

DYNAMIC BINARY TRANSLATION FOR DETERMINISTIC REPLAY

PIYUS KEDIA



AMARNATH AND SHASHI KHOSLA SCHOOL OF IT

INDIAN INSTITUTE OF TECHNOLOGY DELHI

NOVEMBER 2017

©Indian Institute of Technology Delhi (IITD), New Delhi, 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 डिज़ाइन 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 है, जो सही ढंग से कर्नेल तंत्र, जैसे कि इंटरप्ट, एक्सेप्शन 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 है। 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 ग्रैनुलैरिटी [27] पर सीआरईडब्ल्यू (समवर्ती पढ़ें अनन्य लिखें) प्रोटोकॉल एक संभावित दृष्टिकोण है। सीपीयू के एक्सेस विशेषाधिकार को प्रतिबंधित करने के लिए पृष्ठ ग्रैनुलैरिटी 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 पुनः उत्पन्न करने के लिए दर्ज किया जाता है। यह पृष्ठ-गैन्ग्लर 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 गैन्ग्लरैरिटी 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

Table of contents	xiii
List of figures	xv
List of tables	xix
Nomenclature	xix
1 Introduction	1
2 Related Work	7
3 Fast Dynamic Binary Translation for the kernel	19
3.1 Background	19
3.2 Kernel-mode DBT	29
3.3 A faster design	30
3.4 Design subtleties	32
3.5 Potential inconsistencies due to code-cache addresses living in guest data structures	36
3.6 Optimizations	40
3.6.1 Call-ret optimization	40
3.6.2 Code Cache Optimization	42
3.6.3 Indirect branch optimization	43
3.6.4 Per-CPU code-cache vs shared code-cache	43
3.6.5 Jumptable optimization	44
3.7 Translator switchon and switchoff	45
3.8 Implementation and Results	46
3.8.1 Implementation	46
3.8.2 Experimental Setup and Benchmarks	46

3.8.3	Performance	47
3.8.4	Scalability	53
3.9	Discussion	53
4	Deterministic Replay	55
4.1	Background	55
4.2	Our technique	57
4.2.1	Shadow memory	58
4.2.2	Global variable detection	58
4.2.3	Byte-granularity CREW	60
4.2.4	Lock implementation	60
4.2.5	Lock implementation for relaxed memory models	64
4.2.6	Replay	66
4.2.7	Asynchronous events in kernel execution	68
4.3	Experiments	69
4.4	Discussion	74
5	Conclusions	77
	References	79

List of figures

2.1	Instant replay implementation of reader/writer locks.	8
2.2	VMware’s software virtualization overheads.	16
2.3	Dynamo-Rio kernel (DRK) overheads.	17
3.1	A DBT framework	20
3.2	An illustrative example of dynamic binary translation.	21
3.3	Dispatcher orchestrating the dynamic execution of the loop example.	23
3.4	Dispatcher: Dispatcher first saves the guest registers in its memory and then switch to its own stack. It then searches for <code>nextpc</code> 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.	24
3.5	Direct branch chaining: Translated basic blocks are linked together for efficiency. The dispatcher overwrites the target addresses <code>edge1</code> and <code>edge2</code> with <code>tx-BB3</code> and <code>tx-BB2</code> after translating the respective basic blocks.	25
3.6	Code block: does not terminate at a conditional branch instruction. e.g., a code block starting at <code>BB3</code> in loop example only terminates on function return.	26
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 <code>call</code> instruction. The edge block transfers control to the dispatcher.	26
3.8	At runtime, a hash function is applied to the target <code>pc</code> to index into the jumtable. The jumtable is a hash-table storing the mapping between a native PC value (<code>pc</code>) and its translated code-cache address (<code>tx-pc</code>).	27

