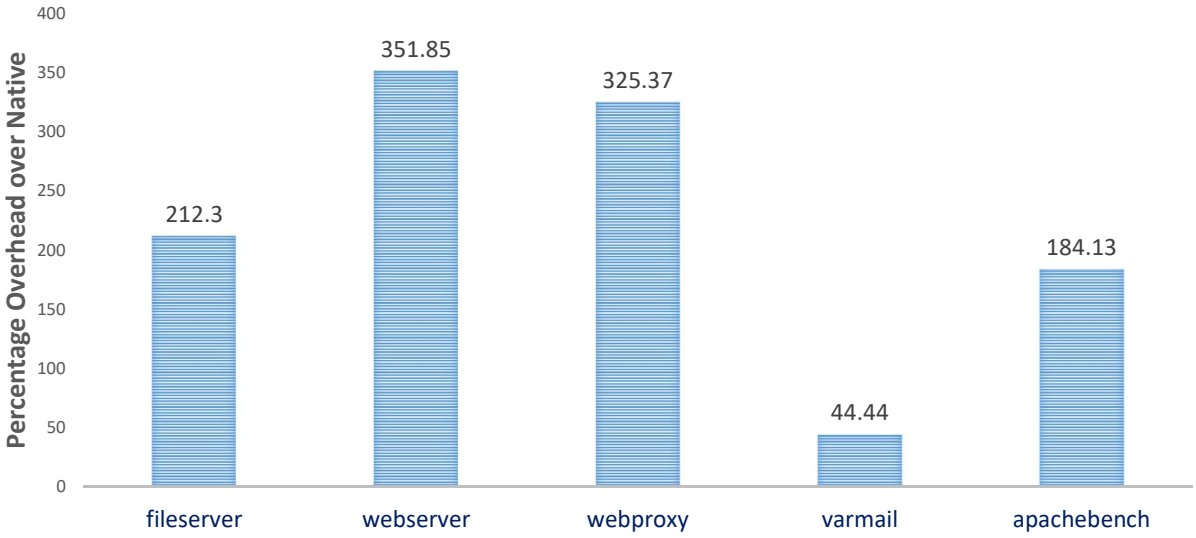


Fig. 2.3 Dynamo-Rio kernel (DRK) overheads.

terministic replay scheme implemented on top of DBT-based instrumentation is discussed in Chapter 4.

Chapter 3

Fast Dynamic Binary Translation for the kernel

3.1 Background

Dynamic binary translation (DBT) is a mechanism that allows transformation of code as it executes. Translation is done at the granularity of *basic blocks*. A basic block is a sequence of assembly instructions with only one *branch instruction* at the exit. A branch instruction jumps to a target basic block (the address could be in memory, register, and in the instruction itself). Translation at the basic block granularity allows the dispatcher to translate only those instructions that are going to execute. In `x86` assembly, it is possible that after executing a branch instruction the execution never executes the subsequent instruction, therefore the translation stops at a branch instruction. On the other hand, all the instructions within a basic-block are guaranteed to execute and hence they are translated all at once. Interestingly, a basic block may not be the most-efficient granularity for translation (explained later in this chapter).

For comparison with VMMs, we also use the term guest for the native code (i.e., the code which is going to transformed dynamically). We use **tx-pc** to denote the translated address corresponding to the native address **pc**. Figure 3.1 shows the various components of a DBT framework.

- **Dispatcher:** The dispatcher is the main component of a DBT framework. The dispatcher executes in the same address space as its guest, and yet it hides from the guest. The dispatcher implements two important functionalities.
  - **Disassemble:** The dispatcher takes a program counter as input and disassembles the basic block starting at the the program counter.

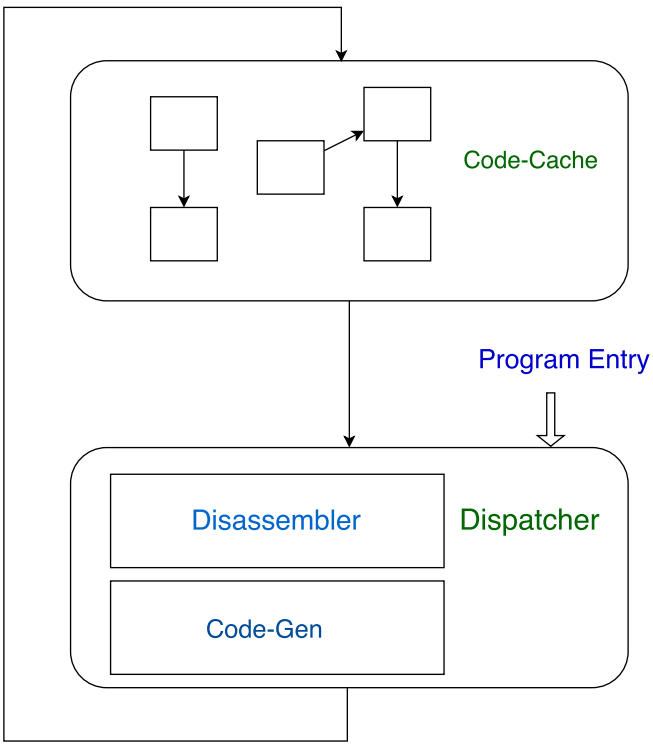


Fig. 3.1 A DBT framework

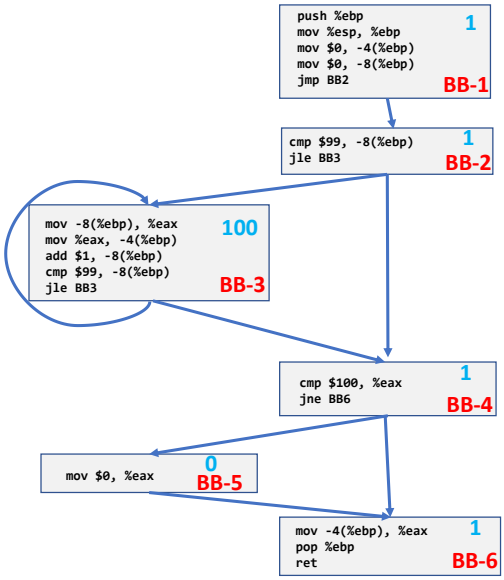
  - **Code-gen:** After disassembling, the dispatcher generates the output code corresponding to the input basic block. The output code is produced in a way that the dispatcher regains control after the execution of the translated basic block. The translation of the last branch instruction sets the next program counter value before jumping back to the dispatcher. The dispatcher then translates the basic block starting at the next program counter and jumps to it. This translate and execute loop continues until the program terminates.

- **Code-cache:** Another component of a DBT framework is the *code-cache*. A typical guest executes the same basic block multiple times. Already translated basic blocks are cached into a code-cache to reduce translation overheads. Before translation, the dispatcher first looks in the code-cache for an existing translation. If a translation already exists, the dispatcher jumps to the target block; otherwise, it translates and caches the basic block in the code-cache.

To understand the working of a dispatcher let's look at a simple program as shown in Figure 3.2a. This program computes the sum of all natural numbers which are less than 100. After, the computation the routine checks whether the sum is equal to 100 and sets the result to zero if the equality check succeeds. Notice, this will never happen, and we will see that this

```
main() {
   int i, sum = 0;
   for (i = 0; i < 100; i++)
      sum += i;
   if (sum == 100)
      sum = 0
   return sum;
}
```

(a) A sample program with loops.



(b) A control flow graph of basic-blocks corresponding to the `main` routine.

```
BB1
   push %ebp
   mov %esp, %ebp
   mov $0, -4(%ebp)
   mov $0, -8(%ebp)
   jmp BB2
BB3
   mov -8(%ebp), %eax
   mov %eax, -4(%ebp)
   add $1, -8(%ebp)
BB2
   cmp $99, -8(%ebp)
   jle BB3
BB4
   cmp $100, %eax
   jne BB6
BB5
   mov $0, %eax
BB6
   mov -4(%ebp), %eax
   pop %ebp
   ret
```

(c) Assembly code corresponding to the `main` routine.

Fig. 3.2 An illustrative example of dynamic binary translation.

piece of code is never going to be translated.

The output assembly is shown in Figure 3.2c. The control flow graph(CFG) of the basic blocks are shown in Figure 3.2b. The numbers in the basic blocks represent the total number of executions of the given basic-block throughout the execution of the program. `BB1` is the entry basic block which branches to `BB2` after execution. `BB2` branches to the loop body (`BB3`) after checking the loop condition. Because this condition is true, the program executes the loop body (`BB3`) until the loop condition is met (100 times), and finally branches to `BB4`. `BB4` checks for another condition on the computed sum (which is not true) and jumps to `BB6`. `BB5` never gets a chance to run during the lifetime of the program.

Let's see how the dispatcher executes this program in Figure 3.3. The dispatcher takes an entry basic block (`BB1`) as input for dynamic execution. This example shows a simple use

case of a DBT framework, where it does nothing special but generates the identical behavior for every instruction. We will see that, even in this simple use case, the translation of a single instruction may contain multiple instructions. BB1(Figure 3.3a) terminates at a direct branch instruction whose target is BB2. Notice that the dispatcher cannot create an identical translation for the jmp instruction, because otherwise it will start executing original code. Therefore, the dispatcher emits translation code to store the jump target in a variable called nextpc. The nextpc variable resides in the dispatcher address space. A branch instruction is emitted to regain control after the execution of the basic block. After regaining control, the dispatcher reads the target basic block using nextpc, translates, and executes it.

The next basic block BB2(Figure 3.3b) contains a conditional jump instruction. A conditional jump instruction has two potential targets. If the condition is met during runtime the control flow goes to the jump target, i.e., BB3, in this example, otherwise, the next instruction (BB4) is executed. To handle two different targets, in the translation of a conditional jump instruction, we add an extra jump instruction to jump to the next basic block, for the case when the condition is not met during runtime. In this example, after the execution of BB2, BB3 is going to execute next.

BB3 contains the main body of the loop. Notice, that BB3 also contains a fragment of already translated code (BB2), but we have no way to reuse that translation because the first instruction in these basic blocks is different. Because of the sharing of code among basic blocks, some translated code is duplicated in the code cache. While duplications can be avoided by inserting more branches in the translated code, most DBT frameworks avoid this to reduce runtime overheads. In the example, the loop body in BB3 will execute 100 times. After the first execution, subsequent executions of the same basic block reuse the existing translation from the code-cache (assuming enough space in code-cache to store all the translations) [1].

When the loop condition is not true anymore, the dispatcher is asked to execute BB4. In this program, BB4 never sets the nextpc to BB5, and therefore, BB5 remains untranslated during the lifetime of the program. Finally, the dispatcher translates the last basic block (BB6). This basic block contains an indirect jump instruction ret, whose target is determined at runtime. We explain indirect branch handling mechanism later in this chapter.

The dispatcher and guest code executes in the same address space but the invocation of dispatcher is completely transparent to the guest. To facilitate this, guest state must be saved before a jump to the dispatcher and the dispatcher must restore this guest state before returning to the translated code. Also, the dispatcher and guest codes must execute on different stacks.

Figure 3.4 shows the pseudo code of the dispatcher. The guest state is stored in the dis-

[1] In case of memory pressure, a replacement policy is enforced to evict translated basic blocks from the code cache. In this case, a cache miss of already translated basic block causes retranslation of the basic block

```
push %ebp
mov %esp, %ebp
mov $0, -4(%ebp)
mov $0, -8(%ebp)
jmp BB2
```

```
push %ebp
mov %esp, %ebp
mov $0, -4(%ebp)
mov $0, -8(%ebp)
set nextpc to BB2
jmp dispatcher
```

(a) Input and output assembly generated by the dispatcher for the entry basic block. Before, jumping to the dispatcher, the output code sets the `nextpc` to `BB2`, which is used by the dispatcher to translate and execute `BB2`.

```
cmp $99, -8(%ebp)
jle BB3
```

```
cmp $99, -8(%ebp)
set nextpc to BB3
jle dispatcher
set nextpc to BB4
jmp dispatcher
```

(b) Dispatcher reads `BB2` from `nextpc` and generate output code. Notice, that this basic block contains a conditional jump instruction which has two targets. The output code includes an additional jump instruction to handle the case when the conditions were not met during the actual execution.

```
mov -8(%ebp), %eax
mov %eax, -4(%ebp)
add $1, -8(%ebp)
cmp $99, -8(%ebp)
jle BB3
```

```
mov -8(%ebp), %eax
mov %eax, -4(%ebp)
add $1, -8(%ebp)
cmp $99, -8(%ebp)
set nextpc to BB3
jle dispatcher
set nextpc to BB4
jmp dispatcher
```

(c) This basic block is the main body of the loop. This loop body executes 100 times. After the first execution, the output code is reused from the code-cache in subsequent executions.

```
cmp $100, %eax
jne BB6
```

```
cmp $100, %eax
set nextpc to BB6
jne dispatcher
set nextpc to BB5
jmp dispatcher
```

(d) The interesting property about this basic block that it never sets the `nextpc` to `BB5`. As a result, the dispatcher never translates `BB5`.

```
mov -4(%ebp), %eax
pop %ebp
ret
```

```
mov -4(%ebp), %eax
pop %ebp
tx-ret /* discussed later */
```

(e) This is the final basic block. Notice, the translation of the `ret` instruction is somewhat complicated, as the return target is stored in the stack that is only known at runtime. We are deferring this discussion for now.

Fig. 3.3 Dispatcher orchestrating the dynamic execution of the loop example.

```
dispatcher()
{
   save guest registers in dispatcher memory
   switch to dispatcher stack
   tx_pc = lookup_code_cache(nextpc, found)
   if (!found)
      tx_pc = translate(nextpc)
   restore guest registers from dispatcher memory
   jmp *tx_pc
}
```

Fig. 3.4 Dispatcher: Dispatcher first saves the guest registers in its memory and then switch to its own stack. It then searches for `nextpc` in the code cache. On cache miss, dispatcher does the actual translation, restore guest registers, before making an indirect jump to the translated basic block.

patcher memory and then the dispatcher switches to its own stack. If the translation corresponding to `nextpc` does not exist in the code cache, the dispatcher translates the basic block starting at `nextpc`, adds this basic block to the code cache, and jumps to the target code after restoring the guest state.

Even with the code cache, calls to the dispatcher are not cheap. In Figure 3.3, `BB3` makes 100 calls to dispatcher, and this can be prevented by dynamically linking the translated basic blocks. An alternate translation for `BB3` is shown in Figure 3.5, and is called *direct branch chaining*. The central idea is to modify the jump to dispatcher instruction, to jump to `tx-nextpc`, when the dispatcher is called. Because jumping to the dispatcher is not a single `jmp` instruction (e.g., setting `nextpc` is also required), we put the jump to dispatcher stub in a separate location (also called an *edge*), and use the original branch instruction (conditional or direct jump) to jump to `edge`. Now, when the dispatcher is invoked through the `edge`, the dispatcher patches the target of the jump to `edge` instruction with `tx-nextpc`. To patch the target, the dispatcher also needs to know the address of the instruction, which needs patching. For this, the dispatcher keeps an additional variable `prevpc` to store the address of jump to edge instruction as shown in Figure 3.5. `prevpc` is also stored in the dispatcher address space.

