BTMMU: An Efficient and Versatile Cross-ISA Memory Virtualization

Kele Huang
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
kele.hwang@gmail.com

Fuxin Zhang*
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
fxzhang@ict.ac.cn

Cun Li
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
cun.jia.li@gmail.com

Gen Niu
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
niugen@loongson.cn

Junrong Wu
School of Computer Science, Beijing Institute of Technology
China
3120181048@bit.edu.cn

Tianyi Liu
Department of Computer Science, University of Texas at San Antonio
USA
tianyi.liu@utsa.edu

Abstract

Full system dynamic binary translation (DBT) has many important applications, but it is typically much slower than the native host. One major overhead in full system DBT comes from cross-ISA memory virtualization, where multi-level memory address translation is needed to map guest virtual address into host physical address. Like the SoftMMU used in the popular open-source emulator QEMU, software-based memory virtualization solutions are not efficient. Meanwhile, mature techniques for same-ISA virtualization such as shadow page table or second level address translation are not directly applicable due to cross-ISA difficulties. Some previous studies achieved significant speedup by utilizing existing hardware (TLB or virtualization hardware) of the host. However, since the hardware is not designed with cross-ISA in mind, those solutions had some limitations that were hard to overcome. Most of them only supported guests with smaller virtual address space than the host. Some supported only guests with the same page size. And some did not support privileged memory accesses.

This paper proposes a new solution named BTMMU (Binary Translation Memory Management Unit). BTMMU composes of a low-cost hardware extension of host MMU, a kernel module and a patched QEMU version. BTMMU is able to solve most known limitations of previous hardware-assisted solutions and thus versatile enough for real deployments. Meanwhile, BTMMU achieves high efficiency by directly accessing guest address space, implementing shadow page table in kernel module, utilizing dedicated entrance for guest-related MMU exceptions and various software optimizations. Evaluations on SPEC CINT2006 benchmark suite and some real-world applications show that BTMMU achieves 1.40x and 1.36x speedup on IA32-to-MIPS64 and X86_64-to-MIPS64 configurations respectively when comparing with the base QEMU version. The result is compared to a representative previous work and shows its advantage.

CCS Concepts: • **Computer systems organization → Architectures**; • **Software and its engineering → Main memory**; **Virtual memory**; **Software as a service orchestration system**.

Keywords: memory virtualization, cross-ISA, binary translation, hardware, BTMMU

ACM Reference Format:
Kele Huang, Fuxin Zhang, Cun Li, Gen Niu, Junrong Wu, and Tianyi Liu. 2021. BTMMU: An Efficient and Versatile Cross-ISA Memory Virtualization. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '21), April 16, 2021, Virtual, USA.* ACM, New York, NY, USA, 13 pages.
https://doi.org/10.1145/3453933.3454015

*Corresponding author

VEE '21, April 16, 2021, Virtual, USA

ACM ISBN 978-1-4503-8394-3/21/04…$15.00
https://doi.org/10.1145/3453933.3454015

1 Introduction

Memory virtualization for full system DBT is challenging due to the following facts:

First, the virtual memory system of guest and host architecture can be very different. They can differ in address space size, page size, organization ways, protection bits, etc. For instance, the virtual address space's size is 32 bits on x86_32, while 48 bits on x86_64. And the page size is typically 4k on x86, while 8k on UltraSPARC architecture 2007 [20] and 16k on Loongson [12]. Thus mapping guest memory access into host access can be quite complex. Effective techniques for same-ISA virtualization such as shadow page table and second level address translation [1] are no longer directly applicable.

Second, guest virtual address (GVA) and host virtual address (HVA) accesses are closely interleaved. As shown in Figure 1, there are two kinds of memory accesses in translated code. 1) accesses to the emulated guest CPU states or emulators' data structures. The addresses of this kind of accesses are known HVA. 2) accesses to guest memory. The addresses of this kind of accesses are often unknown at translation time. They are generated at run time and need to be converted into HVA. The code in Figure 1 is just for illustration because it directly uses GVA without any conversion. The working code has to add a code sequence to perform the address translation or switch the address space somehow.

