Optimizing Control Transfer and Memory Virtualization in Full System Emulators

DING-YONG HONG, Academia Sinica

CHUN-CHEN HSU, National Taiwan University

CHENG-YI CHOU, Academia Sinica

WEI-CHUNG HSU and PANGFENG LIU, National Taiwan University

JAN-JAN WU, Academia Sinica

Full system emulators provide virtual platforms for several important applications, such as kernel and system software development, co-verification with cycle accurate CPU simulators, or application development for hardware still in development. Full system emulators usually use dynamic binary translation to obtain reasonable performance. This paper focuses on optimizing the performance of full system emulators. First, we optimize performance by enabling classic control transfer optimizations of dynamic binary translation in full system emulation, such as indirect branch target caching and block chaining. Second, we improve the performance of memory virtualization of cross-ISA virtual machines by improving the efficiency of the software translation lookaside buffer (software TLB). We implement our optimizations on QEMU, an industrial-strength full system emulator, along with the Android emulator. Experimental results show that our optimizations achieve an average speedup of 1.98X for ARM-to-X86-64 QEMU running SPEC CINT2006 benchmarks with train inputs. Our optimizations also achieve an average speedup of 1.44X and 1.40X for IA32-to-X86-64 QEMU and AArch64-to-X86-64 QEMU on SPEC CINT2006. We use a set of real applications downloaded from Google Play as benchmarks for the Android emulator. Experimental results show that our optimizations achieve an average speedup of 1.43X for the Android emulator running these applications.

CCS Concepts: • **General and reference** → **Metrics**; • **Software and its engineering** → **Virtual machines**; **Runtime environments**

Additional Key Words and Phrases: Control transfer optimizations, memory virtualization optimizations

ACM Reference Format:

Ding-Yong Hong, Chun-Chen Hsu, Cheng-Yi Chou, Wei-Chung Hsu, Pangfeng Liu, and Jan-Jan Wu. 2015. Optimizing control transfer and memory virtualization in full system emulators. *ACM Trans. Archit. Code Optim.* 12, 4, Article 47 (December 2015), 24 pages.
DOI: <http://dx.doi.org/10.1145/2837027>

1. INTRODUCTION

A full-system emulator is a software-based approach to emulating the entire instruction set architecture (ISA), including privileged instructions, such that an entire guest

This work is supported in part by the Ministry of Science and Technology of Taiwan under grant number NSC102-2221-E-001-034-MY3.

Authors' addresses: C.-C. Hsu, P. Liu, and W.-C. Hsu, Computer Science and Information Engineering Department, National Taiwan University, No. 1, Sec. 4, Roosevelt Rd., Taipei, Taiwan; emails: {d95006, pangfeng, hsuwc}@csie.ntu.edu.tw; D.-Y. Hong, C.-Y. Chou, and J.-J. Wu, Institute of Information Science, Academia Sinica, 128 Academia Road, Section 2, Nankang, Taipei 115, Taiwan; emails: {dyhong, maxchou, wuj}@iis.sinica.edu.tw.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2015 ACM 1544-3566/2015/12-ART47 \$15.00

DOI: <http://dx.doi.org/10.1145/2837027>

OS could be booted up in the emulator and run virtually on a host OS with a completely different ISA. Such virtualization systems have many important and practical applications, such as enabling the implementation of secure environments in which operating systems are isolated, or to speed up CPU execution flow tracing and OS kernel debugging by emulating a slower platform (e.g., ARM) on a faster one (e.g., x86-64).

This article focuses on full-system emulation with dynamic binary translation (DBT) techniques, especially for cross-ISA system-level emulation, that is, the guest and the host belong to different ISAs. Dynamic binary translation allows programs compiled for one ISA (e.g., Intel IA-32) to be run on platforms based on a different ISA (e.g., IA-64). For example, commercial products such as BlueStacks [2015] and AMIDuOS [2015] use cross-ISA system emulators to run ARM Android applications on AMD64 Windows machines. Android application developers also use ARM64 Android emulator to test their applications when ARM64 machines are not available.

Popular full-system emulators such as QEMU [Bellard 2005] and Simics [Magnusson et al. 2002] use DBT techniques. Improving the performance of system-level DBT involves overcoming many challenges that are different from those faced by DBT at the process (application) level. For process VMs, the host OS is the OS. The memory address space of process virtual machines (VMs) is managed by the host OS and the DBT's job is to map the virtual address space of the process to the host virtual memory. For a system VM, however, the memory used in each process of the guest VM is managed by the guest OS. This raises two problems.

(1) The virtual address in each process must be mapped to the guest physical address that is managed by the guest OS, and the guest physical address is allocated and assigned by the host OS. Thus, the guest physical address must be mapped to the host virtual address in an additional step.

(2) All software-based caching techniques used to improve memory access in process VMs are now subject to the condition that the virtual addresses are managed by the guest OS, and may be changed by the guest OS during context switching or system calls. A naive approach is to flush such caches, but this could significantly increase the cost of context switching. Hence, all optimizations related to memory access introduced to process VMs must be rethought and redesigned.

In this article, we investigate two optimizations related to memory access: (1) branch optimization, including block linking [Cmelik and Keppel 1994] and indirect branch translation cache (IBTC) [Scott et al. 2004]; and (2) software-based translation lookaside buffer (software TLB) to improve the performance of address translation. We discuss design issues encountered during implementation in system mode DBT. We also propose effective methods to solve these problems.

For branch optimization, the first issue is that, when a cross-page branch is executed, the DBT must ensure that the branched guest page is valid, otherwise an exception should be raised. Another issue is that the DBT must efficiently and effectively detect the validity of the branches across page boundaries. To solve the cross-page problem, we introduce the software *instruction TLB (iTLB)* to efficiently validate cross-page branches and to mitigate the large overhead incurred by walking the guest page table. *Cross-page block linking (CPBL)* is also proposed to handle direct branches across pages. With the proposed approaches, the emulation performance is further enhanced as a result of re-enabling the optimizations of block linking and IBTC in the full-system DBTs.

For memory translation optimization, to speed up the aforementioned multilevel memory translation in system VMs, similar to hardware TLB, DBT systems, such as QEMU, keep the latest memory translations in a special cache, called *Software Translation Lookaside Buffer (SoftTLB)*, to translate a guest virtual address directly to a host virtual address. However, even with SoftTLB, memory translation still consumes

a significant portion of execution time. For example, Chang et al. [2014] reported that QEMU spends nearly 40% of execution time in memory translation.

We propose two optimizations to improve the performance of SoftTLB in cross-ISA system mode emulation. We begin by identifying the overhead induced by inefficient support of multiple page sizes in SoftTLB. We find that most overhead comes from unnecessary SoftTLB flushes due to invalidation of large pages. We propose an optimization, *SoftTLB partial-flush*, to precisely track the SoftTLB entries used for large pages so that we only need to flush these used entries instead of the whole SoftTLB. Second, the optimization *Dynamically Resizing SoftTLB* improves performance by increasing the SoftTLB hit rate and avoiding unnecessary overhead for SoftTLB flushing. The key idea in this optimization is to resize SoftTLB according to the per-process SoftTLB utilization.