In Figure 3.5, after the first execution of `BB3`, the conditional jump instruction branches to `edge1`. The dispatcher finds `BB3` in the code cache, and subsequently checks if `prevpc` is set. In this case, `prevpc` is set to P1. The dispatcher replaces the target of the instruction at location P1 (i.e., `edge1`) with `tx-BB3`. On the subsequent execution of `BB3`, if the condition is met, `BB3` will directly jump to itself, without making any call to the dispatcher. After the 100 iterations, the dispatcher will get called again through `edge2`. This time, the dispatcher would translate `BB4` and patch `edge2` at P2 with `tx-BB4`.

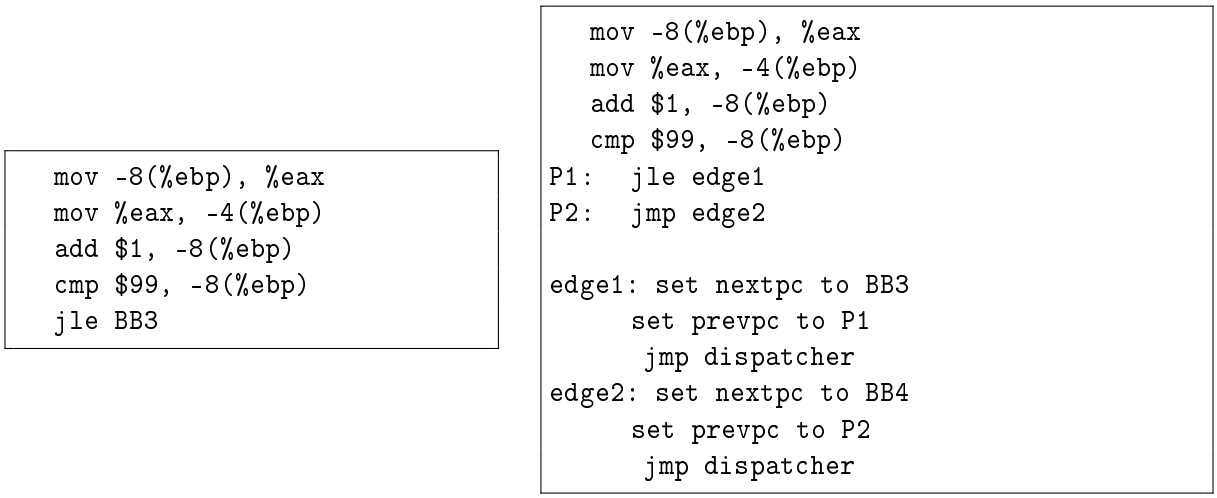


Fig. 3.5 Direct branch chaining: Translated basic blocks are linked together for efficiency. The dispatcher overwrites the target addresses `edge1` and `edge2` with `tx-BB3` and `tx-BB2` after translating the respective basic blocks.

The translation of a conditional branch instruction contains one extra jump instruction (Figure 3.5), which adds extra overheads as compared to native execution. To eliminate the cost of this additional `jmp`, the dispatcher may translate a *code block* (also called trace) instead of a basic block. A code block does not terminate on a conditional branch instruction. Due to this, the translation of a code block may contain multiple edges. One drawback of this approach is: spurious instructions (which are never going to execute) are also translated and added to the code-cache. Let's see what happens if we translate a code block instead of a basic block in our loop example in Figure 3.3. In Figure 3.6, we translate a code block starting at `BB3`. It turns out that this code block only terminates at function return. In Figure 3.5, we need `edge2` because `tx-BB4` and `tx-BB3` could be at different locations in the guest address space, and hence an extra jump is required to execute them in sequential order. In a code block, we allocate them consecutively to eliminate the extra jump. One drawback, as you can see in Figure 3.6, is that we also translate `BB5`, even though it is never going to execute. Interestingly, due to this optimization, the output code block looks similar to input code block. We use code blocks instead of basic blocks, to minimize runtime overheads due to extra branches.

We next discuss how other branch instructions, e.g., `call`, `ret`, and indirect jumps and calls are translated.

A direct `call` instruction is translated to push the next program counter value on the stack followed by a `jmp` to the target procedure. A direct `call` instruction is also chained with the target procedure using `direct branch chaining` (Figure 3.7).

However, a `ret` instruction is tricky as the return address is known only at runtime. The

```
   mov -8(%ebp), %eax
   mov %eax, -4(%ebp)
   add $1, -8(%ebp)
   cmp $99, -8(%ebp)
   jle BB3
   cmp $100, %eax
   jne BB6
    /* BB5: redundant */
   mov $0, %eax
   mov -4(%ebp), %eax
   pop %ebp
   ret
```

```
   mov -8(%ebp), %eax
   mov %eax, -4(%ebp)
   add $1, -8(%ebp)
   cmp $99, -8(%ebp)
P1:  jle edge1
   cmp $100, %eax
P2:  jne edge2
   mov $0, %eax    /* redundant */
   mov -4(%ebp), %eax
   pop %ebp
   tx-ret

edge1: set nextpc to BB3
     set prevpc to P1
      jmp dispatcher
edge2: set nextpc to BB6
     set prevpc to P2
      jmp dispatcher
```

Fig. 3.6 Code block: does not terminate at a conditional branch instruction. e.g., a code block starting at BB3 in loop example only terminates on function return.

```
call target
```

(a) Input: A call instruction pushes the next pc on the stack before jumping to target pc.

```
    push nextpc
P1: jmp edge

edge:
set prevpc to P1
set nextpc to target
jmp dispatcher
```

(b) Output: Instead of using the call instruction (because it will push the translated address; lose transparency) next pc is pushed using a different instruction.

Fig. 3.7 Translation of a function-call instruction. The native code is a single instruction, shown in Figure 3.7a. The translated code is shown in the Figure 3.7b. The translated code saves the value of the PC of the next instruction (in the instruction stream) on stack, just as the hardware would do for the `call` instruction. The `edge` block transfers control to the dispatcher.

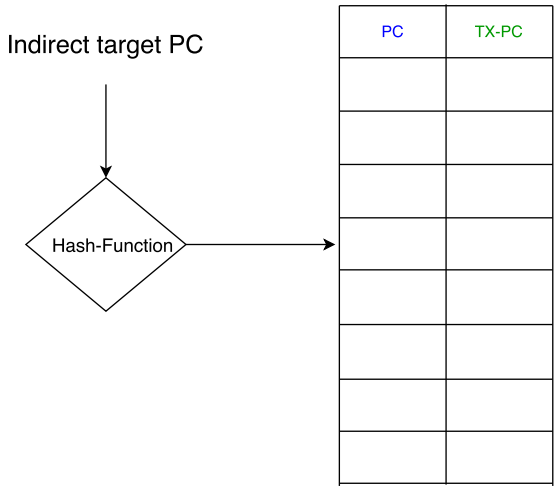


Fig. 3.8 At runtime, a hash function is applied to the target pc to index into the jumptable. The jumptable is a hash-table storing the mapping between a native PC value (`pc`) and its translated code-cache address (`tx-pc`).

`ret` instruction is translated similar to an indirect branch instruction. An indirect branch instruction target is only known at runtime. The target jump address is available in either a register or a memory location. The `x86` architecture supports several indirect branch instructions, e.g., `call *reg`, `call *MEM`, `jmp *reg`, and `jmp *MEM`. `reg` and `MEM` operands store the target in a register and memory location respectively. One way to handle this is to call the dispatcher every time an indirect branch instruction is encountered. The dispatcher can look into the code cache to find the corresponding translation and directly jump to it. However, dispatcher calls are very slow. For efficiency, the dispatcher maintains a *jumptable* (see Figure 3.8): the jumptable contains mappings from `pc` to `tx-pc`, and is simple enough to be indexed through a few lines of assembly code. The indirect branch instruction in Figure 3.9 is translated to perform a fast lookup in the jumptable first and jump directly to the code-cache address if a match is found; otherwise, it calls the dispatcher. If the dispatcher is invoked due to an indirect branch, then the dispatcher adds the corresponding mapping to the jumptable to avoid future calls to the dispatcher. In other words, the jumptable acts as a cache for the most frequently seen jump targets at that jump instruction. Because the jump table is small, a hash collision may evict an existing entry to make space for a new entry. Dispatcher calls occur on jumptable misses.

The translation of an indirect `call` instruction is similar to the indirect `jmp` instruction, except that it additionally pushes the next native program counter on the stack, before computing and branching to the target address.

A `ret` instruction is also an indirect branch instruction. The target return address value

```
Input:
/* jump target of an indirect jump is stored
 * in memory, which is only known at runtime
 */
jmp *MEM

Output:
/* A small cache of original and translated pc is maintained
 * in memory for faster lookup during runtime. On a cache
 * miss a somewhat slower jump to dispatcher is made
 */
save temporary registers and flags
jtarget = lookup_hash(MEM)
if (found)
{
  restore registers and flags
  jmp *jtarget
}
set nextpc
restore registers and flags
jmp dispatcher

Dispatcher:
save guest state
tx_pc = disassemble_and_code_gen(nextpc)
add nextpc, tx_pc to the jumptable
restore guest state
jmp *tx_pc
```

Fig. 3.9 The translation of an example indirect instruction, `jmp *MEM`. The native code is shown in the `Input` block. The translated code is shown in the `Output` block. The translated code looks up the jumptable (call to `lookup_hash()`); if found, it jumps to the corresponding translated code address (`jtarget`); if not found, it jumps to the dispatcher. The dispatcher does a complete lookup to determine the translated address for `nextpc`. If not already translated, the dispatcher translates the code block starting `nextpc` before restoring the guest state, and jumping to the translated address.

is obtained from the stack at runtime and handled similarly to the indirect instruction. For a user-mode DBT framework, apart from these instructions, the `syscall` and `signals` also needed to be handled correctly to run translated code, when the CPU re-enters the user-mode [17].

3.2 Kernel-mode DBT

Unlike user-mode DBT, kernel-mode DBT imposes new transparency requirements, and correct handling of them incurs high-performance overheads. VMware's Virtual Machine Monitor (VMM) [2] translates the kernel mode execution of a guest virtual machine to safely execute multiple untrusted virtual machines on commodity hardware without hardware virtualization support (this was required when hardware support for virtualization was not available). VMWare's paper[2] reports 10x slowdowns for the `syscall` benchmark. Another related work, DRK [28], implements a kernel-mode DBT framework as a loadable module inside the Linux kernel, and reports 2-5x slowdowns on kernel intensive benchmarks. A typical VMM requires more mechanisms than a kernel module DBT framework. For example, unlike a kernel module, a VMM has to implement shadow page tables and emulate privileged instructions in software to precisely emulate correct behavior. However, DBT is useful for many other applications (apart from virtualization), such as debugging [57, 65], profiling [64], monitoring [37], and several applications in security [47, 56]. We intend to employ DBT for the implementation of an efficient record/replay system.

One of the problems with both the approaches is – they can not be used for any optimizations because of their high overheads. In this section, we discuss the different transparency requirements for kernel-mode execution and the mechanism required to handle them correctly.

The transition from user-mode to kernel-mode happens due to interrupts, exceptions, and system calls. DBT framework for the kernel needs to take control at these entry points to run the kernel in translated mode. The interrupts and exceptions push the current program counter value on the stack and jump to the corresponding handler as specified in the interrupt descriptor table (IDT). If an interrupt or exception occurs during the execution of the translated code, the hardware pushes the code-cache address on the stack. For transparency, the code-cache address needs to be replaced with the native address before jumping to the translated handler. DRK [28] replaces the entries in the IDT with the address of the dispatcher, to invoke the dispatcher on kernel entry points. The dispatcher then substitutes the code-cache address with its native counterpart before branching to the target code. Similarly, `iret` must return to a code-cache address possibly by treating it as an indirect control transfer or calling the dispatcher before `iret`.

The x86 hardware ensures *precise exceptions and interrupts*. If an exception triggers during the execution of an instruction, then the precise exception property ensures that the changes made by the partially executed instruction are rolled back. Similarly, the precise interrupt property delays the delivery of the interrupt to the next instruction boundary when an instruction is interrupted midway. These properties need careful consideration while implementing DBT for the kernel. Both these properties do not affect the guest behavior if the translated instruction is identical to the original instruction (because in this case, the hardware would obey the properties anyways). However, in a typical DBT framework, an instruction may be translated into several instructions (we call this the *translated set* of instructions); in this case, it becomes the responsibility of the DBT framework to ensure that the preciseness properties of exceptions and interrupts are obeyed.

If an exception triggers during the execution of the translated set, then the corresponding changes made by the previous instructions in the translated set must be rolled back before injecting the exception (to emulate precise exceptions). For precise interrupts, the delivery of interrupts must be delayed until all the instructions in the translated set have executed. The rolling-back and delayed injection of exceptions and interrupts are expensive. This is because of a direct cost involved in rolling back, and an indirect cost of maintaining the data structures to support rollback. One way of implementing late injection is to dynamically insert a software interrupt instruction after the last instruction in the target set [28].

Another transparency requirement is, the system should not miss any interrupts. Apart from the translated code, the dispatcher is also invoked frequently for various reasons. If an interrupt triggers during the execution of the dispatcher, then it must be queued for reinjection.

Overall, these techniques are complex and expensive and discussed in detail in [2, 28]. In our experience with the Linux kernel, we find that the OS does not depend on most of these transparency requirements in the common case. However, there are some special cases where it does, and these cases can be handled specially. Relaxing these requirements simplifies the design of the DBT framework with huge performance benefits. However, it imposes new reentrancy and concurrency issues as discussed in the next section.

3.3 A faster design

At the high level, we relax precise exceptions and interrupts requirements, allow code-cache addresses to live in the guest stack, and identity translate `call` and `ret` instructions to achieve performance improvements over previous work. Identity translation of `call` and `ret` instructions avoids the need to translate the ret instruction into an indirect-branch jumptable-lookup. Because, the ret instruction is quite commonly executed, this results in significant performance

savings. However, this implies that code-cache addresses can now live on the guest stacks, and can persist across context-switches. Further, code-cache addresses can live in the guest stack due to our different handling of interrupts and exceptions, as we discuss below.

Relaxing these properties allows the guest code to observe some state that it would never have observed during native execution. If the guest kernel depends on any of these properties, then the translated execution will behave incorrectly. In our experiments with modern OSes, we find that an OS rarely depends on some of these invariants, barring a few special patterns. Fortunately, such special patterns can be handled specially, through constructs provided by these modern operating systems. We list below some correctness issues that may arise due to our DBT scheme:

- The first correctness issue arises due to the presence of code-cache addresses on the guest stack. If the guest code depends on the original program counter values on the stack, then the guest code will behave incorrectly. One example in the Linux kernel where it happens is the page-fault handler, which reads the program counter value from the interrupt frame pushed on the stack and takes a decision based on the virtual address of the program counter[2]. We discuss this special case and a study of other such cases for different operating systems in Section 3.5.

- The second correctness issue also involves code-cache addresses living in kernel data structures. A typical DBT system invalidates code-cache addresses for a number of reasons (e.g., memory pressure, adaptive optimization etc.). Because our design allows code-cache addresses to be present in guest stacks, the invalidation of a code-cache address would require replacing all of them with the fresh translation of the corresponding code block. This requires an atomic walk through the scheduler thread list, followed by unwinding and fixing the return addresses. In our design, we disallow cache invalidation. In our experiments with the Linux kernel, we found that 10 MB memory suffices for all the kernel code executing a various range of kernel intensive workloads and normal desktop applications. In cases where we indeed require cache invalidation, we propose a novel scheme discussed in Section 3.7.

