中央研究院
資訊科學研究所
Institute of Information Science, Academia Sinica • Taipei, Taiwan, ROC

TR-IIS-15-002

Optimizing Control Transfer and Memory Virtualization in Full System Emulators

Chun-Chen Hsu, Ding-Yong Hong, Cheng-Yi Chou, Jan-Jan Wu, Wei-Chung Hsu, and Pangfeng Liu

Feb. 04, 2015 || Technical Report No. TR-IIS-15-002

http://www.iis.sinica.edu.tw/page/library/TechReport/tr2015/tr15.html

Optimizing Control Transfer and Memory Virtualization in Full System Emulators

Chun-Chen Hsu[‡], Ding-Yong Hong[§], Cheng-Yi Chou[§],
Jan-Jan Wu[§], Wei-Chung Hsu[‡], and Pangfeng Liu[‡]

[‡]Department of Computer Science and Information Engineering, National Taiwan University
{d95006,pangfeng,hsuwc}@csie.ntu.edu.tw
[§]Institute of Information Science, Academia Sinica
{dyhong,maxchou,wuj}@iis.sinica.edu.tw

Abstract

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 1.92X for ARM-to-X86-64 QEMU running SPEC CINT2006 benchmarks with train inputs. 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 1.42X for the Android emulator running these applications.

1 Introduction

Cross-ISA virtualization techniques allow programs compiled for one instruction set architecture (ISA) (e.g., Intel IA-32) to be run on platforms based on a different ISA (e.g., IA-64). A dynamic binary translator and a binary optimizer are used to translate a guest instruction into a sequence of host instructions with a different ISA. Some of these virtualization systems could emulate the entire ISA including privileged instructions. Hence, an entire guest OS could be booted up and run virtually on a host OS with a completely different ISA. This type of technology is used in popular functional simulators such as Simics [25] and QEMU [10]. 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 paper focuses on cross-ISA system-level emulation, i.e., the guest and the host belong to different instruction set architectures, using dynamic binary translation (DBT) techniques. Improving the performance of cross-ISA 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 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 which is managed by the guest OS, and the guest physical address is allocated and assigned by the host OS. So 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 VM 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 virtual machines must be rethought and redesigned.

In this paper, we investigate two optimizations related to memory access: (1) branch optimization, including

block linking [16] and indirect branch translation cache (IBTC) [28], and (2) software-based translation lookaside buffer (software TLB) to improve 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 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. To the best of our knowledge, this is the first work using software-based approach to optimize cross-page branches.

For memory translation optimization, to speed up the aforementioned multi-level 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. [14] 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 [27] emulator and the Android emulator of the Android Open Source Project (AOSP) [5] 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 1.92X 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. For the Android emulator, our optimization achieves an average 1.43X speedup against the unmodified Android emulator for Android benchmarks. We also put our works in the GitHub.com for interested readers to download. Please refer to Section 6.

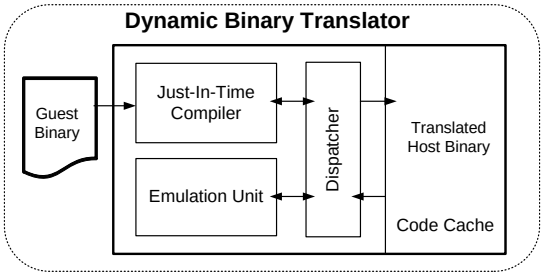


Figure 1: A general framework of a dynamic binary translator.

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

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. User-mode DBTs has been widely used for runtime profiling [24, 26, 30, 32], transparent performance optimization [6, 13] and migration of legacy code [7, 15, 33]. 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, e.g. 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 [23] and Xen [8], has proposed hardware-assisted or paravirtualized approaches to trap and emulate such sensitive instructions.