We implemented these optimizations on the official QEMU v2.2.0 [QEMU 2015] emulator and the Android emulator of the Android Open Source Project (AOSP) [Android-Emulator 2015] version 5.0.1_r1, which is also based on QEMU. Our experimental results demonstrate that, for ARM-to-X86_64 QEMU emulation, our optimization achieves an average 1.98X speedup against the unmodified official QEMU v2.2.0 for SPEC 2006 CPU Integer benchmarks and a speedup of up to 3X for a memory-bound, cache-sensitive benchmark. Our optimizations also achieve an average speedup of 1.44X and 1.40X for IA32-to-X86-64 QEMU and AArch64-to-X86-64 QEMU on SPEC CINT2006. For the Android emulator, our optimization achieves an average 1.43X speedup against the unmodified Android emulator for Android benchmarks. We also put our work on GitHub.com for interested readers to download. Please refer to Section 6.

The rest of the article is organized as follows. Section 2 gives an overview of control transfer optimization and software TLB optimization, with a discussion of related work. Section 3 describes the control transfer problem in cross-ISA system-level emulation and presents our approach to solving this problem efficiently. Section 4 presents our optimizations for software TLB. Section 5 reports our experimental results, and Section 6 presents our conclusions.

2. BACKGROUND AND RELATED WORK

Dynamic Binary Translation

Dynamic binary translators (DBTs) emulate a guest binary code in one ISA on a host machine with a same or different ISA. It operates directly on binaries with no need to access the source code of the guest operating systems and applications, which is important for executing unmodified and proprietary guests. There are generally two types of binary translators: user-mode DBT and full-system DBT. User-mode DBTs emulate an application binary interface, while full-system DBTs emulate the entire guest ISA interface, including privileged instructions. DBTs have been widely used for runtime profiling [Witchel and Rosenblum 1996; Luk et al. 2005; Nethercote and Seward 2007; Zhao et al. 2010], transparent performance optimization [Bala et al. 2000; Bruening et al. 2003] and migration of legacy code [Baraz et al. 2003; Chernoff et al. 1998; Zheng and Thompson 2000]. In this section, we briefly introduce DBT architecture, the virtualization of CPU and memory for full-system virtualization, and related works.

Figure 1 illustrates the architecture of a DBT system. A typical DBT system generally has four components: a just-in-time (JIT) compiler, an emulation unit, a dispatcher, and a code cache. When the translation starts, the JIT compiler fetches guest binary code and translates it to the binary code of the host ISA. The translated host code is cached in a software-based code cache to enable reuse and to amortize code translation overhead. The emulation unit provides special handlers for exceptions and interrupts,

Fig. 1. A general framework of a dynamic binary translator.

for example, emulating I/O devices. The dispatcher coordinates the translation and execution of binary code. It determines whether to resume execution in the code cache or to kick-start the JIT compiler if an untranslated guest code is encountered.

For CPU virtualization, most modern CPU architectures contain sets of privileged and unprivileged instructions. When attempting to execute such instructions in a de-privileged mode, the misbehaving instructions must be trapped and correctly handled. Previous work, such as KVM [Kivity et al. 2007] and Xen [Barham et al. 2003], has proposed hardware-assisted or paravirtualized approaches to trap and emulate such sensitive instructions.

In contrast, DBTs, such as VMWare [Adams and Agesen 2006] and QEMU [Bellard 2005], solve such CPU virtualization problems through binary rewriting. While translating the guest binary, the semantic of the guest instructions is translated based on current guest CPU states and privilege levels. The sensitive instructions issued from the guest system are also translated to safe host instructions. Hence, one instruction in different guest-privileged modes is translated into precise emulation code. In addition, the translator emits trapping code around an illegal instruction once it is detected during translation.

Control Transfer Optimizations

Control transfer optimizations refer to optimizations that can transfer execution directly from one translation block to another without interference from the runtime system. Existing control transfer optimizations and superblock optimizations in dynamic binary translation (e.g., block chaining [Cmelik and Keppel 1994; Witchel and Rosenblum 1996; Smith and Nair 2005], trace optimization [Bala et al. 2000; Bruening et al. 2003; Luk et al. 2005; Hsu et al. 2013; Hong et al. 2012], and indirect branch target caching [Scott et al. 2004]) have been shown to be effective and have been widely adapted to high-level language VMs [Inoue et al. 2011], or dynamic scripting languages [Gal et al. 2009; Bebenita et al. 2010].

However, control transfer optimizations are not implemented as easily as user-mode simulations because the target code page may not be valid among different processes. For example, it is possible for two processes to map a common code page at the virtual address of a control transfer instruction and have different code pages mapped at the target address. As a consequence, control transfer instructions that jump between these two virtual addresses will generate chains that are not valid for all of the processes sharing the page.

Thus, the control transfer optimizations are not widely used in system mode emulation, or only adopted conditionally. For example, QEMU [Bellard 2005], a retargetable DBT system supporting many guest and host ISAs, applies the optimization of block

linking only for branches among the same guest page. In another example, Bohm et al. [2011] built traces with the limitation that blocks from a single trace cannot span across page boundaries.

There are hardware-based and software-based solutions for cross-page control transfers in system emulation. IBM's DAISY system [Ebcioglu et al. 2001] runs a PowerPC guest on a VLIW machine. DAISY formed several basic blocks into a tree-region that can leverage the advantage of VLIW architecture to increase the number of instructions per cycle (IPC). To address transfers that cross-page boundaries, a special hardware instruction called `LOAD_REAL_ADDRESS_AND_VERIFY` (LRAV) is used to detect whether a code page is valid and to check that mapping does not change with the creation of the tree-region. The Transmeta code-morphing software [Dehnert et al. 2003] forms traces that contain several blocks, and chains translation blocks. Although they did not explicitly mention how to handle the page-boundary problem, they mention that, with hardware support for commit and rollback, they can preserve precise exceptions that happen during execution at the x86 instruction boundary.

Embrea [Witchel and Rosenblum 1996] and PinOS [Bungale and Luk 2007] adopt software-based approaches for cross-page control transfers. Both Embrea and PinOS maintain the set of current valid guest virtual pages and check the validity at the entry of each translation block or trace. However, checking the validity at the entry of each translation block or trace may degrade performance because the validity checks are not needed if the target addresses are in the same page as the control instructions. In Section 3, we investigate the cross-page problem in control transfer and propose two software-based optimizations.

Memory Virtualization

Memory virtualization optimization has been well studied and developed in same-ISA virtualization. For example, before hardware-assisted virtualization was supported in AMD64 CPUs, Xen and VMWare [Adams and Agesen 2006] developed shadow-page table approaches to efficiently support memory virtualization. The shadow-page table is essentially the cache of the address translation. The key question is how to maintain coherence between the shadow-page table and the guest-page table.

To maintain coherence, Xen modifies the guest kernel in a process called paravirtualization, so that the virtual machine manager (VMM) is notified when the guest kernel is about to modify the guest-page table. On the other hand, VMWare uses a hardware-page protection technique called tracing to become trapped when the guest-page table is modified so that VMM can maintain coherence between the shadow-page table and the guest-page table. Intel VT-x and AMD-V provide a hardware-assisted mechanism for memory virtualization to efficiently support same-ISA virtualization in AMD64 machines. KVM [Kivity et al. 2007], for example, uses this hardware-assisted memory virtualization mechanism.