3.9	The translation of an example indirect instruction, <code>jmp *MEM</code> . The native code is shown in the Input block. The translated code is shown in the Output block. The translated code looks up the jumtable (call to <code>lookup_hash()</code>); if found, it jumps to the corresponding translated code address (<code>jtarget</code>); if not found, it jumps to the dispatcher. The dispatcher does a complete lookup to determine the translated address for <code>nextpc</code> . If not already translated, the dispatcher translates the code block starting <code>nextpc</code> before restoring the guest state, and jumping to the translated address. . . .	28
3.10	DBT framework for the kernel.	32
3.11	The dispatcher code uses a shared variable (<code>jtarget</code>) to store the translated PC, before restoring the guest state and indirectly jumping to the address stored in <code>jtarget</code> . Between lines 5 and 6, the interrupts may be enabled, which may cause re-entrancy issues on access to the shared variable <code>jtarget</code>	33
3.12	The dispatcher code uses a register <code>ecx</code> to save the translated code cache address (<code>tx_pc</code>). 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 <code>ecx</code> 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.	35
3.13	An example translation of an indirect instruction. Interrupts need to be disabled before <code>lookup_hash()</code> call to avoid race conditions on the jumtable (hash table) with the dispatcher. Similarly, a per-CPU <code>jtarget</code> 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 <code>jtarget</code> . This race condition is solved by saving/restoring <code>jtarget</code> as discussed in Section 3.4.	36
3.14	Pseudo-code showing registry of custom page fault handlers by kernel subsystems in BSD kernels. The <code>pcb_onfault</code> 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 <code>pcb_onfault</code>	39
3.15	A code block could potentially contain multiple call instructions. The <code>target_offset</code> allows the dispatcher to know where to patch the translated code address of the corresponding call target, for direct block chaining.	41
3.16	The translation of an example indirect call instruction. <code>target_offset</code> is used for direct branch chaining.	42

3.17	The translated (pseudo) code generated for a code block involving multiple conditional branches (<code>gcc</code>).	43
3.18	The translation of an example indirect branch instruction “ <code>jmp *MEM</code> ”, which checks against one hardcoded address, <code>pc_target</code> , before looking up the jumtable.	44
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.	45
3.20	<code>lmbench</code> fast operations	48
3.21	<code>lmbench</code> fork operations	48
3.22	<code>fileserv</code> on 1, 4, 8, and 12 processors	48
3.23	<code>webserver</code> on 1, 4, 8, and 12 processors	49
3.24	<code>webproxy</code> on 1, 4, 8, and 12 processors	49
3.25	<code>varmail</code> on 1, 4, 8, and 12 processors	49
3.26	<code>lmbench</code> communication related operations	50
3.27	<code>Apache</code> on 1, 2, 4, 8, and 12 processors	50
3.28	<code>lmbench</code> fast operations with indirect branch optimization	51
3.29	<code>lmbench</code> communication related operations with indirect branch optimization	51
4.1	A sample function accessing a shared variable.	55
4.2	A software-only implementation of CREW.	57
4.3	Pseudo-code of our instrumented reader-writer lock routines. The <code>thread_t</code> structure stores per-CPU state. The <code>owners</code> 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 <code>read_acquire()</code> function checks if the current thread is one of the owners. The <code>write_acquire()</code> function checks if the current thread is the only owner. If the check fails, the <code>acquire_slowpath</code> routine in Figure 4.4 is called to update the ownership information.	61
4.4	Pseudo-code of our instrumented reader-writer slowpath code. The <code>acquire_slowpath()</code> function waits for the current owner CPUs to leave the critical section (i.e., wait for their <code>brlock</code> 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.	62
4.5	Pseudo-code of our instrumented reader-writer lock routines for x86 TSO memory model.	66
4.6	Pseudo-code of our instrumented reader-writer slowpath for x86 TSO memory model.	67

4.7	Pseudo-code of our instrumented replay routine. The <code>replay_head</code> routine waits for all the CPUs to reach their deterministic point. The <code>replay_tail</code> function simply increments the expired shared memory accesses count.	68
4.8	Runtime on 4 processors for native, page-grained CREW, and byte-grained CREW executions.	72
4.9	Log growth rate on 4 processors for native, page-grained CREW and byte-grained CREW executions.	72

List of tables

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.	37
3.2	Linux build time for 1 and 12 CPUs	52
3.3	Statistics on the total number of instructions executed, number of indirect instructions executed, number of without collision (Fastpath) jumtable hits, number of with collision (Slowpath) jumtable 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.	52
4.1	Description of Benchmarks	71