In contrast, DBTs, such as VMWare [4] and QEMU [10], 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 [16, 29], trace optimization [6, 12, 24, 21, 20], and indirect branch target caching [28]) have been shown to be effective and have been widely adapted to high level language virtual machines [22], or dynamic scripting languages [19, 9]. However, these optimizations are either not used in dynamic binary translation in system mode emulation, or only adopted conditionally. For example, QEMU [10], a retargetable DBT system supporting many guest and host ISAs, only applies the optimization of block linking for branches among the same guest page. In another example, Böhm et al. [11] built traces with the limitation that blocks from a single trace cannot span across page boundaries.

Existing solutions for cross-page control transfers come mostly from hardware based solutions. IBM's DAISY system [18] runs a PowerPC guest on a VLIW machine. To the best of our knowledge, DAISY is the first and the only such system that explicitly mentions the page boundary problem for cross-page control transfer optimizations. DAISY formed several basic blocks into a tree-region which 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 [17] forms traces that contains 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.

Unlike their frameworks, we propose a pure software-based approach (the details are discussed in Section 3) to validate cross-page branches. With this approach, we can enable control transfer optimizations developed in user-mode DBT in system mode DBT.

Memory Virtualization

Memory virtualization optimization have been well studied and developed in same-ISA virtualization. For example, before hardware-assisted virtualization was supported in AMD64 CPUs, Xen and VMWare [4] 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 para-virtualization, 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 [23], for example, uses this hardware-assisted memory virtualization mechanism.

Without specially-designed hardware, memory virtualization in cross-ISA virtualization can only use software-based approaches. QEMU and Simics use such

software-based approaches. Tong el. [31] is the first work found in the literature that focuses on optimizing of software-based memory virtualization. They propose several techniques to increase SoftTLB hit rates as well as to reduce the overhead of SoftTLB maintenance. 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 paper, we deal with the cross-page problem of control transfer optimizations that Tong et al. [31] 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. The per-page-table SoftTLB utilization provides more finer granularity than the system-wide SoftTLB utilization.

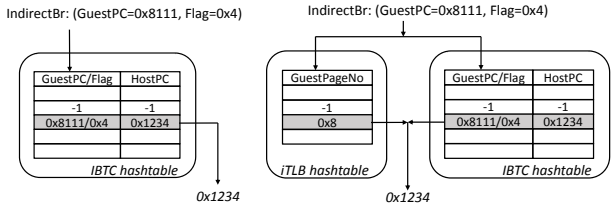
3 Enabling Control Transfer Optimizations

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 translator and code cache, common control transfer optimizations such as *block linking* [16] and *indirect branch translation caching (IBTC)* [28] are respectively used 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 same 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 destination code 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 the wrong code 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)



(a) User-Mode DBT (b) Full-System DBT

Figure 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.

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 branched page is needed because the page property 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 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 is within the same guest page or not. One naive approach to validate 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 hardware TLB, which is a CPU cache to improve virtual address translation speed. Unlike hardware TLB where virtual-to-physical address mappings are recorded for both data and code pages, our virtual iTLB is a simplified hash table and it 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 the guest page size is 4 KBytes, 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.

In the translation of direct branch across two pages, the chaining scheme in user-mode DBTs that emits the host's direct/indirect jump instruction is inadequate and is not used. Instead, we apply the same hash table technique used in IBTC to solve the problem. Although we can redirect such direct branches to look up the same IBTC hash table in our system, we create another hash table used only for the *cross-page block linking* (CPBL). The CPBL hash table caches guest blocks and their translated code addresses to which those guest blocks have recently jumped through cross-page direct branches. On a CPBL lookup hit, the same virtual iTLB is looked up against the branch address to determine its validity. Leaving code cache only occurs when the hash table lookup incurs a miss.

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 the target code page is valid when virtual iTLB hits, but we cannot ensure that the mapping the target code page remain the same. Therefore, when a context switch happens in the guest operating system, we need to flush the virtual iTLB, the IBTC, and CPBL tables. Also, invalidating a page by the guest also results in invalidation to associated entries of the hash tables. For this situation, we search for each entry of the hash tables and flush the entry if it belongs to the invalidated guest page.