Without specially designed hardware, memory virtualization in cross-ISA virtualization can use only software-based approaches. Embrea and QEMU use such software-based approaches. The difference between Embrea and QEMU is that Embrea takes the advantage of architected TLB capability in MIPS R3000 to maintain SoftTLB, which is called *MMU relocation array* in Embrea's terminology. With architected TLB, Embrea avoids flushing SoftTLB entries when context switches happen. It only needs to update the changed entries of SoftTLB. On the other hand, QEMU supports many guest CPUs, some of which (e.g., IA32, AMD64, and ARM) do not support architected TLB. Therefore, QEMU needs to flush the entire SoftTLB when processes are context-switched. Tong et al. [2015] propose several techniques to increase SoftTLB hit rates as well as to reduce the overhead of SoftTLB maintenance in QEMU. In particular, they improve performance by resizing SoftTLB to increase the hit rate, using victim cache to reduce

the overhead of SoftTLB misses, and using helper threads to flush SoftTLB to reduce flushing overhead.

In this article, we deal with the cross-page problem of control transfer optimizations that Tong et al. [2015] do not discuss in their work. They also fail to mention the efficiency problem of SoftTLB described in Section 4.1. Although both works propose dynamical resizing SoftTLB approaches, our approach adjusts the size of SoftTLB according to the per-page-table (per-process-like) SoftTLB utilization information instead of system-wide SoftTLB utilization information.

3. ENABLING CONTROL TRANSFER OPTIMIZATIONS

Cross-Page Problem

Dynamic binary translators translate and execute guest code, often at a granularity of one basic block. To enhance execution performance and avoid frequent switching between the emulation engine and the execution in code cache, common control transfer optimizations such as *block linking* [Cmelik and Keppel 1994] and *indirect branch translation caching (IBTC)* [Scott et al. 2004] are used, respectively, for direct and indirect branches. For simplicity, the following discussion refers to the *guest code page* of the branch target as the target guest page.

When the target guest page differs from the target page of the branch instruction, control transfers across pages in system mode could cause system failure if not handled properly. This is because the target guest page may no longer be mapped in the guest-page tables, thus jumping to an invalid code page will crash the system. Even if the code page is valid in the guest-page table, the mapping of the target code page may be changed after the transfer link is created. Consequently, jumping along the transfer link will result in executing code in a stale page. This usually occurs when the guest operating system performs context switches.

Therefore, when transferring execution controls across pages, we must check that (1) the target guest page is valid in the guest-page table, and (2) the guest physical address of the target guest page remains the same when the transfer link is created. Violating either of these two conditions (e.g., the mapping of the target guest page is changed or is not valid or present in the guest-page table) should be trapped and invoke a page fault in the guest operating system. User-mode DBTs do not require examining these two conditions before jumping to blocks of any page because the conditions will be resolved by the host operating system transparently if violation occurs. Thus, no examining code is emitted around the translated branch instructions. In the following, we introduce two approaches using *software instruction translation lookaside buffer (iTLB)* to efficiently check the validity of pages.

3.1. Page Validation Check with Virtual iTLB

The first approach of page validation is the virtual-iTLB approach. We begin by classifying the branch instructions into two categories: *direct branch across page boundary*, and *indirect branch*. For direct branches that do not cross page boundary, the technique of block linking is applied in the same way as in user-mode DBTs, and no examination on the target guest page is needed because the page remains valid while executing the two blocks.

Indirect branches, such as indirect jump, indirect call, and return instruction, are optimized with IBTC when translating these instructions. IBTC is usually designed as a hash table for fast lookup inside the code cache. Figure 2(a) shows an IBTC lookup example in user-mode DBTs. The lookup in user-mode DBTs is fed the guest virtual address plus some runtime flags. The runtime flags represent the system status, such as privilege level. An IBTC hit returns the next translated code address to jump to. In

Fig. 2. IBTC implementation in user-mode and full-system DBTs. GuestPC and Flag, respectively, represent virtual address of guest basic block and runtime CPU flags; HostPC represents host virtual address of translated code in the code cache; GuestPageNo represents the guest virtual page number. An invalid value is set as -1 .

a system-mode DBT, we need to determine the validity of the jumped guest page if the branch crosses the guest-page boundary.

All indirect branches have to be examined since the indirect branch address is unknown at translation time, and the given information (i.e., the address and flags) is insufficient to determine whether the indirect branch lies within the same guest page or not.

One naïve approach to validating a guest page is to walk the guest-page table; however, this would significantly lengthen the lookup time for every IBTC lookup.

To mitigate such overhead, we introduce a *software virtual iTLB*, virtual iTLB for short, in our DBT system as shown in Figure 2(b). The concept of the virtual iTLB is similar to the hardware TLB, which uses a CPU cache to improve virtual address translation speed. Unlike the hardware TLB, in which virtual-to-physical address mappings are recorded for both data and code pages, our virtual iTLB is a simplified hash table that caches the *guest virtual page number* for the *code page* only.

The virtual page number of the indirect branch address must be matched against the virtual iTLB before jumping across pages. Upon a virtual iTLB miss, the execution goes back to the DBT's dispatcher, performs expensive guest-page table walking, and records the virtual page number in the virtual iTLB if the guest page is valid. Upon a hit, the indirect branch is then allowed to jump to the next guest block. As the example in Figure 2(b) illustrates, the branch address 0x8111 and flag 0x4 are looked up against the IBTC and iTLB hash tables. Assuming that the guest page size is 4KB, the same values are found in the IBTC hash table, as well as the page number 0x8 in the iTLB table. The translated code address 0x1234 is then returned, and the execution jumps directly to that address without leaving the code cache.

We enable CPBL for direct branches across pages by inserting check code before jumping (linking) to the target code page. The check code first loads the page number from the iTLB table, and checks if it is the same as the page number of the target address. If it is, the execution is transferred to the target code page. Otherwise, the execution will be transferred to the dispatcher to perform block linking again to the right code page. The check code consists of only 3 instructions: a load, a compare, and a conditional branch instruction. There is no need to compute the index because the target address of direct branch is known at translation time.

When a context switch happens in the guest operating system, we need to flush the virtual iTLB since the target page tables may be changed. Virtual iTLB only ensures that the target code page is valid when virtual iTLB hits, but we cannot ensure that the

mapping of the target code page remains the same. Therefore, when a context switch happens in the guest operating system, we need to flush the virtual iTLB and IBTC table.

Also, if the guest OS invalidates a code page, we invalidate the corresponding entries in both tables. We check each entry and flush the entry if it belongs to the invalidated guest page.

3.2. Optimization with Physical iTLB

This flushing of tables limits the performance improvement of IBTC and CPBL. Flushing imposes extra overhead, thus the table size should be restricted to ensure that performance is not affected. However, limiting the table size limits the hit rate. To overcome these shortcomings, we propose the *physical iTLB* approach, in which we store the guest virtual page number and the physical page number in iTLB, and the IBTC table contains both the guest physical and virtual addresses of the branch target. With the physical target address information, we can check the second condition that we fail to check in virtual iTLB. Before jumping across pages, both the page numbers of the guest virtual address and the guest physical address of the branch target must be matched against the physical iTLB.