- The third correctness concern arises due to the precise exception and interrupts requirement. We do not emulate precise exception and interrupts. We also disable interrupts during the execution of the dispatcher. Interestingly, OS code rarely (if ever) relies on these hardware properties. This allows significant simplification of the DBT framework, resulting in better performance. By ensuring bounded running times of the dispatcher, we can ensure that an interrupt is not lost, even if it occurs during dispatcher execution.

[2]For example, the Linux kernel allows certain regions of its code to trigger a page-fault, but panics otherwise.

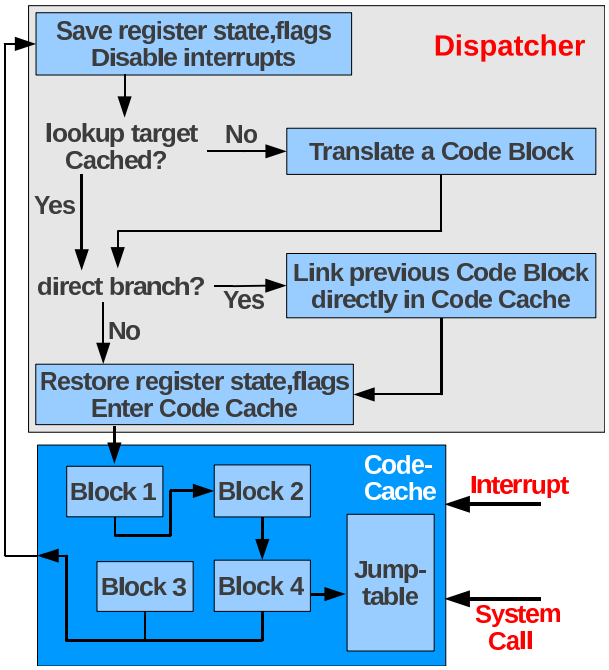


Fig. 3.10 DBT framework for the kernel.

Notice that the NMI (non-maskable interrupt) interrupt cannot be masked, and hence it can occur in the middle of the dispatcher. NMI interrupts are usually used for debugging purposes. We have tested our implementation with the NMI handler running natively, and it does not affect Linux kernel execution with regular applications. It is possible to disable NMI for the full system [3] if it is not required by the guest OS.

Figure 3.10 shows a complete diagram of our DBT framework for the kernel. Notice that we replace the kernel entry points in the IDT table directly with its translated counterpart. This allows direct execution of the translation block on interrupts, exceptions, and system calls.

3.4 Design subtleties

Various reentrancy and concurrency issues can arise due to our design, and we discuss them in this section. Consider the dispatcher code in Figure 3.11, where it translates a code block and jumps to the target block. For simplicity let us assume *uniprocessor* execution.

The first problem is, where to save the guest state. Recall that a part of the guest state is emulated by storing it in memory, and instructions accessing this emulated state are translated accordingly. If we store this emulated guest state in a shared location, then an interrupt

[3] NMI can be disabled by setting NMI bit (13) at the local vector table(LINT0) address in local APIC register.

```
1. save guest state and flags
2. disable interrupts
3. switch to dispatcher stack
4. jtarget = lookup_code_cache(nextpc)
5. restore guest state and flags
6. jmp *jtarget
```

Fig. 3.11 The dispatcher code uses a shared variable (`jtarget`) to store the translated PC, before restoring the guest state and indirectly jumping to the address stored in `jtarget`. Between lines 5 and 6, the interrupts may be enabled, which may cause re-entrancy issues on access to the shared variable `jtarget`.

before line-2 could cause the translated interrupt handler to run directly at this point, and an exit to the dispatcher could overwrite the previous guest state. This is a reentrancy issue and to handle this, we save the guest state to the guest stack itself. This could potentially cause a transparency issue because the guest code can now observe some values on the stack that it would never have observed while running natively. Most kernel code usually uses the guest's interrupt stack in a bracketed call/return (or interrupt/iret) fashion. And hence, simply storing an extra value at interrupt delivery, and restoring it at interrupt return (by translating the `iret` instruction appropriately) would suffice in most cases. However, there are some rare scenarios where the interrupt/exception handler redirects the execution to somewhere else, and such cases need to be handled specially. We discuss such rare scenarios and our solution in Section 3.5.

The second reentrancy problem could arise due to the shared variable `jtarget` at line-6. Notice that the dispatcher restores the guest flags at line-5. This could enable interrupts and an interrupt just before line-6 directly calls the translated handler, which could potentially call the dispatcher at some point. This nested call to the dispatcher overwrites the `jtarget`, and the interrupted code would end-up executing a wrong translated block.

To handle this scenario, we save the `jtarget` at the kernel entry/exit points. This is similar to how interrupts handlers save/restore registers before and after an interrupt. In addition to saving and restoring guest registers and stack, we also save and restore the `jtarget` on kernel entry and exit points.

The Linux kernel maintains an interrupt context data structure that consists of all the states needed to be saved and restored. However, adding an extra entry (`jtarget`) to existing structure would require modification to the source code. Instead of adding a new entry, we repurpose existing unused entries in the structure for saving `jtarget`. We found that `ds` segment value is always identical for every transition to and from kernel space. We use the `ds` slot

in the interrupt context for saving and restoring `jtarget`. For simplicity, if we use `jtarget` elsewhere in this thesis, we are referring to this special variable that is safe to access, across interruptions. On a multicore system, concurrent calls to the dispatcher could also overwrite `jtarget`. To prevent this, we use a `per-CPU` location for saving `jtarget`[4].

It may not always be possible to find scratch space to store `jtarget` for any arbitrary kernel. In this case, a different translation (Figure 3.12) can be used to avoid the reentrancy problem. In this scheme, the dispatcher steals a register (by saving it to the stack) to store `jtarget`. The translated code block would then need to be prepended by a stub to first restore the register value before executing the translated code block. We have not implemented or evaluated this scheme.

A similar reentrancy problem could arise due to the handling of indirect branches. Figure 3.13 shows the pseudo-code corresponding to the indirect branch instruction. Notice that the `lookup_hash` is not thread-safe even if on a single core system. This is because reads and writes to the hash-table are not atomic. The updates to the hash-table are performed in the dispatcher. To disallow the invocation of dispatcher during hash-lookup, we disable interrupts just before the hash lookup in the assembly code. Because the hash-table is always mapped in memory, this piece of code never triggers any exception, and thus would run atomically with respect to the dispatcher, on a uniprocessor. On a multiprocessor system, a concurrent dispatcher could also update the hash-table. To avoid this conflict, we use a `per-CPU` hash-table. The second problem at lines 6-7 is similar to the indirect branch problem in the dispatcher, as discussed before (an interrupt can occur between the restoration of flags and the indirect jump at line 7). We handle this by using a `per-CPU` `jtarget`.

Apart from these issues, conflicts may arise if the dispatcher and the guest share the same data structures, e.g., if the dispatcher uses the guest's `malloc` function. In this case, several race conditions may arise, e.g., if the guest's `malloc` function acquires a lock, and the translated code calls the dispatcher after acquiring that lock (from within the malloc() function), a subsequent call to `malloc` by the dispatcher (for allocating its own memory) would result in a deadlock (as the dispatcher will try and acquire the same lock). To prevent this reentrancy issue, we do not reuse the existing libraries of the guest OS; instead, the dispatcher implements its own libraries which share nothing with the guest code.

```
/* An alternative translation can be used, if
 * we do not find unused space in the kernel's
 * interrupt stack frame for saving/restoring
 * jtarget
 */

1. save guest state and flags
2. disable interrupts
3. switch to dispatcher stack
4. tx_pc = lookup_code_cache(nextpc)
5. restore guest state
6. push ecx     /* A register is always safe to use
                 * because the interrupt handlers
                 * save and restore registers during the entry and
                 * exit of an interrupt handler
                 */
7. mov tx_pc, ecx
8. restore flags
9. jmp *ecx

translated code-block:
pop ecx             /* restore the register value if it was
                     * called through an indirect branch instruction
                     */
original translation
```

Fig. 3.12 The dispatcher code uses a register `ecx` to save the translated code cache address (`tx_pc`). To deal with re-entrancy issues, the dispatcher saves the register to stack, before returning to stack. The first instruction of the translated block, pops the register `ecx` from the stack. The extra stub instruction to pop the register value, is prepended to translated code block. This mechanism protects against potential re-entrancy issues caused due to an interrupt occurring during the control transfer from the dispatcher to the code cache.

```
Input:
jmp *MEM

Output:
1.  save temporary register t1, t2 and flags
2.  mov MEM, t1
3.  disable interrupts
 /* Linux uses fs segment for per-CPU variables */
4.  %fs:jtarget = lookup_hash(t1)
5.  if (%fs:jtarget) {
6.      restore t1, t2 and flags
7.      jmp *%fs:jtarget
8.  }
9.  set nextpc (uses %fs:jtarget)
10.  restore t1, t2 and flags
11. jmp dispatcher
```

Fig. 3.13 An example translation of an indirect instruction. Interrupts need to be disabled before `lookup_hash()` call to avoid race conditions on the jumptable (hash table) with the dispatcher. Similarly, a per-CPU `jtarget` is used to avoid race conditions due to multi-core concurrency. Interrupts may get enabled between lines 6 and 7, causing a race condition on `jtarget`. This race condition is solved by saving/restoring `jtarget` as discussed in Section 3.4.

3.5 Potential inconsistencies due to code-cache addresses living in guest data structures

Our design allows code-cache addresses to be present in guest stacks. This enables faster translation, but at the same time creates correctness concerns. We studied a number of operating systems (Table 3.1) and found out cases where the guest execution depends on the program counter value pushed by the interrupt/exception handler on the stack. In our Linux kernel implementation, we found that the page fault handler depends on the PC (program counter) value of the faulting instruction. The Linux kernel memory is always mapped in the page-tables, and the kernel expects page-faults only in certain regions of the kernel code (`copy_from_user`, `copy_to_user`) identified by their PC values. These kernel-space routines access user-space memory. The compiler emits information about the PC values of these potentially excepting instructions, in the form of an exception table. The page-fault handler checks if the faulting PC belongs to one of the PCs stored in the exception table, and raises a panic if not. If the faulting PC belongs to the exception table, the kernel handles the fault accordingly, by invoking the corresponding exception handler. In our design, the PC value pushed on the stack is the

[4]Linux kernel uses %fs segment register to implement `per-CPU` variables

OS	Unconventional uses of the interrupted/excepting program counter 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.

code-cache address, which would be (incorrectly) absent in the compiler generated exception tables.

Similar patterns, where certain exception handlers are sensitive to the excepting program counter 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 program counter 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

program counter 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 3.14. 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` block, an exception frame containing the exception handler pointer is pushed onto 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 [5]. Because on an exception inside the `__try` block, the kernel's exception handler overwrites the excepting program counter, our DBT design can incorrectly cause execution of native untranslated code.

Fortunately, such patterns are few and can usually be handled as special cases, as we describe below.

On Linux, a kernel module can also have its own exception table. If the address is not found in the mainline kernel's exception table, then the page-fault handler looks for the address inside the modules' exception tables. We use this mechanism to our advantage. During translation, if the program counter address is the part of the kernel or module exception table, we add the corresponding translated entry in our DBT module exception table. This ensures correct behavior on kernel page faults.

Similarly, DBT for kernels implementing RAS can be handled by manipulating the RAS registry to include the translated RAS regions too. 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 check

[5] 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 program counter.

```
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. 3.14 Pseudo-code showing registry of custom page fault handlers by kernel subsystems in BSD kernels. The `pcb_onfault` variable is set to the program counter 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`.

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[6].

In general, we believe that for a well-designed kernel, any interrupt or exception handler whose behavior depends on the value of the interrupted program counter value, should ideally also allow a loadable module to influence the handler's behavior because the program counter 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 program counter 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 new prob-

[6] 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.

lem, 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, extra 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 program counter pushed on stack against translations of these hardcoded addresses. If found equal, the program counter value pushed on the stack should be replaced by their corresponding native code address. Similar checks should be added to interrupt return, with the 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.

3.6 Optimizations

We implemented more optimizations to further improve the performance of DBT.

3.6.1 Call-ret optimization

The biggest remaining overhead of a DBT framework are the extra instructions introduced during the translation of indirect branch instructions. In our design, handling of indirect instruction requires disabling of interrupts, followed by a lookup in a *jumptable*. All of these are expensive operations. The most common type of an indirect branch instruction is a `ret` instruction (function return). Even after doing all the other optimizations, the indirect handling of `ret` instruction could impose overheads of up to 2-3x. Most `call` and `ret` instructions execute in bracketed fashion to implement function calls and returns. However, some exceptions exist, e.g., `longjmp` and `setjmp` routines. We find that the Linux kernel always uses these instructions in a bracketed fashion to implement function calls and returns, and its logic does not depend on the actual PC values pushed on the stack. Based on these observations, we *identity-translate*[7] `call` and `ret` instructions. The `call` instruction pushes the code-cache address of the next instruction on the stack and `ret` directly pops the code-cache address and jumps to it. We do not terminate a code block on call instruction, i.e., a code block may contain multiple `call` instructions, which makes the translation of a `call` instruction very cheap and does not require extra `jmp` instructions. Figure 3.15 shows a sample code block with multiple `call` instructions. After this optimization, the output code block is identical to the input block

[7]The translated instruction is identical to the original instruction, except for branch instruction, where the jump targets are the translated program counters.

```
Input:
        call target1
        call target2
        call target3
        ret
Output:
        call .edge0
        call .edge1
        call .edge2
        ret
.edge0: save_registers_and flags
        clear interrupts
        set target_offset
        set nextpc
        jump to dispatcher
.edge1: ... (similar to .edge0)
.edge2: ... (similar to .edge0)

dispatcher:
    save guest register and flags
    %fs:jtarget = disassemble_and_code_gen(nextpc)
    *(vaddr_t*)target_offset = %fs:jtarget
    restore guest register and flags
    jmp *%fs:jtarget
```

Fig. 3.15 A code block could potentially contain multiple call instructions. The `target_offset` allows the dispatcher to know where to patch the translated code address of the corresponding call target, for direct block chaining.

in this example.

Function calls with indirect targets require special care. Figure 3.16 shows the translation of a code block consisting of an indirect `call` instruction. With call-ret optimization, an indirect call instruction needs to push the code-cache address of the next instruction (labeled .next) on the stack. On jumptable hit, the indirect call instruction at line-8 does this automatically. In the case of a jumptable miss, the dispatcher is called. The dispatcher obtains the code-cache address corresponding to target program counter and directly jumps to it.

To emulate correct behavior, the dispatcher is invoked using a `call` instruction (line-13) followed by a `jmp` instruction. The target of the `jmp` instruction is the return address of the called function. If the procedure is called through the dispatcher, after returning from the procedure, it executes the `jmp` instruction at line-14 and jumps to the code-cache address corresponding to actual return target.