3.2 Optimization with Physical iTLB

This flushing of three hash tables limits the performance improvement of IBTC and CPBL. Flushing itself poses extra overhead, so the table size should be restricted to ensure performance is not affected. However, limiting the size of IBTC and CPBL tables limits their 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/CPBL tables contain 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 occurs, we need only to flush the physical iTLB. The IBTC/CPBL tables do not need to be flushed. Where the mapping of the guest target address changes, the addresses in the IBTC table will not match the physical iTLB, and the cross-page branch will not be taken. Compared to virtual iTLB, the shortcoming of this approach is it requires extra comparison for physical addresses, but it poses advantages in that (1) we can avoid the flushing overhead induced from IBTC tables; (2) the entries in IBTC/CPBL tables can survive across context switches; and (3) we can enlarge IBTC tables to improve performance by increasing the hit rate.

4 Optimizations for Software Translation Lookaside Buffer

In this section, we propose two optimizations to improve the efficiency of the software translation lookaside buffer (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.

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 [2] and ARM [1]. To run a guest system with multiple page sizes, VM must also support multiple page sizes in its software translation lookaside buffer.

```
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
```

Figure 3: Routine of SoftTLB Lookup

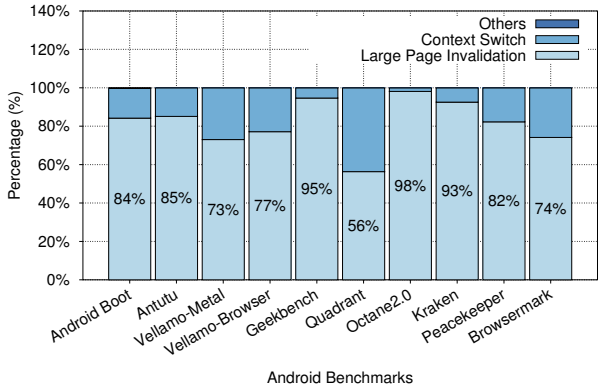


Figure 4: Breakdown of SoftTLB flushes in emulation of Android benchmarks

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 lookup the SoftTLB.

In 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 size 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 lookup a guest virtual address, we have to first decide which SoftTLB should be used. Tong el. [31] 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 sub-pages of the same size. When accessing a larger page, only the accessed sub-page 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. Please refer to Section 5 for detail benchmarks description and experimental settings. We classify the causes of SoftTLB flushes into three 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. The third part

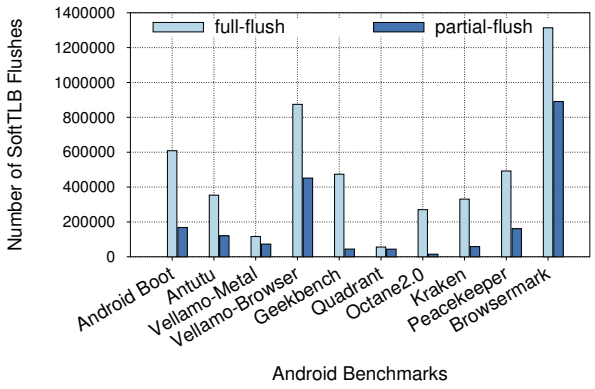


Figure 5: Number of SoftTLB flushes in the full-flush approach and our partial-flush approaches.