When context switches occur, we need only to flush the physical iTLB. The IBTC table does not need to be flushed because the lookup of the IBTC table will be missed after the physical iTLB is flushed, and no cross-page branch will be taken. Compared to virtual iTLB, the shortcoming of this approach is that it requires extra comparison for physical addresses, but it poses advantages in that (1) we can avoid the flushing overhead induced from the IBTC table; (2) the entries in IBTC tables can survive across context switches to increase the hit rate; and (3) we can enlarge IBTC tables to improve performance by increasing the hit rate.

In our implementation, we insert the iTLB check code at the end of translation blocks that contain cross-page direct branches and indirect branches. We refer to this approach as *lazy-check* as opposed to the *proactive-check* approach in Embra [Witchel and Rosenblum 1996] and PinOS [Bungale and Luk 2007] which inserts the check code in the entry of every translation block. Embra is *proactive* because it checks page validity for every block while our approach is *lazy* because it checks the validity of the target guest page only when a cross-page branch is about to be taken.

Compared to proactive check, lazy check can prevent unnecessary checks. For example, suppose that Block A and Block C both jump to Block B, and Block A and B are in the same page while Block C and B are in different pages. Transfers from A to B are not checked in lazy check because A and B are in the same page so that B must be valid when jumping from A to B. But transfers from C to B are checked in lazy check because C and B are in different pages. By inserting the check in the beginning of Block B, the code will check all transfers to it, including A-to-B transfers, which is not necessary. By inserting the check at the end of Block C, as we did in the lazy-check approach, we make sure that the code checks only transfers from C to B. We will compare the performance of these two approaches in Section 5.1.2.

4. OPTIMIZATIONS FOR SOFTWARE TRANSLATION LOOKASIDE BUFFER

In this section, we propose two optimizations to improve the efficiency of the SoftTLB of address translation. The first optimization reduces unnecessary SoftTLB flushes induced by large page invalidation. The second optimization dynamically resizes SoftTLB such that we can increase the SoftTLB hit rate while reducing the SoftTLB flush overhead.


```

1 LOOKUP_sTLB:
2   mov  %ebp, %esi
3   and  $0xfffffc03, %esi
4   mov  %rbp, %rdi
5   shr  $0x5, %rdi
6   and  $0x1fe0, %edi
7   lea  0x4c8(%r14, %rdi, 1), %rdi
8   cmp  (%rdi), %esi
9   mov  %ebp, %esi
10  jne  LOOKUP_MISS
11
12 LOOKUP_HIT:
13   add  0x10(%rdi), %rsi
14   mov  (%rsi), %ebp

```

Fig. 3. Routine of SoftTLB lookup.

4.1. Supporting Multiple Page Sizes in Guest CPU

Modern microprocessors support multiple page sizes to reduce the number of TLB misses and the number of page lookups, such as IA32/AMD64 [Intel 2015] and ARM [ARM 2007]. To run a guest system with multiple page sizes, a VM must also support multiple page sizes in its SoftTLB.

Before describing how to support multiple page sizes in SoftTLB, we first introduce the basic mechanism of software-based TLB. SoftTLB is usually implemented as a directly mapped hash table relying on virtual guest addresses for efficiency. This is because, unlike the fully associative hardware TLB, SoftTLB cannot search its content in a parallel manner. For example, as shown in Figure 3, ARM-to-x86_64 QEMU takes 9 instructions to look up the SoftTLB.

In the LOOKUP_sTLB routine, we obtain the page-frame address from Lines 3 and 5. In Lines 2, 4, 6, and 7, we look up SoftTLB by right-shifting and bit-AND operations. We compare the page-frame address (stored in %esi) and the address in SoftTLB (stored in %edi) to determine whether it is hit or miss.

The direct-map SoftTLB works well for a single-page-sized architecture. There are many ways to extend the direct map SoftTLB to support multiple page sizes. One possible solution is to have one SoftTLB for each page size. For example, in ARM architecture, the pages can be 4KB, 64KB, 1MB, and 16MB. The ARMv5 also supports tiny 1KB page sizes. This introduces the complexity of SoftTLB lookups. That is, to look up a guest virtual address, we have to first decide which SoftTLB should be used. Tong et al. [2015] experimentally adapted this approach.

To maintain one rather than multiple SoftTLBs, two design choices support multiple page sizes: varied-page-size SoftTLB and uniform-page-size SoftTLB. In varied-page-size SoftTLB, each SoftTLB entry can have different page sizes.

One possible implementation of the varied-page-size SoftTLB is to let the TLB entry have different page sizes in one SoftTLB. In such an implementation, the SoftTLB lookup routing requires at least two more instructions. One is an ALU instruction to calculate the address of the page-size information, and the other is a load instruction to load it. These extra instructions can introduce extra overhead of the SoftTLB lookup, and will hurt the performance of SoftTLB lookup, which is crucial for efficient VM execution.

In the uniform-page-size SoftTLB design, which is used in QEMU, we break down a larger guest page into smaller subpages of the same size. When accessing a larger page,

Fig. 4. Breakdown of SoftTLB flushes in emulation of Android benchmarks.

only the accessed subpage is stored in the SoftTLB. The uniform page size should use the minimum supported page size of the guest ISA. The advantage of this approach is that no extra overhead is introduced into the SoftTLB lookup routine. However, there are two potential shortcomings of this approach.

The first potential shortcoming is that more than one TLB entry is needed for a larger guest page. Each accessed subpage of the larger page takes up one entry in SoftTLB. If the SoftTLB is small, this may introduce misses since we may need to evict entries for other subpages of the larger page. We will deal with this problem in Section 4.2.

The second potential shortcoming is that, when a larger page is invalidated, all entries of subpages belonging to the larger guest page in SoftTLB must be invalidated. A simple solution to this problem is to flush the whole SoftTLB when a larger page is invalidated. We refer to this approach as *full flush*. Full flush is used in QEMU and may be sufficient if larger pages are not frequently used in guest systems. In the following section, we show that full flush is not adequate in that it results in too many SoftTLB flushes.

To investigate the efficiency of full flush, we profile the SoftTLB flushes in the ARM-to-x86_64 Android emulator. Section 5 contains detailed benchmark descriptions and experimental settings. We classify the causes of SoftTLB flushes into two parts. The first part is due to larger page invalidation. The second part is caused by a write to the page table base register when a context switch happens in the guest OS.

In Figure 4, the profiling results show that full flush causes 56% ~ 98% of SoftTLB flushes due to large-page invalidation. The high percentages of large-page invalidation are due to the default page size used by the Android emulator being 1KB, which is necessary to provide backward support for the minimum page size used in ARMv5. As a consequence, the most commonly used 4KB page is treated as a large page. Frequent SoftTLB flushes can affect VM performance in two ways. The first is the increased overhead of SoftTLB flush. The second is that it prohibits increasing the size of SoftTLB due to the flush overhead. As a result, it may lose the performance gains from larger SoftTLB, which can improve performance by increasing the hit rate of SoftTLB.

We propose an approach to efficiently handle large-page invalidation in the uniform-page-size SoftTLB design. The idea is to remember the used entries of one large page so that we can invalidate only these entries when the large page is invalidated, which we refer to as a *partial flush*. Partial flush works as follows. Three data are maintained for each accessed large page: the *starting address*, the *size*, and a *list of SoftTLB entries* occupied by its subpages. This information is called the large-page metadata.

Fig. 5. Number of SoftTLB flushes in the full-flush approach and our partial-flush approaches.