```
Input:
1.  call *target
2.  ret
Output:
3.  save temporary registers and flags
4.  disable interrupts
5.  %fs:jtarget = lookup_hash(target)
6.  if (not found) jmp .edge0
7.  restore temporary registers and flags
8.  call *%fs:jtarget
9.  .next: ret
10. .edge0:
11. set nextpc
12. restore temporary registers and flags
13. call dispatcher
14. jmp .next

dispatcher:
    save guest register and flags
    %fs:jtarget = disassemble_and_code_gen(nextpc)
    *(vaddr_t*)target_offset = %fs:jtarget
    restore guest register and flags
    jmp *%fs:jtarget
```

Fig. 3.16 The translation of an example indirect call instruction. `target_offset` is used for direct branch chaining.

3.6.2 Code Cache Optimization

Figure 3.17 shows an example of a code block with multiple conditional branches. We experimentally found out that these extra edges introduce to the code-cache results in poor instruction cache locality. As discussed earlier, these extra stubs added to the end of the code block execute only once. This is the classic example of cold code sharing the same cache line with hot code. To fix this, we allocate these stubs in a different memory pool called edge cache. We observe a noticeable performance improvement after this optimization, as we discuss in the experiments section.

Recall that we may require a prefix stub, if we are not able to find unused space in the interrupt context to save `jtarget`. We can do similar optimizations to allocate the prefix stub from a different memory pool if required. Our current implementation does not use a prefix stub; instead, we repurpose the unused space in interrupt context to save `jtarget`.

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

3.6.3 Indirect branch optimization

The function call-ret optimization significantly reduces the number of indirect branch instructions, which results in significant performance improvement. The indirect branch instruction is translated to a jumptable lookup to calculate the tx-pc value when the pc is known during runtime (see Figure 3.13). This also involves disabling the interrupts during lookup. These are expensive operations. We experimentally found that, for most of the indirect branch instructions, the target address is usually identical across multiple executions, with high probability. We take advantage of this fact and add a conditional branch instruction in the translation of the indirect branch instruction to compare against one hardcoded address before the lookup as shown in Figure 3.18. `$pc_target`, `tx_pc`, and `.edge0` are updated similar to the direct branch chaining code. Initially, `$pc_target` is set to zero. After the first execution of the indirect branch instruction, the dispatcher is called. The dispatcher then rewrites the `$pc_target`, and `tx_pc` with the `nextpc`, and `tx-nextpc` respectively. The `.edge0` is updated with the address of the indirect branch handler (which does the jumptable lookup and call the dispatcher as in Figure 3.13). `$pc_target` is updated in the last to maintain consistency across the concurrent execution of the current indirect branch instruction. After this optimization, the subsequent executions of the indirect instruction with the same `nextpc` value as `$pc_target` do not involve disabling of interrupts and jumptable lookup.

3.6.4 Per-CPU code-cache vs shared code-cache

In our initial prototype, we used a `per-CPU` code cache. This makes the dispatcher completely concurrent because concurrent calls to dispatcher work on different code-caches. Later we

```
cmp $pc_target, MEM
je tx_pc
jmp .edge0
.edge0:
    save_registers_and_flags
    clear interrupts
    set nextpc
    jump to dispatcher
```

Fig. 3.18 The translation of an example indirect branch instruction "`jmp *MEM`", which checks against one harcoded address, `pc_target`, before looking up the jumptable.

found out that once the code-cache is warmed up, calls to the dispatcher are rare and it is okay to have a shared code-cache across multiple CPUs, to reduce memory pressure. On the other hand, multiple code-caches are beneficial in the scenarios where CPU specific optimizations are required. Figure 3.19 shows an example where a `per-CPU` code-cache results in better generated code than a shared code-cache. In this example, we want to take a decision based on the current CPU-id. With a shared code cache, we first have to load the CPU-id in some register before making a decision, whereas, with a `per-CPU` code cache, we can hard-code the CPU-id in the instruction itself. In this case, the shared code requires an extra register and a load instruction. In our design, a `per-CPU` code-cache adds extra complexity because we allow code cache addresses to be present in the guest stacks. On a call to the guest's scheduler, a thread might end up executing the code corresponding to a different CPU (when it unwinds its stack), if it gets migrated across CPUs. To prevent this inconsistency, on every scheduling decision, we walk the stack and replace the code-cache addresses with the code-cache addresses corresponding to current CPU. We rely on the compiler generated symbol table to interpose on the *scheduler*, and to identify the routines that need to be instrumented.

3.6.5 Jumptable optimization

Our jumptable is a hash table which maps a guest PC value `pc`, to its code-cache translated PC value `tx-pc`. We implement a jumptable as a contiguous array (Figure 3.8). The hash key is computed by masking the lower 12 bits of the PC address. We observed that a significant number of dispatcher calls occur due to hash collisions. To further reduce these dispatcher calls, we handle hash collisions using open addressing. If we find a hit for the computed index, we directly jump to it; otherwise, we perform linear probing up to the next three entries to find a match before calling the dispatcher. If a match is found during linear probing, we swap the current entry with the top, for faster execution of subsequent indirect calls. Recall that this

```
CPU0:
 cmp $0x1, MEM
 je next
 ...
next:
 ...

CPU1:
 cmp $0x2, MEM
 je next
 ...
next:
 ...
```

(a) Per-CPU code cache.