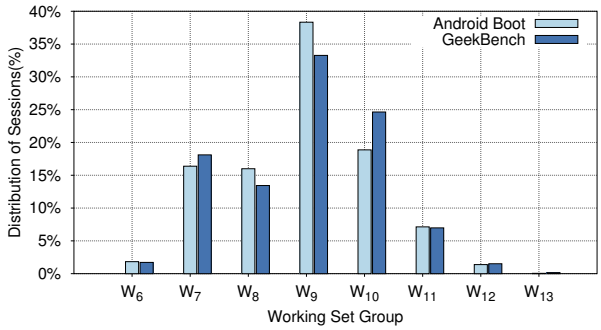


Figure 6: Distributions of execution sessions' working sets.

includes other miscellaneous causes, such as guest system instructions that flush TLB.

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 is because the default page size used by the Android emulator is 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 *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 sub-pages. This information is called the large page metadata.

When inserting a sub-page 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 find its metadata and just flush all SoftTLB entries in the used list, instead of flushing the whole SoftTLB. We use a hash table to store these large page metadata. Also, because this hash table needs to be flushed with SoftTLB, individual hash table can also reduce the flush overhead.

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 The 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 enlarging 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 and the results are shown in Figure 6. For detailed experimental settings please refer to Section 5.

Figure 6 shows the percentage of working set groups. As shown in the Figure, respectively 37% and 33% of Boot and GeekBench sessions 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%. Th Moreover, the results indicate that over 95% of exe-

```
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
```

Figure 7: SoftTLB lookup routine for dynamically resizable SoftTLB.

cution sessions use no more than 2^{11} SoftTLB entries.

From these observations, we conclude that a single size SoftTLB cannot fit to 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 the Figure 6, although 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 re-sizeable 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. Figure 7 shows the modification of the lookup routine.

Line 7 in Figure 7 shows that reserving a dedicated host register may affect the performance of translated code since we lost a free host register in register allocation. However, 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.

The next question is when to resize the SoftTLB. Similar to [31], we resize SoftTLB based on utilization information, where utilization is defined as #used SoftTLB entries/#total SoftTLB entries. This requires profiling the utilization of SoftTLB during VM execution. Ideally, instead of keeping one system-wide set of utilization information in [31], we would keep a set of utilization information for each running process inside the guest operating system. But a full system emulator cannot get process ID inside the guest operating system without the guest ISA support or kernel modification. Therefore, we use the page table base address as the pseudo process ID of the running process in the guest operating system. For each page table, we keep a set of utilization information and a set of SoftTLB size information, referred to the page table metadata.

Table 1: 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 Vellamo-Metal GeekBench	Test Android overall system performance of user experience, CPU, RAM, GPU, I/O.
Quadrant Professional Edition	Test Dalvik VM performance. Note: Skip 2D/3D tests.
Octane 2.0 Karern 1.1	Test javascript performance with Android browser.
Vellamo-Browser Peacekeeper Browsermark	Test Android browser performance.

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.

We then update the per-page-table metadata at two places: at the end of the execution session and when a SoftTLB miss occurs. When the execution is at the end of the session, we compute the SoftTLB utilization and update its SoftTLB size information for this page table. The size information stays unchanged when the utilization is in the *stable range of utilization*. For example, we can set the stable range of utilization to [25%, 50%]. Thus, if the utilization is smaller than the lower bound of the stable range, we reduce the size information. Similarly, if the utilization is larger than the upper bound of the stable range, we increase the size information. Otherwise, the size information stays unchanged. Then, just before a new execution session begins, we re-size the SoftTLB according to the current page table metadata. To avoid sudden bursts of SoftTLB misses, we enlarge the SoftTLB in the event of a SoftTLB miss. When a SoftTLB miss occurs, we enlarge it if the utilization is over 50%.