When inserting a subpage into the SoftTLB, we first search the metadata of the large page. We create the metadata for the accessed large page if it does not already exist. We then add the location of the newly added entry to the used list in the large-page metadata. When invalidating a large page, we look for the metadata of this page and flush all SoftTLB entries in the used list, instead of flushing the entire SoftTLB. We use a hash table to store these large-page metadata and this hash table needs to be flushed when SoftTLB is flushed.

Among different Android benchmarks, partial flush can eliminate 26% to 95% of unnecessary SoftTLB flushes due to large-page invalidation in the full-flush approach. Figure 5 compares the number of flushes in the full-flush approach and our partial-flush approach.

4.2. Supporting Dynamically Resizing Software TLB

4.2.1. SoftTLB Utilization. We further investigate the utilization of SoftTLB to assess potential performance improvement. For convenience, we partition the VM execution time into sessions by SoftTLB flushes. That is, an execution session starts immediately after a SoftTLB flush and ends just before the next SoftTLB flush. We profile the number of SoftTLB entries used for execution sessions. To obtain an accurate number of SoftTLB used entries, we prevent conflict by increasing the number of SoftTLB entries to 2^{16} . During profiling, at the end of each execution session, we count the number of used SoftTLB entries, which can be considered the session's working set.

We group those sessions by their working sets into working set group W_i . The working set group W_i contains sessions with between 2^{i-1} and 2^i entries. That is, if a session is in group W_i , then between 2^{i-1} and 2^i SoftTLB entries are used at the end of this session. We want to observe the working set distribution of each session. We profile the Android Boot and GeekBench benchmarks; the results are shown in Figure 6. For detailed experimental settings, refer to Section 5.

Figure 6 shows the percentage of working set groups. As shown in Figure 6, 37% and 33% of Boot and GeekBench sessions, respectively, use SoftTLB entries between 2^8 and 2^9 in group W_9 . Most of the other sessions are distributed among groups W_7 , W_8 , W_9 , and W_{10} in the Android Boot and GeekBench from 7% to 37%. Moreover, the results indicate that over 95% of execution sessions use no more than 2^{11} SoftTLB entries.

From these observations, we conclude that a single-size SoftTLB cannot fit all sessions. Ideally, the size of SoftTLB should be set to maximize the SoftTLB hit rate as well as to minimize the SoftTLB flush overhead. As we can see from Figure 6, although

Fig. 6. Distributions of execution sessions' working sets.

```

1 LOOKUP_sTLB:
2   mov  %rbp, %rdi
3   mov  %ebp, %esi
4   shr  $0x5, %rdi
5   and  $0xfffffc03, %esi
6   ; dedicate %r15d to hold the mask
7   ; value of the current SoftTLB size.
8   and  %r15d, %edi
9   lea  0x4c8(%r14, %rdi, 1), %rdi
10  cmp  (%rdi), %esi
11  mov  %ebp, %esi
12  jne  LOOKUP_MISS
13
14 LOOKUP_HIT:
15  add  0x10(%rdi), %rsi
16  mov  (%rsi), %ebp

```

Fig. 7. SoftTLB lookup routine for dynamically resizable SoftTLB.

a 2^{12} SoftTLB is sufficient to ensure a high hit rate for over 95% of execution sessions, we incur flush overhead for sessions with low SoftTLB utilization.

We propose a dynamically resizeable SoftTLB to maximize the hit rate and minimize the flush overhead. To minimize the performance impact for the lookup of a resizable SoftTLB, we reserve a host register to hold the value of the SoftTLB size information. Before entering the translation code cache, we need to load the current SoftTLB size value into the dedicated register. If we do not reserve this host register, we need two more instructions to get this value: one is to calculate the address of SoftTLB size relative to register r14, and the other is to load the value from memory to a host register. Figure 7 shows the modification of the lookup routine.

Reserving a dedicated host register does not affect the performance of translated code because, in QEMU, most translation blocks do not fully use all host registers because of the small granularity of its translation unit. That is, QEMU translates one guest basic block at a time, each of which usually contains less than 10 guest instructions.

We resize the SoftTLB for each guest process. Similar to Tong et al. [2015], we resize the SoftTLB based on utilization information, for which utilization is defined as the *number of used SoftTLB entries* over the *number of total SoftTLB entries*. However,

Fig. 8. Illustration of the relationship between SoftTLB utilization and when to resize the SoftTLB table for each process.

instead of keeping one system-wide utilization in Tong et al. [2015], we want to keep a set of utilization information for each running process inside the guest operating system. But a full-system emulator cannot obtain the process ID inside the guest operating system without the guest ISA support or kernel modification. Thus, we use the *page-table base address* as the pseudo-process ID of the running process in the guest operating system and assign a SoftTLB size to each page table. In the following discussion, *per-process* and *per-page-table* are interchangeable terms.

In our implementation, we use a hash table to store per-process utilization information. The hash table uses the base address of the guest-page table as the key. The size of the hash table is fixed by a reasonable number, say 4096 entries. If the number of co-existing processes in the guest system is larger than the hash-table size, it results in having only two processes sharing the same utilization information.

We update the per-page-table SoftTLB size at two places: at the end of the execution session and when a SoftTLB miss occurs (Figure 8 contains an illustration). When an execution session begins, the SoftTLB size of the current page table is loaded and the number of used SoftTLB entries is set to zero. During the execution session, we add missing pages into the SoftTLB, and the utilization of SoftTLB increases. The size of SoftTLB remains unchanged if the increased utilization is in the *stable range*. We will double the SoftTLB size immediately when the utilization is greater than the upper bound of the stable range after inserting a missed page, as in Process 2 in Figure 8. This is to avoid sudden bursts of SoftTLB misses during execution. If the utilization is smaller than the lower bound of the stable range at the end of the execution session, we reduce the SoftTLB size by half, as in Process 5 in Figure 8. We then update the new SoftTLB size of the page table and start a new execution session. Also, we set the upper bound and the lower bound of SoftTLB size to avoid SoftTLB being too large or too small.

5. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our optimizations implemented on the official QEMU v2.2.0 [QEMU 2015] emulator and the Android emulator of the AOSP [Android-Emulator 2015] version 5.0.1_r1. The Android emulator is a special

Table I. Workloads and Optimization List

Benchmarks	Description
SPEC CINT2006 (Train Inputs)	Standardized benchmark for testing single-process performance. Run on QEMU v2.2. Compiled by GCC 4.8.3 with O3 flags.
Android Booting	Boot the Android system.
Antutu	Test Android overall system performance of user experience, CPU, RAM, GPU, I/O.
Vellamo-Metal	
GeekBench	
Quadrant Professional Edition	Test Dalvik VM performance. Note: Skip 2D/3D tests.
Octane 2.0	Test javascript performance with Android browser.
Kraken 1.1	
Vellamo-Browser	Test Android browser performance.
Peacekeeper	
Browsermark	
	Optimization Lists
Abbreviation	Description
Baseline	Official QMEU/Android Emulator
CPBL	Cross-Page Block Linking + Physical iTLB
IBTC	Indirect Branch Target Caching + Physical iTLB
PF	Partial Flush for Large Page Invalidation
RS	Dynamically Resizing SoftTLB

Note: All benchmarks are ARM programs.

version of QEMU designed for Android kernel/application development. Both emulators are configured as ARM-to-X86_64 full-system emulators.