```
 mov %fs:var, reg
 cmp reg, MEM
 je next
 ...
next:
 ...
```

(b) Shared global code cache.

Fig. 3.19 Code translation for code that depends on the current CPU-id, for a per-CPU code-cache and a shared global code cache.

lookup logic is programmed in assembly code. After this optimization, we find that the calls to the dispatcher reduce drastically; this is significant because dispatcher calls constitute a major source of DBT overheads.

3.7 Translator switchon and switchoff

Our DBT framework is a kernel module, which can dynamically start and stop translation (discussed in Section 4.3). The call-ret optimization adds extra complexity to translator *switchon* and *switchoff*. At the time of switchon, the threads in the scheduler list contain the native addresses. With call-ret optimization, if a thread got scheduled after our module is loaded, it may momentarily run native code after it unwinds its stack. However, when a thread enters through the kernel entry points it always executes the translated code. To fix this issue on translator switchon, we stop all the CPUs and iterates through the scheduler list to replace all the native return addresses with the code-cache addresses. The program counter values are identified by following the stack's frame pointers. A similar problem exists during switchoff. In this case, the thread stacks may contain code-cache addresses, which need to be replaced by native guest addresses. Again, we perform a similar walk and replace the code-cache addresses with their native counterparts.

Finally, we discuss our code-cache replacement policy. As pointed out earlier, we do not allow code-cache replacement in a way that existing DBT frameworks do. If we indeed

need to do code-cache replacement, a translator switchoff followed by a translator switchon results into a complete invalidation of the code-cache and a fresh translation begins for future execution of the kernel code, with an empty code cache. In our experiments with around 10MB code cache, we did not need to do cache replacement even after running the translated kernel several days with regular desktop applications and some kernel intensive benchmarks.

3.8 Implementation and Results

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

3.8.1 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).

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].

3.8.2 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 additional 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 [28] (with identical configurations).

All our benchmarks are kernel-intensive; the performance overhead of our system on user-level compute-intensive benchmarks are 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 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 the performance for two variants of our translator: `default` (all optimizations except indirect branch optimization 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 (without and with collision), and the number of dispatcher entries. The corresponding results are labeled `prof-default` (all optimizations except indirect branch optimization 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.

3.8.3 Performance

We first discuss the performance overhead on a single core. Figures 3.20, 3.21, and 3.26 plot our performance results. All these workloads intensely exercise the interrupt and exception subsystem of the kernel. The "fast" kernel operations in Figure 3.20 exhibit less than 20% overhead, except `write` (35% overhead) and `read` (25% overhead). We found 11% improvement in `Protection(Prot)`. Figure 3.21 plots the performance of fork operations in `lmbench`. Here, we observe 1-1.5% performance improvement with DBT. Similarly, Figure 3.26 plots the performance on communication-related microbenchmarks. DBT overhead is higher for `tcp` (69%) and `sock` (22%); for others, overhead is less than 15%.

We experimentally found that the higher overheads of `tcp`, `write`, `read`, and `sock` are

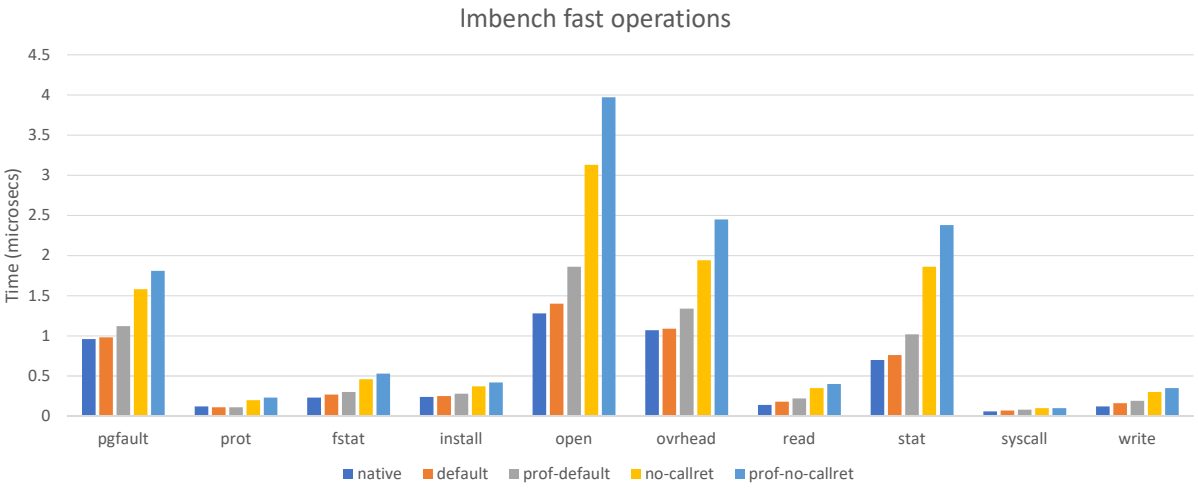


Fig. 3.20 `lmbench` fast operations

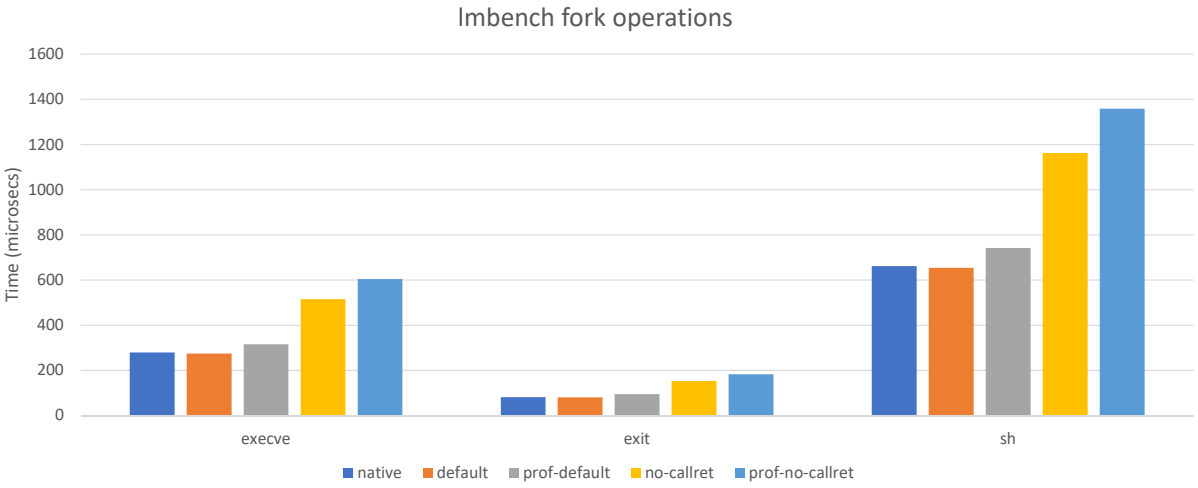


Fig. 3.21 `lmbench` fork operations

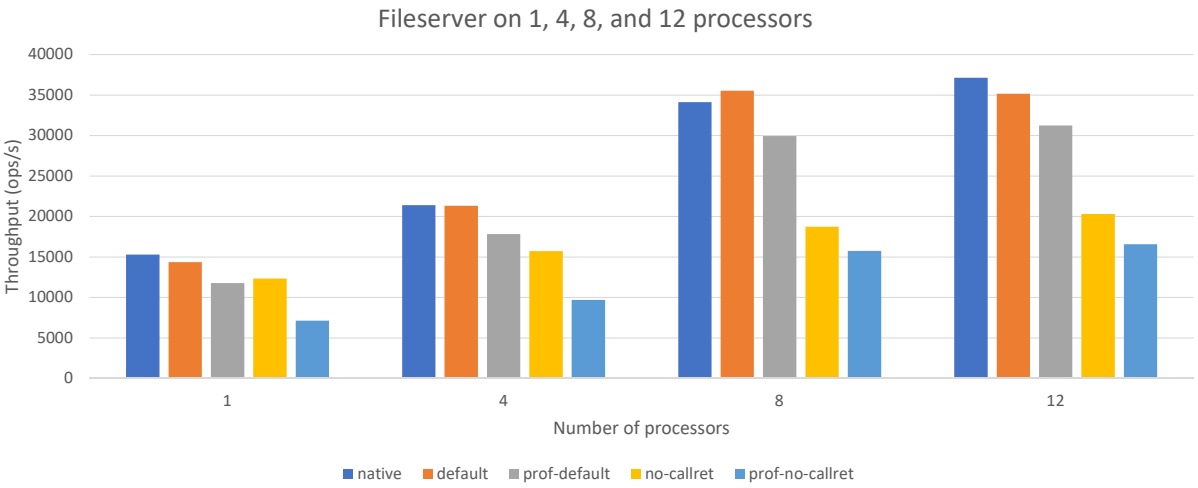


Fig. 3.22 `fileserver` on 1, 4, 8, and 12 processors

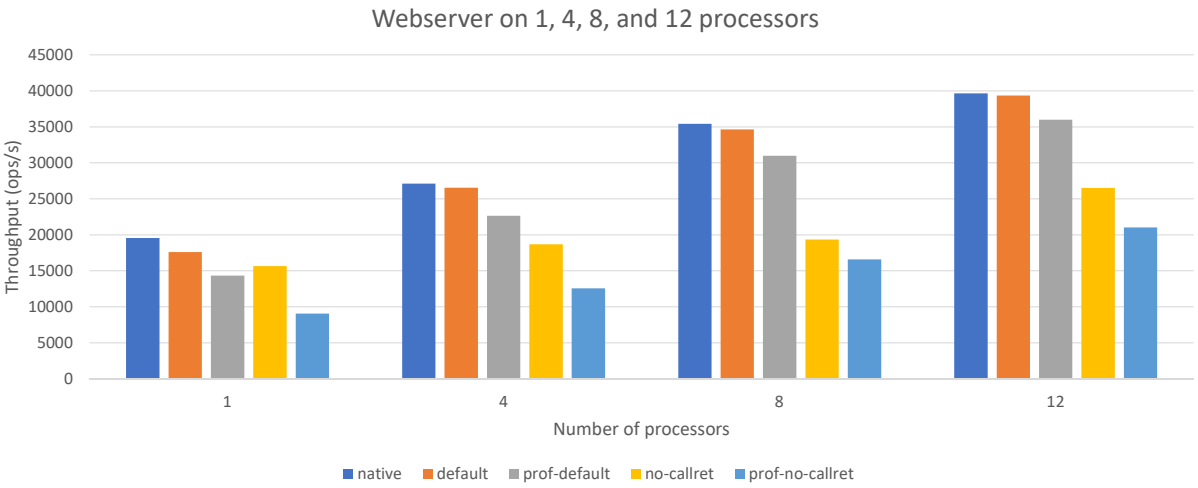


Fig. 3.23 `webserver` on 1, 4, 8, and 12 processors

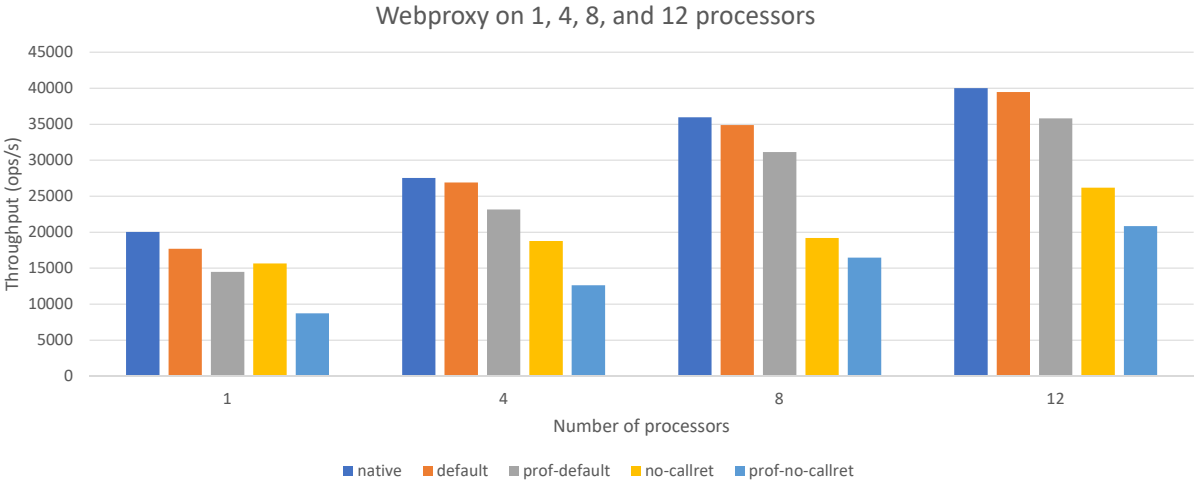


Fig. 3.24 `webproxy` on 1, 4, 8, and 12 processors

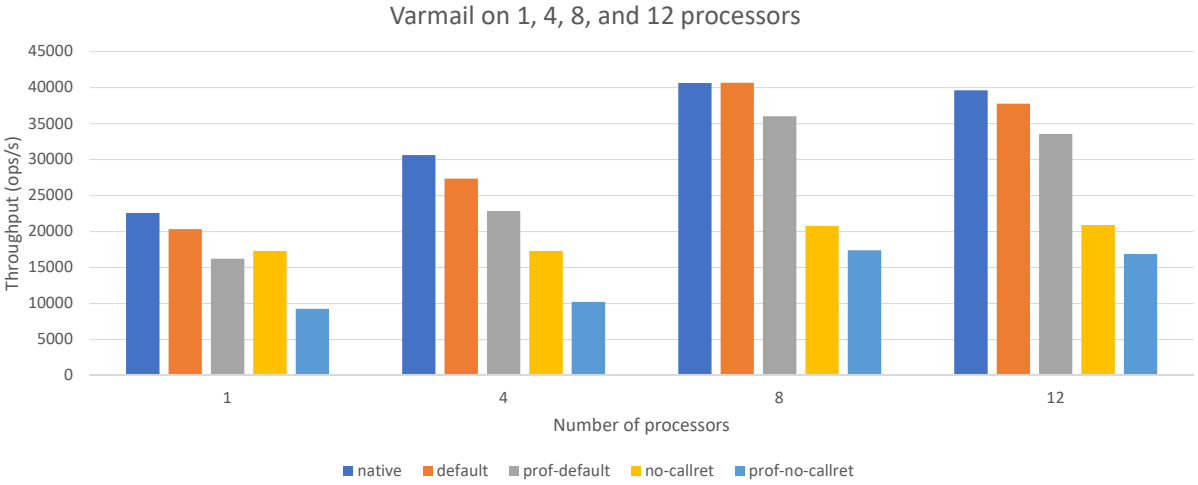


Fig. 3.25 `varmail` on 1, 4, 8, and 12 processors

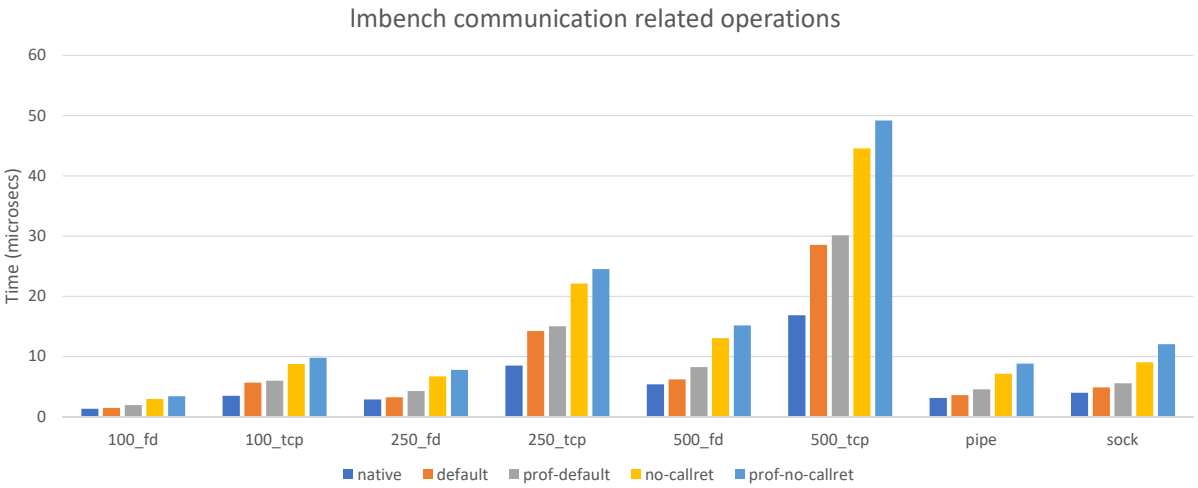


Fig. 3.26 `lmbench` communication related operations

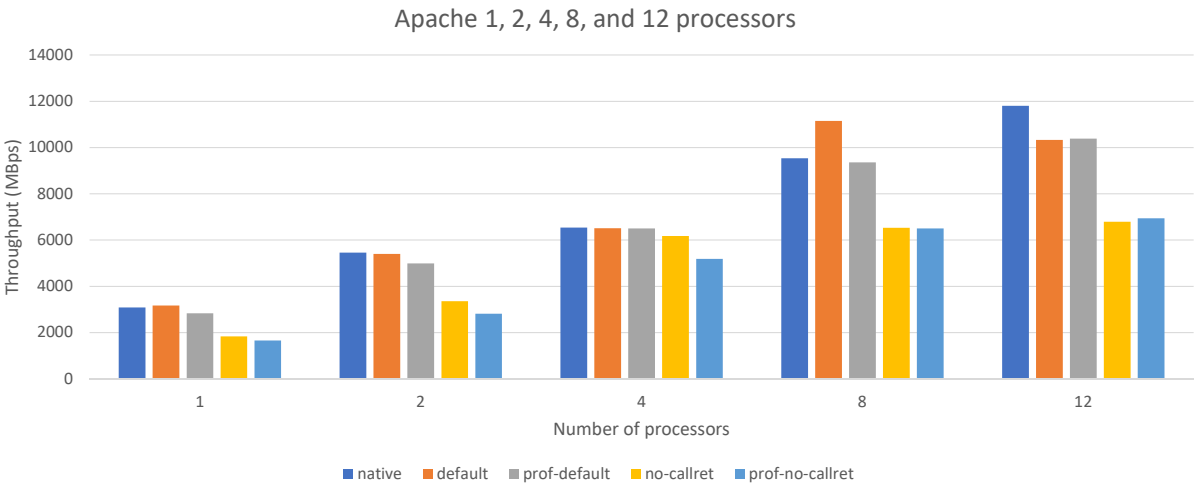


Fig. 3.27 Apache on 1, 2, 4, 8, and 12 processors

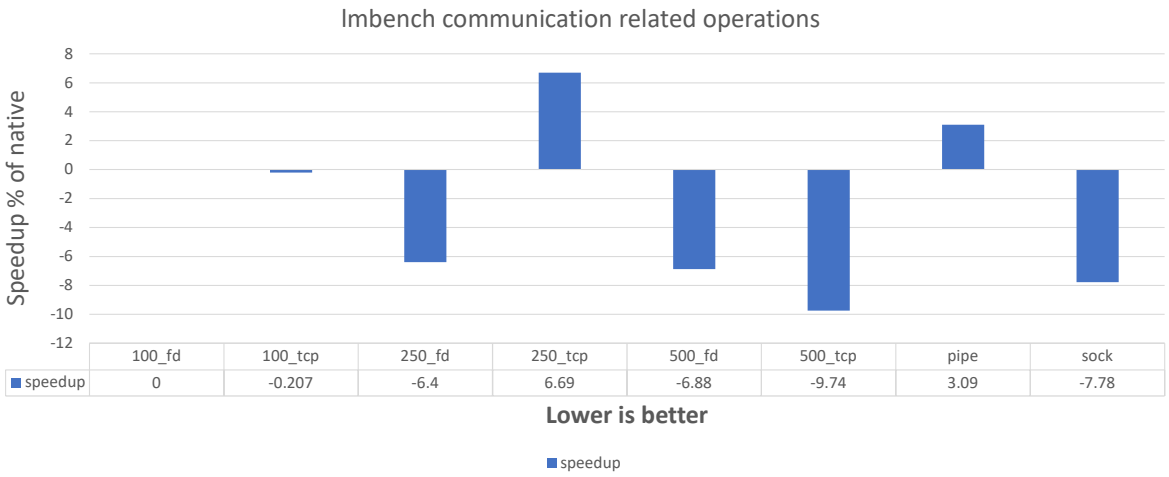


Fig. 3.28 `lmbench` fast operations with indirect branch optimization

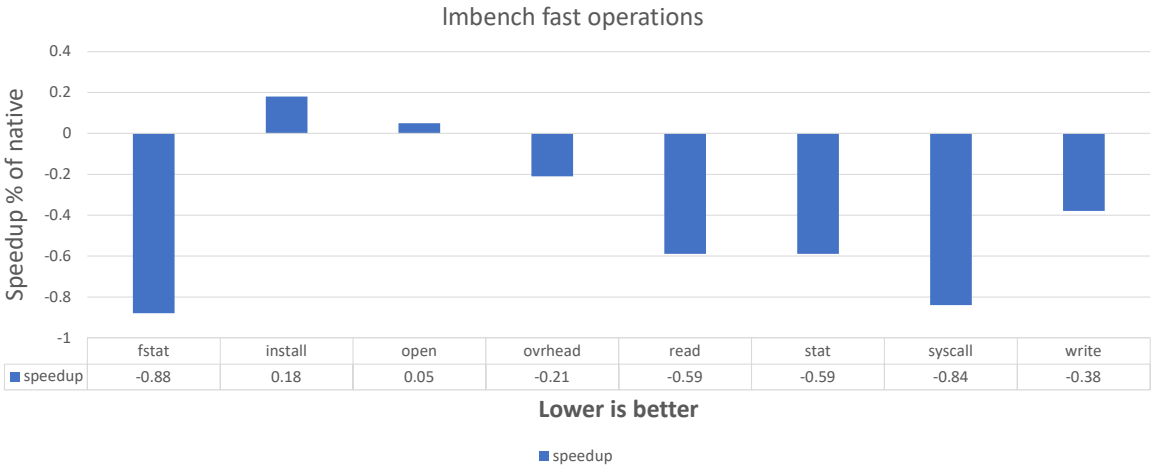


Fig. 3.29 `lmbench` communication related operations with indirect branch optimization

	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

	Without Call Optimization					With Call Optimization				
	Total (x1B)	Indirect (x1M)	Fastpath (x10K)	Slowpath (x1K)	Dispatcher Entries	Total (x1B)	Indirect (x1M)	Fastpath (x10K)	Slowpath (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 without collision (Fastpath) jumptable hits, number of with collision (Slowpath) 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.

due to higher percentage of indirect instructions. For these benchmarks, we found the indirect branch optimization very helpful. With indirect branch optimization, the "fast" kernel operations in Figure 3.29 exhibit less than 1% performance overheads for all the benchmarks. Similarly, with indirect branch optimization, the communication-related microbenchmarks show improvement in some cases (see Figure 3.28). For example, `500_tcp`, `250_fd`, `500_fd`, and `sock` report up to 10% improvement over native. For other benchmarks, the overheads are always less than 7%.

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.

3.8.4 Scalability

To further study the scalability and performance of our translator, we plot the performance of different programs with increasing number of processors. Figures 3.22, 3.23, 3.24, 3.25 plot the 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 eight processors. For other `filebench` workloads, DBT overhead is between 0-10%.

Figure 3.27 shows the throughput of `apache` webserver when used with `apachebench` client over the network. DBT overheads are always less than 12%. We observe performance improvement of 17% (for eight processors) and 2.5% (for one 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 the 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 (with and without collision) 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% without collision, 10.48% with collision). With call-ret optimization, the jumptable hit rates were always above 99.99% (84.94% without collision, 15.05% with collision). In all our experiments, the number of dispatcher entries was roughly equal to the number of jumptable misses.

3.9 Discussion

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

- We avoid back-and-forth translation of interrupted/excepting program counters 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 program counter value (pushed on the 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 program counters 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. Because of the call/ret optimisation, it is possible for an attacker to execute native code by simply overwriting the return address on the stack and thus our scheme is not appropriate for security applications in its most optimized form. However, using our DBT platform, it is possible to implement existing techniques for enforcing control flow integrity(CFI). 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, it is possible to write static or dynamic analysis tools to determine if an optimization is legal. Verification for the call-ret 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).

Chapter 4

Deterministic Replay

The goal of deterministic replay is to recreate the program execution using a trace of non-deterministic events. Deterministic replay has applications in debugging [3, 40, 45, 50, 51, 58], fault tolerance [7, 14–16], intrusion detection [26], remote attestation [36], computer forensics [26], dynamic analysis [20, 48, 52], profiling [4], testing and verification [45, 52], trace generation [12, 52, 63] and more. We study the problem of efficiently recording and replaying a multiprocessor operating system with all its applications. The biggest challenge in deterministic replay is to record the order of shared memory reads and writes. In this chapter, we discuss our method to record shared memory non-determinism using dynamic binary translation efficiently.

4.1 Background

We motivate our approach with an example of a function `foo()` (Figure 4.1). `foo()` dereferences the `ptr` argument. If multiple threads call `foo()` with the same `ptr`, the order of accesses by these threads to this shared variable needs to be recorded for deterministic replay.

Several hardware and software based approaches have been proposed to record this non-determinism due to concurrent accesses of the shared variable. In this thesis, we only focus on full system record/replay. The most relevant work which supports full system record/replay

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

Fig. 4.1 A sample function accessing a shared variable.

is SMP-Revirt [27]. SMP-Revirt implements the concurrent reads exclusive write (CREW) protocol to restrict the behavior of a system such that it can be logged efficiently. A CREW protocol allows concurrent reads to a shared variable by multiple CPUs, but restricts writes to a single CPU. SMP-Revirt uses hardware page protection mechanism to implement CREW for an unmodified virtual machine. Their technique was introduced during the early days of hardware virtualization when the memory management unit (MMU) used the shadow page table for memory virtualization. SMP-Revirt changed the MMU implementation in the hypervisor to create a unique shadow page table for each CPU. To implement CREW, SMP-Revirt always maps a physical page into multiple shadow page tables in write-protected mode; however, a page with write access is mapped into a single CPU's shadow page table. Every page is associated with the set of owners (CPUs). A page owned by multiple CPUs is in read-shared mode – in the case of a single owner, the page is in write-exclusive mode. If a CPU tries to access a page without adequate ownership, it results in a trap to the hypervisor. The hypervisor then relinquishes the desired privilege, by taking away privilege from other CPUs. This privilege transfer must be recorded to reproduce the same transfer during replay.

SMP-Revirt works very well when there is sparse sharing, but incurs huge overheads for workloads exhibiting dense sharing patterns among CPUs. Further, the overheads increase rapidly with the number of cores. The fact that ownership is tracked at page granularity, causes spurious ownership transfers if multiple CPUs access a page on different indexes. This is called the *false sharing* problem where there is no actual sharing, but yet sharing overheads occur due to the granularity of sharing. In this page-granular CREW scheme, the ownership conflicts are very expensive because they also involve an expensive trap into the hypervisor. A hypervisor trap involves a world switch [18] followed by the execution of thousands of instructions in the hypervisor that incur significant overheads.

In general, software-based CREW can be implemented with an additional reader/writer lock for every shared memory access (see Figure 4.2). Here, the `acquire_read` and `release_read` functions protect a read access, while `acquire_write` and `release_write` protect a write access. A lock is associated with every shared memory address. Multiple readers are allowed to enter the *read-critical-section*, but a single writer is permitted to enter the *write-critical-section*. We can ensure the deterministic order of shared memory accesses by recording the order of lock acquisitions. As an optimization, the lock can also be implemented using an ownership protocol. Multiple readers can own the lock simultaneously; however, only one writer is allowed to own the lock. In this case, we only need to record the lock acquisition if the readers/writers do not already own the lock. If a CPU does not own a lock, it needs to record a deterministic state of all the owners before acquiring the lock. This is necessary to reproduce the same deterministic state during the replay.