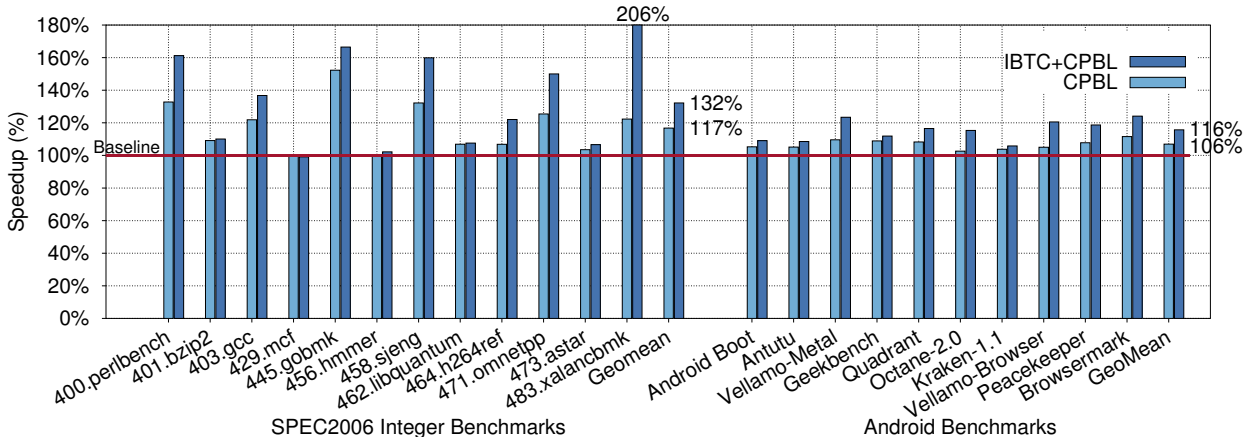


Figure 9: Performance breakdown of control transfer optimizations.

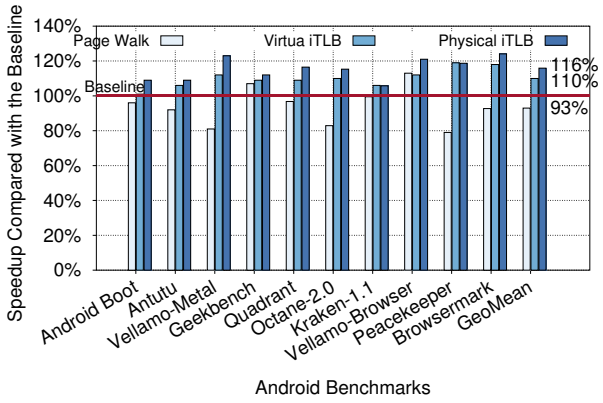


Figure 8: Performance results for Android benchmarks of page validation approaches for cross-page control transfers.

5 Experimental Results

In this section, we evaluate the performance of our optimizations implemented on the official QEMU v2.2.0 [27] emulator and the Android emulator of the Android Open Source Project (AOSP) [5] version 5.0.1_r1. The Android emulator is a special version of QEMU designed for Android kernel/application development. Both emulators are configured as ARM-to-X86_64 full system emulators.

Both QEMU and Android emulator are configured to have a ARM Cortex-A9 CPU with 2GB memory. We run Linaro Ubuntu 13.08 image [3] in QEMU, and Android v5.0.1_r1 images with Goldfish kernel 3.4.67 in Android emulator. The host machine has an Intel Core i7-5930k 3.50 GHz with 16GB RAM and the operating system is 64-bit Gentoo Linux 3.16.5. Detailed information of workloads are shown in Table 1.

For performance comparison, we use official QEMU v2.2.0 and 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 baseline. But if the benchmark's score is timing information, such as SPEC CINT2006 and Karern v1.1, we report the reciprocal of the normalized number because higher performance figures are preferred. Optimizations are abbreviated as shown in Table 1.

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 use two hash tables with 2^{13} entries for IBTC and CPBL tables, which is the best balance between performance gained and flushing overhead. Since the IBTC and CPBL tables do need to flush when context switches, we set the size of physical iTLB to 2^{16} entries to obtain a higher hit rate. In both approaches, we set the iTLB table to 2^{12} entries.