Both QEMU and the Android emulator are configured to have an ARM Cortex-A9 CPU with 2GB memory. We run Linaro Ubuntu 13.08 image [Linaro 2013] in QEMU, and Android v5.0.1_r1 images with Goldfish kernel 3.4.67 in the Android emulator. The host machine has an Intel Core i7-5930k 3.50GHz with 16GB RAM and the operating system is 64b Gentoo Linux 3.16.5. Detailed information of workloads is shown in Table I.

For performance comparison, we use the official QEMU v2.2.0 and the Android emulator v5.0.1_r1 as our performance baseline. We take the median of 3 runs as the final performance result for each benchmark. All reported performance results are normalized to the baseline. However, if the benchmark's score is timing information, such as in SPEC CINT2006 and Kraken v1.1, we report the reciprocal of the normalized number because higher performance figures are preferred. Optimizations are abbreviated as shown in Table I.

5.1. Experimental Results for Enabling Control Transfer Optimizations

5.1.1. Page Validation Approaches. We begin by comparing the page validation approaches for cross-page control transfers. We compare three approaches, the page walk, the virtual iTLB, and the physical iTLB described in Section 3. We use each approach to enable CPBL and IBTC.

For virtual iTLB, we set the size of the IBTC table to 2^{13} entries, which is the best balance between performance gained and flushing overhead. For physical iTLB, we set the size of the IBTC table to 2^{16} entries to obtain a higher hit rate since we do not need to flush the IBTC table when context switches in the physical iTLB approach. In both approaches, we set the iTLB table to 2^{12} entries.

The performance results of the SPEC CINT2006 and Android benchmarks are shown in Figures 9(a) and 9(b). The results show that we achieve an average speedup of 1.38X and 1.16X in physical iTLB, which outperforms the average speedup of 1.37X and 1.1X in virtual iTLB for SPEC CINT2006 and Android benchmarks, respectively. As expected, the page walk has the worst performance: only 0.64X and 0.93X of the baseline.

(a) SPEC CINT2006 Benchmarks.

(b) Android Benchmarks.

Fig. 9. Performance results for SPEC CINT2006 and Android benchmarks of page validation approaches for cross-page control transfers.

There is no performance difference between physical iTLB and virtual iTLB in SPEC CINT2006 benchmarks because there are not many TLB flushes caused by context switches. SPEC CINT2006 benchmarks are single-thread applications with only 1 active process during execution. Therefore, there are few context switches and TLB flushes. As a result, there is less flush overhead in the virtual iTLB approach.

On the other hand, Android benchmarks usually run multiple processes during execution. The physical iTLB could outperform the virtual iTLB in Android benchmarks because it avoids the flushing overhead and also benefits from higher hit rates due to larger table sizes. We use physical iTLB in the following experiments.

5.1.2. Performance Comparison of Lazy Check and Proactive Check. In this section, we compare the performance of *lazy check* and *proactive check* described in Section 3.2. Figure 10 shows the performance comparison results. The results show that lazy check achieves an average speedup of 1.38X and 1.16X, which outperforms the average speedup of 1.34X and 1.12X, respectively, in proactive check for SPEC CINT2006 and Android benchmarks.

5.1.3. Performance Breakdown of CPBL and IBTC. Figure 11 shows the performance breakdown of CPBL and IBTC with physical iTLB in the SPEC CINT2006 and Android

Fig. 10. Performance comparison of lazy check and proactive check for page validation. Note that the Y-axis starts at 100%.

benchmarks. For SPEC CINT2006, as shown in Figure 11(a), CPBL achieves an average speedup of 1.19X compared to the baseline performance, while CPBL + IBTC achieves an average speedup of 1.38X. 483.xalancbmk provides maximum performance of more than 2.24X speedup, while 429.mcf and 456.hmmr show no significant improvement at all with these two optimizations.

For the Android benchmarks, as shown in Figure 11(b), CPBL achieves an average speedup of 1.06X compared to the baseline performance, while CPBL + IBTC achieves an average speedup of 1.16X. Vellamo-Metal and Browsermark achieve an average speedup of 1.24X, the maximum performance improvement among the Android benchmarks. Six benchmarks achieve speedups exceeding 1.15X: Vellamo-Metal, Quadrant, Octance-2.0, Vellamo-Browser, Peacekeeper, and Browsermark. Overall, the Android benchmarks gain less improvement from these control transfer optimizations than single-process workloads because context switches happen more frequently in Android workloads.

We profile the frequencies and hit rates of CPBL and IBTC in SPEC CINT2006 and Android benchmarks in Table II to show the relationship between the speedup and the behavior of these optimizations. The profiled IBTC frequency is the number of executed indirect branches and the CPBL frequency is the number of executed direct branches or conditional branches across pages. The hit rate is the ratio of successful transfers due to IBTC or CPBL optimizations. Note that the profiled number comes from all processes in the guest operating system during the execution of one benchmark since it is difficult to profile the behavior of one particular process in the guest operating system. We minimize the interference of other processes by reducing unnecessary processes in the guest operating system.

As shown in Table II, benchmarks with greater speedup values also have high frequency and hit ratios; these include 400.perlbench, 445.gobmk, 458.sjeng, 471.omnetpp and 483.xalancbmk in SPEC CINT2006, and browser benchmarks (Vellamo-Browser, Peacekeeper, Browsermark) in the Android benchmarks. For IBTC, the benefit comes from the return instructions and indirect calls/branches. If an application frequently called shared library functions, it will gain performance improvement from IBTC because dynamically linked library functions are invoked via indirect calls/jumps through the procedure linkage table (PLT) and returned by return instructions. Both are indirect jumps that benefit from IBTC optimization. Also, if jump tables are frequently used in one benchmark, it will improve due to IBTC. For CPBL, cross-page direct branches

(a) SPEC CINT2006 Benchmarks.

(b) Android Benchmarks. Note that Y-Axis starts at 100%.

Fig. 11. Performance breakdown of control transfer optimizations.

usually happen in large applications in which the frequently called functions or jumped targets are not in the same code page.

The most improved benchmarks by CPBL and IBTC are dynamic script language interpreter (400.perlbench), artificial intelligence programs (445.gobmk and 458.sjeng), discrete event simulator (471.omnetpp), XML processor (483.xalancbmk), as well as Android browser benchmarks. These benchmarks all have frequent indirect branches and cross-page branches, and our iTLB approach is very effective.

5.2. Experimental Results on SoftTLB Optimizations

5.2.1. Experimental Results of Partial Flush. In this section, we evaluate the performance of the full-flush and partial-flush approaches described in Section 4.1 with different SoftTLB sizes (2^8 entries and 2^{12} entries). For partial flush, we set the large-page metadata hash table to 2^{12} entries.

In Figure 12, for each benchmark, the results depicted in the first and second bars have 2^8 SoftTLB entries, while the results in the third and fourth have 2^{12} SoftTLB entries. Full flush and partial flush achieve average speedups of 1.16X and 1.17X, respectively, with 2^8 SoftTLB entries, thus there is no performance difference between the two approaches. Due to search overhead and the flushes of the large-page metadata hash table, partial flush does not exhibit much performance gain even if the number of SoftTLB flushes is reduced.