```
void foo(int *ptr)
{
  acquire_read(&ptr->lock);
  temp = *ptr;
  release_read(&ptr->lock);
  temp++;
  acquire_write(&ptr->lock);
  *ptr = temp;
  release_write(&ptr->lock);
}
```

Fig. 4.2 A software-only implementation of CREW.

Apart from recording shared memory accesses, a typical record/replay framework requires logging of other events that can not be deduced deterministically during replay. These events are interrupts and their timings, I/O port reads, special instructions (`rdtsc`, `rdscp`), device direct memory accesses (DMA), etc. The hypervisor emulates I/O port reads and DMA in software that makes it easier to record them. A race between a concurrent emulated DMA and a concurrent thread can be resolved using the CREW protocol. A hypervisor can configure a virtual machine to generate a trap on special instructions for recording their values. For an asynchronous interrupt, we need to record the timing in addition to the interrupt vector number, to inject them at the same time during the replay. For example, a hypervisor can use fast hardware performance counters to count the number of retired instructions to measure the timing of asynchronous events.

4.2 Our technique

There are two main problems with the page-grained CREW approach: slow ownership transfer, and false sharing. To mitigate these problems, we use DBT to implement CREW at byte granularity, thus eliminating false sharing. We also handle ownership transfers within the guest itself, thus avoiding the expensive hypervisor traps and world switches. To do this, we use shadow memory (Section 4.2.1) to associate a reader-writer lock with each accessed byte in main memory.

A large fraction of memory accesses are CPU-private, e.g., stack accesses. Tracking ownership of CPU-private memory locations would incur unnecessary overheads. We rely on a *training phase* to identify the instructions that may access shared memory. An instruction that accesses a memory location that has previously been accessed by another thread is deemed to be accessing shared memory (full discussion in Section 4.2.2).

While many user-applications do not exhibit a high degree of sharing, kernel code usually involves dense sharing patterns. Thus, we use the software implementation of CREW for kernel-mode execution, whereas the page-protection based CREW (ala SMP-revirt) is used for user-mode execution. In our SMP-Revirt implementation, we observe a lot of ownership conflicts due to true and false sharing inside the kernel. We believe that these overheads are hard to avoid with a page-granular ownership tracking scheme, and hence it is better to use an instrumentation-based byte-granular CREW inside the kernel for faster handling of shared-memory ownership conflicts. Although instrumentation adds overheads to the common case, the faster execution of ownership conflicts outperforms the page based CREW in most cases.

4.2.1 Shadow memory

Shadow memory is a technique to track extra information about the main memory used by an application, during runtime. A shadow byte relates to some byte in main memory. Usually, the shadow memory is invisible to the original application and additional instrumentation is added to the normal execution, which manipulates shadow memory for gathering information about the main memory, e.g., for monitoring or profiling purposes. We implement fast shadow memory for the 32-bit Linux kernel using shadow memory. The 32-bit Linux kernel reserves the top one GB (0xc0000000-0xffffffff) virtual address space for its exclusive use, in each process's page table. We reserve the top 512 MB of this kernel space for implementing shadow memory. Our DBT engine uses one byte corresponding to every byte in the main memory to implement the corresponding reader/writer lock.

At initialization time, our shadow-memory module walks through the kernel page table to map a shadow page corresponding to every original page in the kernel address space. Further updates to the page table are intercepted, and the corresponding mappings are appropriately created for shadow memory. We change the Linux kernel source code to implement this feature.

4.2.2 Global variable detection

Instrumenting all memory accesses is avoidable, as a large fraction of the memory accesses are thread-private, e.g., stack accesses. Our software implementation of CREW instruments only those instructions that *may* access a shared address. Our DBT-based instrumentation module works on unmodified binaries, and it is generally hard to detect a shared memory access by statically looking at the instructions. Thus, to identify global variables, we rely on a "training phase" before the DBT module is ready for recording the kernel. During the training period, we instrument every memory access instruction to use shadow memory for detecting

shared accesses. Every byte of main memory corresponds to one byte of shadow memory. Individual bits in the shadow byte are associated with different CPUs. On a memory access, the instrumentation code sets the bit corresponding to the executing CPU in the shadow byte. If a memory location is accessed by multiple CPUs and has been written at least once, it is marked as *global* (or shared). Further, when the instrumentation code encounters a global memory location, it marks the corresponding program counter (PC value) as a *shared memory accessor*, i.e., an instruction that may access shared memory. To prevent misdetection of a global variables due to memory reuse by the allocators, `malloc` and `free` are modified to reset the shadow bytes corresponding to the object being allocated/freed.

Our algorithm to detect shared (global) locations and instructions that may access shared locations, is best-effort, and could result in both over-approximation and under-approximation.

The algorithm could result in over-approximation (i.e., some instructions may incorrectly be identified as shared memory accessors) in the following cases:

- The algorithm may fail to detect a custom memory pool/allocator inside the kernel. A re-allocation from the pool may incorrectly mark a variable shared.

- The algorithm does not capture ordering relationships between accesses to multiple memory locations. For example, the *happens-before* relationship has been used widely in the literature [30, 39] to capture relationships between synchronization primitives and shared variables. In these cases, if a shared variable is always appropriately protected by a synchronization primitive (e.g., lock), then it is not necessary to track the ownership information for the shared variable — it suffices to track ownership for the synchronization variable in this case. Because our algorithm does not capture this ordering information, it would track ownership for *both* the synchronization primitive and the shared-variable in our example. Our algorithm to detect shared-memory locations and shared-memory accessing instructions is similar to the lock-set algorithm used in Eraser [55], in the sense that like us, Eraser is also order-agnostic.

Over-approximation may lead to over-instrumentation resulting in higher overheads for both recording and replaying; however over-approximation does not compromise the soundness (correctness) of the scheme.

Conversely, our technique could also result in under-approximation, if some behavior of the code is left unexercised during the training phase. In these cases, we may incorrectly identify some instructions as private-memory accessors, even though they may access shared memory in the production run. Under-approximation is a correctness (soundness) issue, and may lead to replay failure. A sound solution would perhaps need to rely on a combination

of static and dynamic approaches to identify instructions that may access shared-memory regions. We leave this for future work.

Even if our algorithm fails to detect all shared-memory accessing instructions (due to under-approximation limitations outlined before), it will only manifest as a replay failure. We argue that such cases should be relatively rare (assuming that the training phase has high code coverage), and can be handled using a guided search during the replay phase as proposed by multiple previous works [3, 51]. The guided search should be able to reproduce the data race after multiple replays; the number of replays required can be bounded using our best-effort scheme to identify shared memory accessing instructions.

4.2.3 Byte-granularity CREW

We use DBT to implement the CREW protocol at byte granularity. The instrumented code maintains the ownership information in shadow memory. We reserve one byte of shadow memory corresponding to every byte in the main memory. Our system reserves one bit for every CPU in the shadow byte, and thus our current implementation can only record up to eight CPUs. This limitation can be relaxed by allocating more space to shadow memory. To simplify instrumentation, the shadow byte is kept at a constant offset from the main memory. We instrument every instruction that may access shared memory (as detected through the training phase Section 4.2.2) to perform a (fast) ownership check. For read accesses, the ownership check involves checking if the current core is one of the owners; for write accesses, the instrumentation code checks if the current CPU is the exclusive owner the main memory byte. If the fast ownership check fails, the `acquire_slowpath` routine (Figure 4.4) tries to atomically acquire the ownership from the current owners. If the ownership acquisition succeeds, the deterministic state of all the conflicting CPUs is logged in the replay-log, so that this ownership transfer can be reproduced identically during the replay. As we have discussed before (Section 4.1), these ownership checks and transfers can also be thought-of as reader/writer locks, and we will discuss them as such.

4.2.4 Lock implementation

A reader/writer lock acquire/release is placed around every instruction that may access a shared variable (detected during the training phase Section 4.2.2). A `read_acquire()` routine is inserted before every read access, and `write_acquire()` is inserted before every write access. A `release()` routine is appended after the shared memory access (read/write) finishes. Every memory location is associated with a reader-writer lock. The `rwlock_t` structure is essentially a bitmap of owners that are allowed to access the corresponding memory loca-

```
cpu_t
{
   long sequence_number; // count the number of shared loads and stores
   byte_t id;            // unique id for each cpu
   byte_t mask;          // (1 << id)
   long  brlock;         // big-reader lock
}

rwlock_t
{
  byte_t owners;         // A bitmap of cpu-ids currently holding this lock
}

read_acquire(struct rwlock l):
 1 cur_cpu.brlock = true;
 2 if ((l.owners & cur_cpu.mask) == 0) // the current cpu is one of the owners
 3 {
 4     acquire_slowpath(l, cur_cpu, true);
 5 }
 6 cur_cpu.sequence_number++;

write_acquire(struct rwlock l):
 1 cur_cpu.brlock = true;
 2 if (l.owners != cur_cpu.mask) // the current cpu is the only owner
 3 {
 4     acquire_slowpath(l, cur_cpu, false);
 5 }
 6 cur_cpu.sequence_number++;

release(struct lock l):
 1 cur_cpu.brlock = false;
```

Fig. 4.3 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. If the location 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. If the check fails, the `acquire_slowpath` routine in Figure 4.4 is called to update the ownership information.