Due to page limits, we only show the performance results of Android benchmarks in Figure 8. The results show that we achieve an average speedup of 1.16X in physical iTLB, which outperforms the average speedup of 1.1X in virtual iTLB. As expected, the page-walk has the worst performance: only 0.93X of baseline. The physical iTLB outperforms the virtual iTLB because it avoids the flushing overhead and also benefits from higher hit rates due to larger table sizes. We will use physical iTLB in the following experiments.

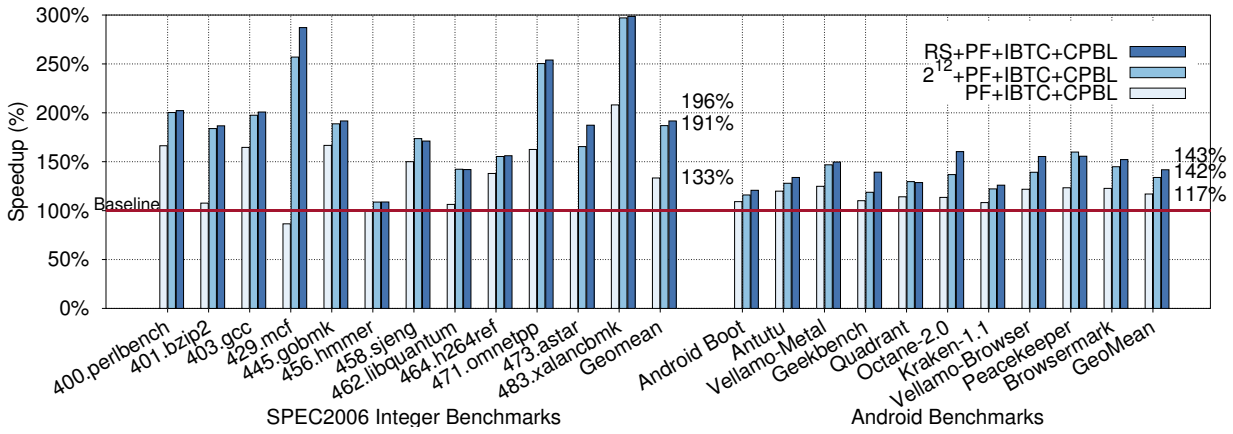


Figure 10: Performance breakdown of SoftTLB optimizations.

5.1.2 Performance Breakdown of CPBL and IBTC

Figure 9 shows the performance breakdown of CPBL and IBTC with physical iTLB. In single process workloads, CPBL achieves an average speedup of 1.17X compared to the baseline performance, while CPBL + IBTC achieves an average speedup of 1.32X. 483.Xalancbmk provides maximum performance of more than 2X speedup, while 429.mcf and 456.hmmer show no improvement at all with these two optimizations.

Performance is related to the frequency of cross-page transfers. Cross-page block links usually happen in library calls since library functions are likely to be in different code pages from the program text section. So if the program contains lots of library calls, it may benefit from CPBL optimizations. Similarly, if programs contains more indirect branches, such as return instructions, they benefit more from the IBTC optimizations.

In the Android benchmarks, 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 Android benchmarks which, overall, gain less improvement from these control transfer optimizations than single process workloads because context switches happen more frequently in Android workloads.

5.2 Experimental Results on SoftTLB Optimizations

5.2.1 Experimental Results of Partial-Flush

In this subsection, 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. Again, due to space limitations, we only show the performance of Android benchmarks.

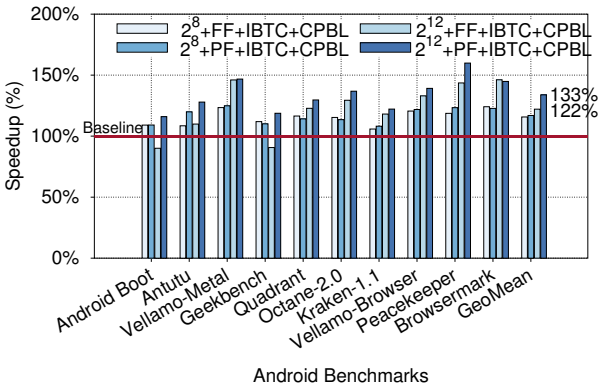


Figure 11: Performance comparison between Full-Flush (FF) and Partial-Flush (PF) on Android Benchmarks different SoftTLB sizes.