Table II. Speedup and Statistics of CPBL and IBTC for the SPEC 2006 and Android Benchmarks

SPEC CINT2006					
Benchmarks	Speed up (CPBL+IBTC)	CPBL		IBTC	
		Frequency (Billions)	Hit Rate	Frequency (Billions)	Hit Rate
400.perlbench	1.70X	5.12B	99.83%	3.33B	99.89%
401.bzip2	1.13X	3.81B	99.33%	0.59B	98.11%
403.gcc	1.43X	0.13B	98.22%	0.09B	98.84%
429.mcf	1.06X	0.58B	99.49%	0.02B	94.33%
445.gobmk	1.66X	12.84B	99.96%	3.20B	99.92%
456.hmmmer	1.03X	0.45B	99.89%	0.17B	99.75%
458.sjeng	1.52X	15.49B	99.98%	9.96B	99.98%
462.libquantum	1.10X	0.10B	99.67%	0.01B	98.15%
464.h264ref	1.39X	4.41B	99.87%	11.54B	99.98%
471.omnetpp	1.60X	9.39B	99.97%	9.93B	99.99%
473.astar	1.12X	3.90B	99.83%	3.37B	99.94%
483.xalancbmk	2.24X	11.73B	99.78%	17.91B	99.92%
Android Benchmarks					
Android Boot	1.09X	0.14B	94.07%	0.21B	95.62%
Antutu	1.09X	4.79B	99.75%	7.20B	99.80%
Vellamo-Metal	1.23X	0.49B	99.31%	0.92B	98.26%
Geekbench	1.12X	1.60B	99.08%	0.75B	98.70%
Quadrant	1.17X	8.40B	99.96%	19.59B	99.96%
Octane-2.0	1.15X	1.54B	99.08%	1.76B	99.01%
Kraken-1.1	1.06X	2.49B	99.12%	2.77B	98.79%
Vellamo-Browser	1.21X	3.08B	98.25%	3.54B	97.28%
Peacekeeper	1.19X	1.53B	97.94%	2.14B	97.43%
Browsermark	1.24X	3.72B	98.15%	5.39B	97.59%

However, by reducing those unnecessary SoftTLB flushes, partial flush provides an opportunity to improve performance by enlarging the SoftTLB without incurring flushing overhead. As shown in Figure 12, after enlarging the SoftTLB size to 2^{12} entries, the partial-flush- 2^{12} achieves an average speedup of 1.89X and 1.33X, which outperforms the average speedup of 1.75X and 1.22X of full-flush- 2^{12} in SPEC CINT2006 and Android benchmarks, respectively.

The results for full-flush- 2^{12} also demonstrate the downside of full-flush in that flushes for large-page invalidation hurt performance when SoftTLB size is increased. Antutu and GeekBench achieve only 0.9X of the baseline performance in full-flush- 2^{12} . In summary, partial flush effectively reduces the number of SoftTLB flushes and allows us to use a larger SoftTLB size to raise average performance to 1.89X and 1.33X in SPEC CINT2006 and the Android benchmarks.

5.2.2. Overhead of Partial Flush. In this section, we show that partial flush has little overhead when large pages are rarely used. Figure 13 shows performance of partial flush in the QEMU IA32 emulator, for which only a few large pages are used in the OS kernel. As we can see, there is almost no performance difference between the partial-flush and full-flush approaches. Partial flush achieves 1.42X and full flush achieves 1.40X speedup when SoftTLB size is set to 2^{12} . Partial flush shows no overhead even when large pages are rarely used.

5.2.3. Performance of Dynamically Resizing SoftTLB. We evaluate the performance of dynamically resizing SoftTLB. The stable range of utilization is set to [25%, 50%]. From the previous section, we know that large table sizes achieve good performance with partial flush. We compare the performance of dynamic resizing with fixed-size SoftTLB.

(a) SPEC CINT2006 Benchmarks.

(b) Android Benchmarks.

Fig. 12. Performance comparison between Full-Flush (FF) and Partial-Flush (PF) with different SoftTLB sizes.

Fig. 13. Performance of partial flush in the QEMU IA32 emulator.

(a) SPEC CINT2006 Benchmarks.

(b) Android Benchmarks. Note that the Y-axis starts at 100%.

Fig. 14. Performance breakdown of SoftTLB optimizations.

The fixed-size SoftTLB is set to 2^{12} entries. The SoftTLB size of dynamic resizing ranges from 2^8 to 2^{16} , and the default size is 2^{11} . Performance results are shown in Figure 14.

For single-workload benchmarks, CPBL + IBTC + PF + RS achieves an average speedup of 1.98X from 1.40X of CPBL + IBTC + PF, while CPBL + IBTC + PF + 2^{12} achieves an average speedup of 1.91X. Ten out of 12 benchmarks show the same performance between dynamic resizing and fixed-size SoftTLB. RS only outperforms fixed-size SoftTLB in 429.mcf and 473.astar, where 429.mcf is a memory bound and cache-sensitive benchmark.

On the other hand, in Android benchmarks, CPBL + IBTC + PF + RS achieves an average speedup of 1.42X from 1.17X of CPBL + IBTC + PF, and CPBL + IBTC + PF + RS outperforms CPBL + IBTC + PF + 2^{12} from 1.33X to 1.42X. RS shows a significant improvement in GeekBench, Vellamo-Browser, and Octane-2.0 compared to fixed-size SoftTLB.

5.2.4. Performance of Per-Process and System-Wide SoftTLB Resizing. We further compare performance of our per-process resizable SoftTLB (Per-Process) with system-wide resizable SoftTLB (System-Wide) proposed in Tong et al. [2015]. The hash table of SoftTLB utilization information has 2^{12} entries for per-process resizable SoftTLB. The results are shown in Figure 15. As shown in Figure 15(a), there is no performance difference

(a) SPEC CINT2006 Benchmarks.

(b) Android Benchmarks.

Fig. 15. Performance comparison of Per-Process and System-Wide TLB resizing approaches. Note that the Y-axis starts at 100%.

between Per-Process and System-Wide approaches. This is because, in the SPEC CINT2006 benchmarks, there is only one active process in the system during execution such that Per-Process does not behave differently from System-Wide.

On the other hand, Per-Process outperforms System-Wide in the Android benchmarks, as shown in Figure 15(b). On average, Per-Process achieves a 1.43X speedup while System-Wide achieves a 1.38X speedup, on average. Per-Process outperforms System-Wide because Android benchmarks tend to have multiple processes running concurrently during execution, especially for browser benchmarks, such as Octane-2.0, Kraken-1.1, Vellamo-Browser, Peacekeeper, and Browsermark.

Per-Process could resize SoftTLB according to separate SoftTLB utilization information of each process, and a complex execution environment in which many processes are running concurrently could allow Per-Process to outperform System-Wide.

5.2.5. Performance of IA32 and AArch64. In this section, we evaluate the performance of our control transfer and memory optimizations on different guest CPUs. We use gcc 4.8.3 to compile SPEC CINT2006 for Intel's IA32 version and ARM AArch64 64b version with -O3 flag. Both QEMU emulators are configured to have 2GB RAM, and the QEMU AArch64 emulator is configured to have an ARMv8 Cortex-A57. We show the

Fig. 16. Performance of CPBL+IBTC+PF+RS optimizations for IA32 and AArch64 full-system emulators.

speedup of CPBL+IBTC+PF+RS optimizations compared to the official QEMU v2.2.0 in Figure 16.