```
acquire_slowpath(struct rwlock l, cpu_t cur_cpu, bool read):
 1  cur_cpu.brlock = false;
 2
 3  %eax := 0;
 4  /* lock the current memory index */
 5  while (lock xchgb(%al, &l.owners) == 0)
 6  {
 7     cup_relax();
 8  }
 9
 10 /* current set of owners is stored in %eax */
 11 owners := %eax;
 12 for each cpu c in bitmask(owners)
 13 {
 14    while (c.brlock);
 15    add c.sequence_number, c.id and cur_cpu.sequence_number to the record
    log
 16 }
 17
 18 /* reacquire the big reader lock */
 19 cur_cpu.brlock = true;
 20
 21 if (read)
 22     l.owners = (owners | cur_cpu.cpu_mask);
 23 else
 24     l.owners = cur_cpu.cpu_mask;
```

Fig. 4.4 Pseudo-code of our instrumented reader-writer slowpath code. The `acquire_slowpath()` function waits for the current owner CPUs to leave the critical section (i.e., wait for their `brlock` 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 CPU repeatedly. The sequence numbers for current CPU and owner CPUs are logged with every ownership update event, to record the order of events.

tion. At the high-level, `read_acquire()` and `write_acquire()` routines simply check for the shared/exclusive ownership of the current CPU. If this check fails, then the CPU atomically takes away the ownership of all the CPUs by resetting the owner bitmask (ownership invalidation). This is done to ensure that no other CPU can concurrently access the memory location after the ownership invalidation.

This scheme would work fine if the ownership checks and the memory access could be performed in one atomic instruction. However, the critical sections of the `read_acquire()` and `write_acquire()` functions consist of multiple instructions, and to ensure that no other CPUs are concurrently in their critical section, we additionally use a per-CPU big-reader lock (`brlock`) [21]. We carefully optimize the implementation for our scheme for the case where ownership transfers are rare.

We now discuss the different race conditions that are possible, and the correctness of our locking scheme. We do so by enumerating the different scenarios of concurrent execution (e.g., when `read_acquire()` executes concurrently with another `read_acquire()`), and discussing why the lock implementation is sound in all these cases.

1. **Concurrent execution of [read|write]_acquire with another [read|write]_acquire:** The `read_acquire` and `write_acquire` functions implement the fast-path checks, and simply check for ownership of the executing CPU. The ownership change can only happen in the acquire_slowpath. Because the ownership checks are read-only, the concurrent execution of `[read|write]_acquire` and another `[read|write]_acquire` is always safe.

2. **Concurrent execution of `[read|write]_acquire` with `acquire_slowpath`:** The acquire_slowpath function first atomically invalidates the ownership. This ensures that the subsequent ownership check at line-2 in `[read|write]_acquire` routines always goes to the slowpath. Furthermore, a spinlock at line-5 in `acquire_slowpath` ensures that only one CPU can proceed after this point. Before ownership transfer, the current CPU needs to reach a deterministic point that is reproducible during replay. This deterministic point ensures that all the CPUs have already safely accessed the current memory location and have seen the new state of the lock (i.e., nobody owns the lock). The `xchg` instruction at line-5 ensures that all the CPUs see the new state of the lock atomically. However, another problem exists: other CPUs may still be in their critical section. To ensure that all the other CPUs have already accessed the current memory location, we use big-reader locks.

   A big-reader lock is efficient in our case of read-mostly access patterns: the lock is acquired in read-mode on the fast path, and acquired in the write-mode on the slow

path. Because slow-path is much rarer than fast-path, big-reader locks are an appropriate choice in this scenario.

The big-reader lock is acquired at line-1 in the `[read|write]_acquire` routine and released only after the successful access of shared memory (in release() routine). So, in the `acquire_slowpath` routine, acquiring the per-CPU big-reader lock of owner CPUs ensures the completion of outstanding accesses to current memory byte by all the CPUs. It is also possible that some of the CPUs are currently in the `acquire_slowpath` and currently waiting in the spinlock at line-5. To prevent a deadlock during acquisition of per-CPU locks of these CPUs, we release the per-CPU lock at line-1 in the `acquire_slowpath()` function.

Once the current CPU is convinced that all the other CPUs are at the deterministic point after acquiring the big-reader lock, it simply logs the ID(s) of the owner CPU(s), deterministic counters of the owner CPU(s), and its own deterministic counter in its record log to reacquire the ownership during replay. The deterministic counter (sequence number) is a per-CPU counter that counts the number of retired shared memory accesses. The current CPU acquires its own per-CPU big reader lock at line-19 before acquiring the ownership. At line-22,24 the ownership information is stored in the reader-writer locks to allow other CPUs to acquire the spinlock at line-5. But this does not prevent the current CPU to access the shared location because the other CPUs must wait for this CPU to reach its deterministic point (i.e. release of the big-reader lock).

3. **Concurrent execution of `acquire_slowpath` with another `acquire_slowpath`:** The concurrent execution of two `acquire_slowpath` routines gets serialized through the spinlock at line-5. The spinlock protects the acquisition of the big-reader lock at line-19; the spinlock is released after the successful acquisition of the big-reader lock at lines-22,24.

4.2.5 Lock implementation for relaxed memory models

While the above implementation works on a machine with sequentially consistent memory model, there are more issues for relaxed memory models. On the x86 architecture with the TSO memory consistency model [34], a load can be reordered with earlier stores, if they access different memory locations. Consider a case when a load at line-2 (Figure 4.3) in [read|write]_acquire gets reordered with the acquisition of big-reader lock at line-1 (Figure 4.4). Notice that, in this case, the slowpath may incorrectly assume that no other thread is currently in their critical section and would log a wrong sequence_number. In our original

implementation, we overlooked this issue, and our evaluation is based on the above implementation. Since the probability of such event is rare, a search during replay can fix these inconsistencies [3, 51]. Next we propose an alternate way to get rid of this limitation. One way to ensure correctness is to add a fence after acquiring the big-reader lock, but that would be unacceptably inefficient for the fast-path code. We discuss a faster approach:

At line-12 (Figure 4.4) in the slowpath function, the current CPU iterates the current set of owners to finish their critical sections. As we pointed out earlier that after resetting the current owners at line-5 in slowpath routine (ownership invalidation), a subsequent access to the corresponding memory location always results in a call to the slowpath. For the deterministic point, we just need to make sure that the CPU(s), which have already successfully checked the ownership information must have accessed this memory location and incremented the sequence_number. Because, on x86 TSO, a load or store can not be reordered with subsequent stores, just ensuring that the sequence number update happened at least once on all the owner CPU(s) after the ownership invalidation, would be enough to ensure that no other CPU(s) is accessing the conflicting memory location concurrently. On x86, for efficiency, the updates are stored in a per-CPU store buffer and only flushed out to the shared cache on certain events (e.g., not enough space in store buffer). One way to check whether all the owner CPU(s) have successfully accessed a shared location is: take the snapshot of the sequence number of all the owner CPU(s) after ownership invalidation and wait for them to change at least once. A change in the sequence number implies that the remote CPU(s) have flushed their cache line (which contains the sequence number) at least once after the ownership invalidation. However, this still does not necessarily ensure that the remote CPU(s) have successfully exited their critical section: a flush to the write buffer may happen due to other writes or some other event, and in this case, the updated value we have, may be an older sequence number prior to the invalidation of ownership. However, after this point, another change in sequence number should imply that all the owner CPU(s) have updated the sequence number at least once after the ownership invalidation, because the cache line which contains the sequence number was flushed twice after the ownership invalidation. (If a memory location has been updated twice, we assume that there must have been at least one store-buffer flush between the two updates, because otherwise both updates should have been absorbed by the store buffer). One problem with this approach is: we only increment the sequence_number at a shared memory access. If the other CPU(s) are not accessing the shared memory for a longer duration, then the slowpath may unnecessarily consume CPU cycles without doing any useful work. To ensure bounded waiting times in the slowpath, we increment the sequence_number at the start of every basic block that does not access any shared memory. Even so, certain instructions may block still causing the waiting time to grow unbounded (e.g., I/O emulation code, the slowpath itself, or

```
cpu_t
{
   long sequence_number;           // count the number of shared load and
       stores
   byte_t id;                      // unique id for each cpu
   byte_t mask;                    // (1 << id)
   long old_sequence_number[MAX_CPUS]; // temporary space used in slow path
   byte_t state;                   // state may be BLOCK or ACTIVE
}

rwlock_t
{
  byte_t owners;      // A bitmap of cpu-ids currently holding this lock
}

read_acquire(struct rwlock l):
  if ((l.owners & cur_cpu.mask) == 0) // the current cpu is one of the owners
  {
      acquire_slowpath(l, cur_cpu, true);
  }

write_acquire(struct rwlock l):
  if (l.owners != cur_cpu.mask) // the current cpu is the only owner
  {
      acquire_slowpath(l, cur_cpu, false);
  }

release(struct lock l):
  cur_cpu.sequence_number++;
```

Fig. 4.5 Pseudo-code of our instrumented reader-writer lock routines for x86 TSO memory model.

user-mode execution may all potentially block). In all of these events, a separate flag is set to notify that the CPU is in a block state. The slowpath only waits for another CPU's sequence number to change, if the other CPU is not in block state (see Figure 4.6). In this new scheme, the fast-path code (Figure 4.5) does not require the big-reader lock.

4.2.6 Replay

During the record phase, we log three entries corresponding to every owner CPU: current CPU's sequence_number, owner CPU's ID, and owner CPU's sequence_number. If there are multiple owners, ID and sequence_number is logged for all of them. During replay, every instruction that can access a shared memory is prepended and appended with `replay_begin`

```
acquire_slowpath(struct rwlock l, cpu_t cur_cpu, bool read):
 1   cur_cpu.state = BLOCK;
 2   id := cur_cpu.id;
 3   %eax := 0;
 4   /* lock the current memory index */
 5   while (lock xchgb(%al, &l.owners) == 0)
 6   {
 7      cup_relax();
 8   }
 9   /* current set of owners is stored in %eax */
 10  owners := %eax;
 11  for each cpu c in bitmask(owners)
 12  {
 13     c.old_sequence_number[id] = c.sequence_number
 14     c.iter[id] = 0;
 15  }
 16  do
 17  {
 18     need_iteration = false;
 19     for each cpu c in bitmask(owners)
 20     {
 21        If (c.iter[id] == 2 || c.state == BLOCK)
 22        {
 23           c.iter[id] = 2;
 24           continue;
 25        }
 26        need_iteration = true;
 27        If (c.old_sequnce_number[id] != c.sequence_number)
 28        {
 29           c.old_sequence_number[id] = c.sequence_number;
 30           c.iter[id]++;
 31        }
 32     }
 33  } while (need_iteration);

 34  for each cpu c in bitmask(owners)
 35  {
 36     add c.sequence_number, c.id and cur_cpu.sequence_number to the record
     log
 37  }
 38  cur_cpu.state = ACTIVE;
 39  memory_barrier();
 40  if (read)
 41    l.owners = (owners | cur_cpu.cpu_mask);
 42  else
 43    l.owners = cur_cpu.cpu_mask;
```

Fig. 4.6 Pseudo-code of our instrumented reader-writer slowpath for x86 TSO memory model.

```
struct log_t
{
  long sequence_number;
}

replay_begin(cpu_t cur_cpu, log_t cur_log):
  while (cur_log.sequence_number == cur_cpu.sequence_number)
  {
    owner_cpu_id := read_from_log();
    owner_sequence_number := read_from_log();
    while (get_cpu(owner_cpu_id).sequence_number < owner_sequence_number);
    cur_log.sequence_number = read_from_log();
  }