In Figure 11, for each benchmark, the results depicted in the first and the second bars have 2^8 SoftTLB entries, while the results in the third and the fourth have 2^{12} SoftTLB entries. Full-flush and partial-flush respectively achieve average speedups of 1.16X and 1.17X 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.

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 11, after enlarging the SoftTLB size to 2^{12} entries, the partial-flush-2^{12} achieves an average speedup of 1.33X, which outper-

forms the average speedup of 1.22X of full-flush-2^{12}.

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 only achieve 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 1.33X.

5.2.2 Performance of Dynamically Resizing SoftTLB

We evaluate the performance of dynamically resizing SoftTLB. The stable range of utilization is set to [25%, 50%]. The table size ranges from 2^6 to 2^{14} entries. 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. The fixed-size SoftTLB is set to 2^{12} entries. Performance results are shown in Figure 10.

For single workload benchmarks, CPBL + IBTC + PF + RS achieves an average speedup of 1.91X from 1.33X of CPBL + IBTC + PF, while CPBL + IBTC + PF + 2^{12} achieves an average speedup of 1.87X. 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 Octance-2.0 compared to fixed-size SoftTLB.

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 dynamic binary translation in system level emulations. By enabling two classic dynamic binary optimizations (indirect branch target caching and cross-page block linking/chaining) average performance speedups of 1.32X and 1.16X speedup are respectively achieved on SPEC CPU 2006 integer benchmarks and popular Android benchmarks. 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 focus on improving the performance of memory virtualization of cross-ISA virtual machines by improving the efficiency of the software translation lookaside buffer (TLB). 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 ARM-to-X86_64 QEMU and an ARM Android emulator show that our optimizations improve SPEC CINT2006 integer benchmarks by an average of 1.92X. For Android benchmarks, we achieve an average speedup of 1.42X. The results show that our optimizations improve performance on system level emulators running real applications.

We have made our implementation available for download at the GitHub.com. 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 paper shows that control transfers 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. Also, further experiments should be conducted on other architectures, such as emulating X64_64 on ARM64, to compare optimization improvement.

Acknowledgment

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

References

[1] Cortex-a9 technical reference manual. http://infocenter.arm.com/help/index.jsp.

[2] Intel developer manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[3] Linaro versatile express 13.08 release. http://releases.linaro.org/13.08/ubuntu/vexpress.

[4] ADAMS, K., AND AGESEN, O. 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* (New York, NY, USA, 2006), ASPLOS XII, ACM, pp. 2–13.

[5] ANDROID.GOOGLESOURCE.COM. Android qemu emulator. https://android.googlesource.com/platform/external/qemu.git.

[6] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (New York, NY, USA, 2000), ACM, pp. 1–12.

[7] BARAZ, L., DEVOR, T., ETZION, O., GOLDENBERG, S., SKALETSKY, A., WANG, Y., AND ZEMACH, Y. Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium-based systems. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* (Dec. 2003), pp. 191–201.

[8] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM, pp. 164–177.

[9] BEBENITA, M., BRANDNER, F., FAHNDRICH, M., LOGOZZO, F., SCHULTE, W., TILLMANN, N., AND VENTER, H. Spur: a trace-based jit compiler for cil. *SIGPLAN Not. 45* (October 2010), 708–725.

[10] BELLARD, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 41–46.

[11] BOHM, I., VON KOCH, T. E., KYLE, S., FRANKE, B., AND TOPHAM, N. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proc. PLDI* (2011).

[12] BRUENING, D. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, Sep 2004.