For the QEMU IA32 emulator, the average speedup is 144% and the performance ranges from 108% to 265%. For the QEMU AArch64 emulator, the average speedup is 140% and the performance ranges from 108% to 185%. Although the performance gain is not as significant as that in the ARM emulator, our optimizations achieve 144% and 140% speedup for IA32 system emulators and AArch64 system emulators, respectively.

6. CONCLUSION

We propose effective optimizations to improve the performance of a cross-ISA system-level emulator, along with efficient approaches to check page validity with the software instruction TLB to enable classic control transfer optimizations of DBT in system-level emulations. By enabling two classic dynamic binary optimizations (IBTC and CPBL/chaining) average performance speedups of 1.38X and 1.16X are achieved on SPEC CINT2006 and popular Android benchmarks, respectively. The results are promising because our approaches allow for the implementation of dynamic binary optimizations, such as trace optimizations, to further improve cross-ISA system-mode emulation performance.

The second group of proposed optimizations focuses on improving the performance of memory virtualization of cross-ISA VMs by improving the efficiency of the SoftTLB. We reduce the overhead of unnecessary SoftTLB flushes resulting from the full-flush approach for large-page invalidation. The proposed partial-flush approach can effectively reduce unnecessary SoftTLB flushes, and can also be used to avoid unnecessary page walks when SoftTLB misses.

We further improve performance by adaptively resizing SoftTLB through per-page-table SoftTLB profiling. In this way, we can resize SoftTLB according to the current utilization of SoftTLB. This can both improve the SoftTLB hit rate and reduce flushing overhead.

Our experimental results on the ARM-to-X86_64 QEMU and ARM Android emulators show that our optimizations improve SPEC CINT2006 integer benchmarks by an average of 1.98X. Our optimizations also achieve an average speedup of 1.44X and 1.40X for IA32-to-X86-64 QEMU and AArch64-to-X86-64 QEMU on SPEC CINT2006. For Android benchmarks, we achieve an average speedup of 1.43X. The results show that our optimizations improve performance on system-level emulators running real applications.

We have made our implementation available for download at [GitHub.com](https://github.com/tkhsu/quick-android-emulator). The optimized Android emulator is available at <https://github.com/tkhsu/quick-android-emulator>. The optimized QEMU is at <https://github.com/tkhsu/quick-qemu>.

6.1. Future Works

This article shows that control transfer optimizations do improve performance for a wide range of applications. We expect more improvement from other dynamic binary optimizations, such as trace optimizations in system-level emulations.

REFERENCES

- Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, 2–13.
- AMIDuOS. 2015. AMIDuOS. Retrieved November 2, 2015 from <http://www.amiduos.com>.
- Android-Emulator. 2015. Homepage. Retrieved November 2, 2015 from <https://android.googlesource.com/platform/external/qemu.git>.
- ARM. 2007. Cortex-A9 Technical Reference Manual. Retrieved November 2, 2015 from <http://infocenter.arm.com/help/index.jsp>.
- Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*. ACM, New York, NY, 1–12.
- L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Yun Wang, and Y. Zemach. 2003. IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. (MICRO-36)*. 191–201. DOI: <http://dx.doi.org/10.1109/MICRO.2003.1253195>
- Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, NY, 164–177.
- Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. 2010. SPUR: A trace-based JIT compiler for CIL. *SIGPLAN Notices* 45, 10, 708–725. DOI: <http://dx.doi.org/10.1145/1932682.1869517>
- Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 41–46.
- BlueStacks. 2015. BlueStacks. (2015). Retrieved November 2, 2015 from <http://www.bluestacks.com>.
- I. Bohm, T. J. K. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham. 2011. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of PLDI*.
- D. Bruening, T. Garnett, and S. Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*. 265–275.
- Prashanth P. Bungale and Chi-Keung Luk. 2007. PinOS: A programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE'07)*. ACM, New York, NY, 137–147. DOI: <http://dx.doi.org/10.1145/1254810.1254830>
- Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. 2014. Efficient memory virtualization for cross-ISA system mode emulation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'14)*. ACM, New York, NY, 117–128. DOI: <http://dx.doi.org/10.1145/2576195.2576201>
- Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. 1998. FX!32: A profile-directed binary translator. *IEEE Micro* 18, 2, 56–64. DOI: <http://dx.doi.org/10.1109/40.671403>
- Bob Cmelik and David Keppel. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'94)*. ACM, New York, NY, 128–137. DOI: <http://dx.doi.org/10.1145/183018.183032>
- James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The transmeta code morphing™ software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*. IEEE Computer Society, Washington, DC, 15–24.
- Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. 2001. Dynamic binary translation and optimization. *IEEE Transactions on Computers* 50, 6, 529–548.

- Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Notices* 44, 6, 465–478. DOI: <http://dx.doi.org/10.1145/1543135.1542528>
- Ding-Yong Hong, Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, Jan-Jan Wu, Pen-Chung Yew, and Wei-Chung Hsu. 2012. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the 10th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'12)*.
- Chun-Chen Hsu, Pangfeng Liu, Jan-Jan Wu, Pen-Chung Yew, Ding-Yong Hong, Wei-Chung Hsu, and Chien-Min Wang. 2013. Improving dynamic binary optimization through early-exit guided code region formation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'13)*. ACM, New York, NY, 23–32. DOI: <http://dx.doi.org/10.1145/2451512.2451519>
- H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. 2011. A trace-based Java JIT compiler retrofitted from a method-based compiler. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*. 246–256.
- Intel. 2015. Intel Developer Manuals. Retrieved November 2, 2015 from <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>.
- A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. 2007. kvm: The Linux virtual machine monitor. In *The 2007 Ottawa Linux Symposium (OLS'07)*. 225–230.
- Linaro. 2013. Linaro Versatile Express 13.08 Release. Retrieved from <http://releases.linaro.org/openembedded/vexpress-lsk/latest>.
- Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, 190–200.
- Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hällberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2, 50–58. DOI: <http://dx.doi.org/10.1109/2.982916>
- N. Nethercote and J. Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of PLDI*. 89–100.
- QEMU. 2015. Retrieved November 2015 from <http://qemu.org>.
- K. Scott, N. Kumar, B. R. Childers, J. W. Davidson, and M. L. Soffa. 2004. Overhead reduction techniques for software dynamic translation. In *Proceedings of IPDPS*. 200–207.
- James E. Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufman, Burlington, MA.
- Xin Tong, Toshihiko Koju, Motohiro Kawahito, and Andreas Moshovos. 2015. Optimizing memory translation emulation in full system emulators. *ACM Transactions on Architecture and Code Optimization* 11, 4, Article 60, 24 pages.
- Emmett Witchel and Mendel Rosenblum. 1996. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'96)*. ACM, New York, NY, 68–79.
- Qin Zhao, Derek Bruening, and Saman Amarasinghe. 2010. Umbra: Efficient and scalable memory shadowing. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*. ACM, New York, NY, 22–31.
- Cindy Zheng and Carol Thompson. 2000. PA-RISC to IA-64: Transparent execution, no recompilation. *Computer* 33, 3 (2000), 47–52.

Received May 2015; revised October 2015; accepted October 2015