replay_end(cpu_t cur_cpu):
  cur_cpu.sequence_number++;
```

Fig. 4.7 Pseudo-code of our instrumented replay routine. The `replay_head` routine waits for all the CPUs to reach their deterministic point. The `replay_tail` function simply increments the expired shared memory accesses count.

and `replay_end` routines respectively (see Figure 4.7). `replay_begin` checks if the current CPU's sequence number is equal to the one in record log. If yes, then the current CPU waits for other CPUs in the record log to reach their deterministic point. This ensures the same order of shared memory access, as it was during record. After the shared memory access, the `replay_tail` routine increments the number of retired shared memory access (sequence_number), as is done during record.

4.2.7 Asynchronous events in kernel execution

One issue with using DBT for record/replay is: the number of branch instructions executed during the record and replay is different. This is because instrumentation code in record/replay is very different, and the dispatcher follows different code-paths for generating different stubs for record and replay. Secondly, the program counter values are different in record/replay that causes uneven dispatcher exits during record and replay, because the *jumptable* depends on the program counter values. Due to these reasons, traditional approaches to count the number of retired instructions (or retired branch instructions) for interrupt delivery (or the delivery of any other asynchronous event) are no longer applicable.

To deal with this issue, we restrict asynchronous events to occur only at certain *determin-*

istic execution points during the kernel execution. Deterministic execution points reflect the execution points that will occur during both record and replay in the exact same sequence and count. In our implementation, we use the I/O instructions, other special instructions like (e.g., `rdtsc`, `rdpmc`) and the x86 `pause` instruction [33] as deterministic points: these events can be identified deterministically during replay because our DBT module does not use any of these instructions, and hence their sequence and count during record and replay will be identical. Further, the time gap between two deterministic points needs to be bounded, to ensure bounded waiting for the delivery of asynchronous events (e.g., interrupts). For example, the identification of the `pause` instruction (which is executed each time the CPU enters the idle state) as a deterministic point allows bounded waiting times for asynchronous events during CPU blocking.

This restriction of delivering asynchronous events (e.g., interrupts) at only deterministic execution points makes the DBT dispatcher execution completely transparent with respect to the translated execution. In other words, even if the dispatcher state may vary during the recorded and replayed executions, the translated code behavior remains identical in both executions.

The challenge in this scheme is to be able to identify deterministic execution points such that they are both *deterministic* (i.e., their sequence and count is preserved across record and replay) and ensure bounded waiting times (i.e., the time gap between two deterministic execution points must be bounded). The latter requirement is harder to meet: if the time gap between two deterministic execution points can grow unbounded, this synchronous delivery of interrupts (or other asynchronous events) could lead to a deadlock. For example, consider a case where `CPU-1` sends an *inter-processor* interrupt to `CPU-2`, and busy-waits for `CPU-2` to receive the interrupt. At the same time, `CPU-2` may be busy-waiting for `CPU-1` to release some lock. This situation causes a deadlock if the busy-wait loop in `CPU-2` does not execute a deterministic instruction that allows the hypervisor to inject the interrupt. This is rare, because usually busy-waiting involves the `pause` instruction. In the few cases in the Linux kernel, where busy-waiting was implemented without the `pause` instruction, we manually inserted a `pause` instruction in the busy wait loop (by changing the Linux kernel source code). Most lock primitives in the Linux kernel already use the `pause` instruction, and so we had to manually modify only a few custom busy wait loops.

4.3 Experiments

We implemented our deterministic replay system for the multiprocessor virtual machine running a 32-bit Linux kernel. We use KVM/QEMU as the hosting platform for the guest oper-

ating system. The device emulation code in QEMU is modified to record the value of ioport reads, network packets, etc. The guest virtual machine is configured to trap on special instructions like `rdtsc`, to record their values. Interrupts for virtual machines are also emulated and preceded by a trap to the hypervisor. However, the delivery of interrupt is asynchronous. To record the timing of asynchronous interrupts, we record the number of branch instructions executed in user-mode, program counter value, and rcx; which gives us a unique timestamp. Notice that we only count the branch instructions executed in user-mode (and not in kernel-mode) because the interrupt delivery timing in the kernel is counted using deterministic execution points (Section 4.2.7).

We use hardware performance counters to count the number of expired user-mode branch instructions. The imprecise instruction count support on x86 hardware forces us to use the precise branch count. A branch count combined with the instruction pointer (`eip`) and provides the exact instruction, where we need to inject interrupt. However, there are some instructions, which may be partial executed (e.g., `rep movs`) during interrupt injection. The `rep` prefix executes the instruction as many times as specified in the count register `ecx` and decrements `ecx` on each iteration. To handle this case we also include `ecx` in our logged timestamp. We disable branch counting during kernel mode execution, as we only inject interrupts at deterministic execution points in the kernel.

During replay, we use a somewhat imprecise performance monitoring interrupt (PMI) mechanism to inject a logged event at a given time epoch (as also done in [27, 63]). To inject the interrupt at the same timestamp as was logged during record, we need to configure the virtual machine to trap at the logged branch count. We use the overflow counter in the `x86` architecture, which generates a trap when the counter overflows. However, this counter is not precise and may cause a trap within an error window of 128 branches. This means that, if we configure it to take a trap after `x` branches, it may trap anywhere between `x to (x + 128)` branches. To deal with this issue, we configure the counter to trap before 128 branches of the original timestamp (i.e., `x-128`) and run the virtual machine in *single-step* mode until we reach the current timestamp value. For recording byte-grained CREW conflicts, we allocate a large per-CPU buffer inside the guest. When the buffer gets full, we trap to the hypervisor for flushing it to the disk before reusing the buffer.

For comparison, we implemented a page-grained CREW mechanism similar to SMP-Revirt [27] inside KVM. We used the extended page table (EPT) hardware [34] and manipulated the present and read/write bits in the EPT page tables, to implement page-grained read-/write CPU-level ownership of pages. We change the existing implementation to use per-CPU extended page tables. 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).

Name	Description
`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 use BTKernel[35] to implement byte-grained CREW protocol inside the guest operating system. BTKernel achieves near native performance for kernel intensive benchmarks for identical translation. We ran our experiments on a four-processor Intel Core i7 with 3GB RAM. Our guest virtual machine is configured with 1 GB of RAM, out of which 512 MB is reserved for shadow memory. We first instrument all the memory instructions to use shadow memory for detecting shared memory accessors. Our training phase involves running kernel intensive workloads (including the benchmarks we use for evaluation) several times for a long duration of time (to minimize the chances of under-approximation during identification of shared-memory accessors Section 4.2.2). The output of the training phase is a list of instruction pointers that need to be instrumented during the record phase. We maintain reader/writer locks corresponding to every byte in main memory, in the shadow memory. We use one bit corresponding to every CPU in the shadow byte, and thus our current implementation supports a maximum of eight CPUs.

We implemented our scheme only for the kernel and tested our implementation with applications that share nothing in the user-mode. For completeness, we also implemented the SMP-Revirt like page-granular CREW protocol for the user-mode in addition to byte-granular CREW for the kernel. One of the problems in running CREW with DBT is: the hypervisor does not know whether a physical page belongs to the kernel or the user. Because the hypervisor only needs to run CREW on user pages, we changed the Linux memory allocator to allocate pages for kernel and user from different pools. This allows the hypervisor to distinguish between a user and kernel page easily, based on its address. We have also tested our implementation while running applications with user-mode sharing.

We successfully replayed all our benchmarks with or without sharing. All results in this thesis are generated for only the kernel (by turning off user-mode page-grained CREW). For

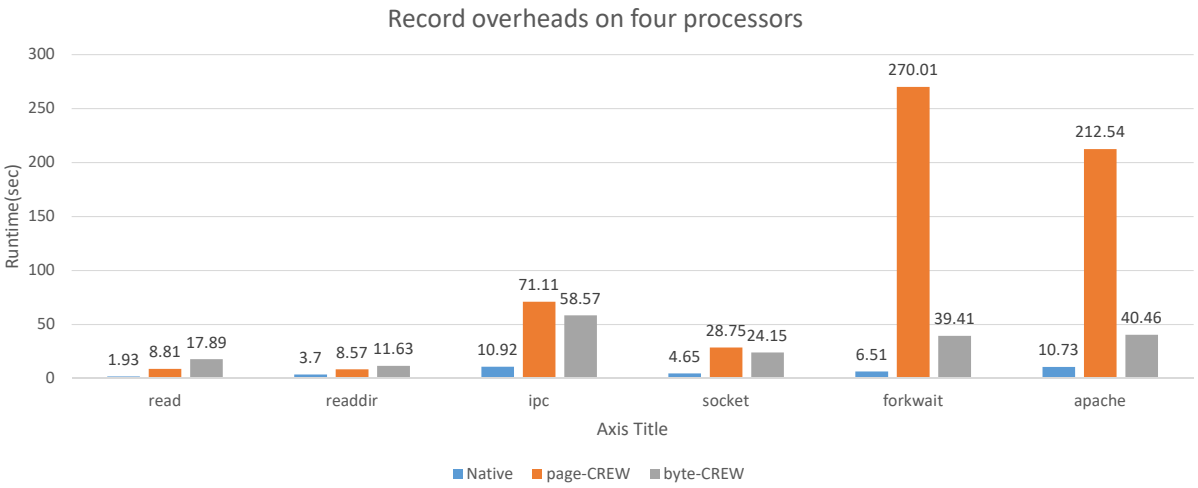


Fig. 4.8 Runtime on 4 processors for native, page-grained CREW, and byte-grained CREW executions.

concurrency, we ran the same application multiple times. We also pin the applications to their respective CPUs, because the migration of a thread from one core to another appears as sharing to the hypervisor, and would distort our results. A more general technique would involve identification of thread-migration events.

We performed experiments to compare with page-grained CREW in terms of recording overheads and log size. Table 4.1 lists our benchmarks. We wrote some micro-benchmarks to exercise different subsystems of the kernel, and to exercise different types of sharing behavior between the different subsystems. The `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` is a real world workload used for the evaluation.

We evaluate the performance of byte-grained CREW with respect to native (KVM) and page-grained CREW. Interestingly, there are a few tradeoffs while using byte-grained CREW, when compared to using page-grained CREW.

- **Log growth:** Byte-grained CREW may result in very high log-growth rates, as compared to page-grained CREW. For example, consider a case, where a CPU is trying to do a bulk copy on a shared page. In page-grained CREW, it would create only one log entry for 4096 bytes (page-size), whereas, for byte-grained CREW, it may create 4096 log entries depending upon the instruction. For example, we translate `rep movsb` instruction to copy one byte at a time in a loop. Most of the generic copy functions like `memcpy`, `strcpy`, etc. use these instructions and create large log sizes in our implementation.

  On the other hand, page-grained CREW results in false sharing and causes extra log

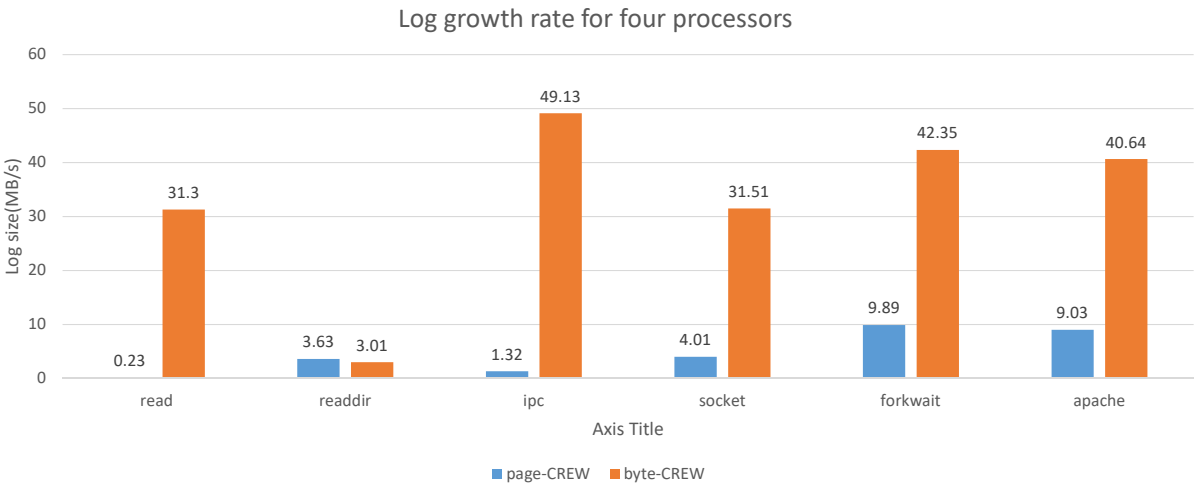


Fig. 4.9 Log growth rate on 4 processors for native, page-grained CREW and byte-grained CREW executions.

entries that are absent in byte-grained CREW. Thus, the log-growth rates for the two schemes are workload-dependent.

- **Concurrency:** As discussed already, the slowpath in page-grained CREW is very slow and sometimes helps the other CPUs to execute in isolation. Due to this, the recorded execution might miss some interesting shared memory interleavings that would have been noticed otherwise during a native run. For example, consider a case where multiple CPUs are trying to access the same page. In the page-granular recording scheme, a trap by one CPU (in failing to access a page because of inadequate permissions) helps the owner CPU(s) to access the page in isolation with the faulty CPU, until the trap handler on the faulty CPU sends an inter-processor interrupt(IPI) to the owner CPU(s) to acquire ownership. On the other hand, the slowpath in byte-grained CREW is relatively fast, and a faulty CPU quickly acquires ownership through shared memory communication (which is much faster than a trap to the hypervisor followed by an IPI). The faster acquisition of ownership in byte-grained CREW facilitates the reproduction of more shared memory interleavings as compared to page-grained CREW. This is particularly useful when record/replay is used to debug a multi-threaded program (in our case, an OS kernel). However, more concurrency causes additional ownerships transfers, which results in higher log growth.

  It is interesting to note that while higher concurrency induces higher overheads, it is also closer in behavior to the native run. Many applications involving testing and debugging, for example, would appreciate the higher-concurrency provided by byte-granular CREW over page-granular CREW.

Figure 4.9 shows the results for log growth rate (MB/s) on four processors. Notice, that byte-grained CREW has substantially higher log growth rate than page-grained CREW for these workloads. Figure 4.8 shows the running time of native, page-grained CREW, and byte-grained CREW on four processors. For `apache` and `forkwait` the byte-grained CREW reports 4-6x overheads as compared to 19-41x overheads of page-grained CREW. The higher log growth rate during the page-grained CREW during `readdir` as compared to `read` benchmark confirms the false-sharing in `readdir`. The high log-growth rate in the `read` benchmark for byte-grained CREW is mostly due to a large number of ownership transfers due to bulk copy. For this benchmark, byte-grained CREW overhead is 9x as compared to the 4x overheads of page-grained CREW. This is because, in page-grained CREW, we do not have any instrumentation overheads, whereas, for byte-grained CREW, we also have to pay instrumentation overheads regardless of sharing. Apart from sharing, byte-grained CREW also suffers from high overheads, if the shared instructions execute very often, due to more frequent ownership transfers. The other benchmarks have a moderate amount of false sharing, and our performance gain for these benchmarks are not significant.

4.4 Discussion

In summary, we implement byte-granular CREW for recording and replaying an operating system kernel. Instead of relying on the page-protection mechanism, we take an instrumentation-based approach. We use a fast DBT framework discussed in Chapter 3 for instrumentation. We only inject interrupts at deterministic points in the kernel. Our scheme allows fast handling of CREW conflicts inside the guest kernel itself, as opposed to the previous works, which handles CREW conflicts inside the hypervisor. In our scheme, the faster handling of CREW conflicts may result in higher log growth rates both due to finer-granularity of tracking and due to higher concurrency. On the other hand, the higher concurrency of byte-granular CREW facilitates more applications of record/replay. Even so, there are many workloads, where the advantages of reducing false-sharing result in reduced overall runtime overheads, when compared to page-granular CREW. Future work may involve dynamically adapting the granularity of ownership tracking, for faster operation.

Our scheme employs a training phase to detect all the instructions that may access a shared variable. Our training phase is best-effort, as it may both under- and over-approximate the identification of shared-memory accessing instructions. By ensuring that the training phase involves high coverage, the probability of such under-approximation is rare, and can be handled in a manner similar to previous search-based replay techniques [3, 51]. The byte-granular CREW solves the problem of false sharing at the cost of instrumentation overheads in the

common case. In the case of large ownership conflicts (very common in an OS kernel), the small overheads of instrumentation always outperform page-granular CREW. Overall, our results provide an empirical study on the various trade-offs involved in using byte-granular vs. page-granular CREW for recording and replaying a full compute system.

Chapter 5

Conclusions

In this thesis, we explore the problem of deterministic replay from a software systems standpoint. We implement a byte-granular CREW protocol to record/replay an operating system kernel with its applications. We present a method based on dynamic binary translation, to explore this design space. We also present a method to implement dynamic binary translation (DBT) for the OS kernel efficiently, which involves efficient and correct handling of asynchronous interrupts and exceptions.

To implement byte-granular CREW, we instrument a reader/writer lock before every instruction that may potentially read/write to shared memory. We employ a 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 order. This scheme outperforms page-granular CREW for many workloads, especially when there is high false-sharing. For workloads involving true-sharing of large memory regions, the overheads of byte-granularity ownership tracking may outweigh the advantages of eliminating false-sharing and reducing ownership transfer overheads.

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, 28] for the kernel exhibit 3-5x overheads for many kernel intensive benchmarks. We implement 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 is perhaps the first to allow dynamic `switchon` and `switchoff` of DBT for a running system.

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 previous work on deterministic replay has largely fo-

cused 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

[2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *International Conference on Architectural Support for Programming Languages and Operating Systems '06*. pages

[3] Gautam Altekar and Ion Stoica. ODR: output-deterministic replay for multicore debugging. In *ACM Symposium on Operating Systems Principles '09*. pages

[4] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 307–320, 2012. pages

[5] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *USENIX Symposium on Operating Systems Design and Implementation '10*. pages

[6] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *ACM/ONR Workshop on Parallel and Distributed Debugging '91*. pages

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

[8] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Concerence '05*. pages

[9] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems '10*. pages

[10] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for c/c++. In *ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications '09*. pages

[11] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *International Conference on Architectural Support for Programming Languages and Operating Systems '92*. pages

[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 *ACM SIGPLAN/SIGOPS international conference on Virtual execution environments '06*. pages

[13] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems '08*. pages

[14] 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

[15] 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

[16] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *ACM Symposium on Operating Systems Principles '95*. pages

[17] Derek Bruening. *Efficient, Transparent and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004. pages

[18] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Transactions on Computer Systems (TOCS)*, 30(4):12, 2012. pages

[19] Yufei Chen and Haibo Chen. Scalable deterministic replay in a parallel full-system emulator. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming '13*. pages

[20] 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

[21] Jonathan Corbet. pages

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

[23] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *USENIX Symposium on Operating Systems Design and Implementation'10*. pages

[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

[25] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: Deterministic shared memory multiprocessing. In *International Conference on Architectural Support for Programming Languages and Operating Systems '09*. pages

[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 *USENIX Symposium on Operating Systems Design and Implementation '02*. pages

[27] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *ACM SIGPLAN/SIGOPS international conference on Virtual execution environments '08*. pages

[28] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *International Conference on Architectural Support for Programming Languages and Operating Systems '12*. pages

[29] Stuart I Feldman and Channing B Brown. Igor: A system for program debugging via reversible execution. In *ACM Sigplan Notices*, volume 24, pages 112–123. ACM, 1988. pages

[30] Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, volume 44, pages 121–133. ACM, 2009. pages

[31] Nima Honarmand and Josep Torrellas. Relaxreplay: Record and replay for relaxed-consistency multiprocessors. In *International Conference on Architectural Support for Programming Languages and Operating Systems '14*. pages

[32] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ACM SIGARCH Computer Architecture News '08*. pages

[33] Intel 64 and IA-32 architectures software developer's manual volume 2. `http://www.intel.com/products/processor/manuals/`. pages

[34] 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

[35] Piyus Kedia and Sorav Bansal. Fast dynamic binary translation for the kernel. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 101–115. ACM, 2013. pages

[36] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Intrusion recovery using selective re-execution. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 89–104, 2010. pages

[37] Naveen Kumar, Bruce R Childers, and Mary Lou Soffa. Low overhead program monitoring and profiling. *ACM SIGSOFT Software Engineering Notes*, 31(1):28–34, 2006. pages

[38] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *ACM SIGMETRICS performance evaluation review*, volume 38, pages 155–166. ACM, 2010. pages

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

[40] 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

[41] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient online multiprocessor replayvia speculation and external determinism. In *International Conference on Architectural Support for Programming Languages and Operating Systems '10*. pages

[42] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *ACM Symposium on Operating Systems Principles '11*. pages

[43] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ACM SIGARCH Computer Architecture News '08*. pages

[44] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *International Conference on Architectural Support for Programming Languages and Operating Systems '06*. pages

[45] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ACM SIGARCH Computer Architecture News '05*. pages

[46] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *ACM/ONR Workshop on Parallel and Distributed Debugging '93*. pages

[47] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005. pages

[48] 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

[49] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *International Conference on Architectural Support for Programming Languages and Operating Systems '09*. pages

[50] 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

[51] 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 *ACM Symposium on Operating Systems Principles '09*. pages

[52] 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

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

[54] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation '96*. pages

[55] 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

[56] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on*, pages 317–331. IEEE, 2010. pages

[57] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005. pages

[58] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference '04*. pages

[59] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: parallelizing sequential logging and replay. In *International Conference on Architectural Support for Programming Languages and Operating Systems '11*, pages 15–26, 2011. pages

[60] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems '10*. pages

[61] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News '95*. pages

[62] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ACM SIGARCH Computer Architecture News '03*. pages

[63] 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

[64] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *ACM SIGPLAN/SIGOPS international conference on Virtual execution environments '11*. pages

[65] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. How to do a million watchpoints: efficient debugging using dynamic instrumentation. In *CC'08/ETAPS'08*. pages

Bio Data

Name: Piyus Kedia
Date of Birth: 22 Dec, 1988
Education: University of Kalyani, B. Tech., May 2010
Publication: Piyus Kedia and Sorav Bansal. Fast dynamic binary translation for the kernel. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 101-115. ACM, 2013.