[13] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization* (2003), pp. 265–275.

[14] CHANG, C.-J., WU, J.-J., HSU, W.-C., LIU, P., AND YEW, P.-C. Efficient memory virtualization for cross-isa system mode emulation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2014), VEE '14, ACM, pp. 117–128.

[15] CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., AND YATES, J. Fx!32: A profile-directed binary translator. *IEEE Micro 18*, 2 (1998), 56–64.

[16] CMELIK, B., AND KEPPEL, D. Shade: a fast instruction-set simulator for execution profiling. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems* (New York, NY, USA, 1994), ACM, pp. 128–137.

[17] DEHNERT, J. C., GRANT, B. K., BANNING, J. P., JOHNSON, R., KISTLER, T., KLAIBER, A., AND MATTSON, J. The transmeta code morphing^{TM}software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the international symposium on Code generation and optimization* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 15–24.

[18] EBCIOGLU, K., ALTMAN, E., GSCHWIND, M., AND SATHAYE, S. Dynamic binary translation and optimization. *IEEE Trans. Comput. 50*, 6 (2001), 529–548.

[19] GAL, A., EICH, B., SHAVER, M., ANDERSON, D., MANDELIN, D., HAGHIGHAT, M. R., KAPLAN, B., HOARE, G., ZBARSKY, B., ORENDORFF, J., RUDERMAN, J., SMITH, E. W., REITMAIER, R., BEBENITA, M., CHANG, M., AND FRANZ, M. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not. 44* (June 2009), 465–478.

[20] HONG, D.-Y., HSU, C.-C., LIU, P., WANG, C.-M., WU, J.-J., , YEW, P.-C., AND HSU, W.-C. Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores. In *CGO '12: Proceedings of the 10th annual IEEE/ACM international symposium on Code generation and optimization* (2012).

[21] HSU, C.-C., LIU, P., WU, J.-J., YEW, P.-C., HONG, D.-Y., HSU, W.-C., AND WANG, C.-M. 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* (New York, NY, USA, 2013), VEE '13, ACM, pp. 23–32.

[22] INOUE, H., HAYASHIZAKI, H., WU, P., AND NAKATANI, T. A trace-based java jit compiler retrofitted from a method-based compiler. In *IEEE/ACM International Symposium on Code Generation and Optimization* (2011), pp. 246–256.

[23] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium* (July 2007), pp. 225–230.

[24] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM, pp. 190–200.

[25] MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HÅLLBERG, G., HÖGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. Simics: A full system simulation platform. *Computer 35*, 2 (Feb. 2002), 50–58.

[26] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. PLDI* (2007), pp. 89–100.

[27] QEMU.ORG. QEMU. http://qemu.org.

[28] SCOTT, K., KUMAR, N., CHILDERS, B. R., DAVIDSON, J. W., AND SOFFA, M. L. Overhead reduction techniques for software dynamic translation. In *Proc. IPDPS* (2004), pp. 200–207.

[29] SMITH, J. E., AND NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufman, 2005.

[30] SRIDHAR, S., SHAPIRO, J. S., NORTHUP, E., AND BUNGALE, P. P. Hdtrans: an open source, low-level dynamic instrumentation system. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments* (New York, NY, USA, 2006), ACM, pp. 175–185.

[31] TONG, X., KOJU, T., KAWAHITO, M., AND MOSHOVOS, A. Optimizing memory translation emulation in full system emulators. *ACM Trans. Archit. Code Optim. 11*, 4 (Jan. 2015), 60:1–60:24.

[32] ZHAO, Q., BRUENING, D., AND AMARASINGHE, S. Umbra: efficient and scalable memory shadowing. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (New York, NY, USA, 2010), ACM, pp. 22–31.

[33] ZHENG, C., AND THOMPSON, C. Pa-risc to ia-64: Transparent execution, no recompilation. *Computer 33*, 3 (2000), 47–52.