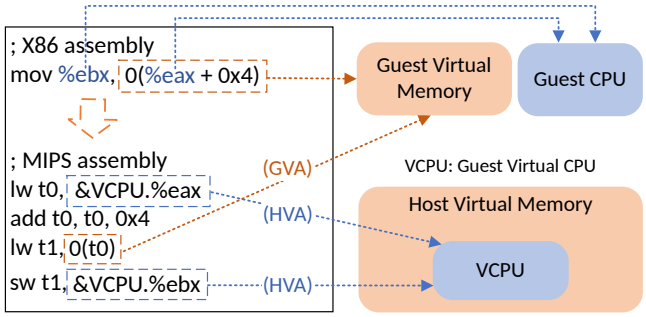


Figure 1. Instruction translation in cross-ISA system emulation of X86 on MIPS.

QEMU [3] uses a generic and software-based approach. It maintains a software MMU (SoftMMU) to map GVA into HVA and generate a code sequence for each guest memory access to perform the address conversion. SoftMMU is flexible so that QEMU can support a large set of combinations of guest and host ISAs. But it greatly impacts performance. Even though many efforts have been devoted to improve the SoftMMU, our experiences show that the latest QEMU version with SoftMMU often runs 10 times slower than the host.

By embedding guest virtual address space into host address space and using host TLB to translate guest addresses, ESPT [7] achieved significant speedup over SoftMMU. HSPT [23] followed the same idea, but it used the *mmap* system call to avoid kernel module dependency. Faravelon et al. [10] also implemented a similar scheme. However, all these works share some limitations. First, the guest's virtual address space has to be smaller than the host's. In fact, to support multiple guest processes' address space mapping (which is vital for performance) or multiple privilege levels, the latter has to be many times larger than the former. All these works choose to emulate 32-bit guests on 64-bit hosts, significantly reducing their applicable range. Second, it is hard to handle different page size combinations of guest and host. Third, it is also hard to handle privileged memory accesses.

By utilizing the memory virtualization support of modern CPU(such as Intel VT-x's Extended Page Tables and AMD SVM's Nested Page Tables), Captive [16] and another scheme of Faravelon et al. [10] achieved significant speedup too. The main idea is running emulators in guest kernel mode to directly control page mappings and rely on the second level address translation hardware to translate emulators' addresses. These schemes also require a larger host virtual address space. And the overhead of switching between guest and host context needs to be carefully controlled to get good performance.

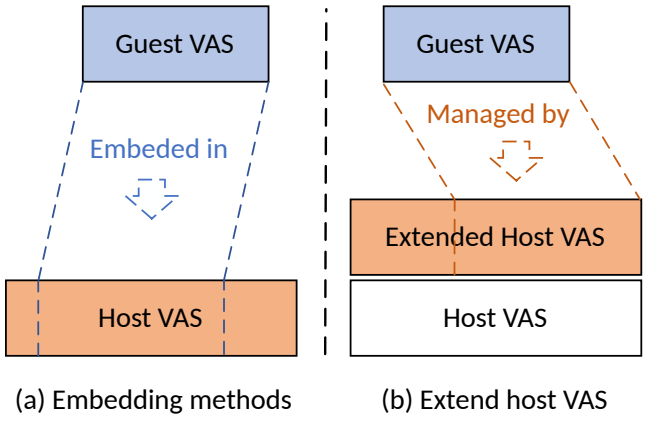


Figure 2. Guest virtual address space handling.

This paper proposes a new approach, named BTMMU *binary translation memory management unit*. Figure 2 shows the difference between BTMMU and previous embedding schemes. BTMMU utilizes a low-cost hardware extension of host MMU called Dual-TLB [25] to implement memory virtualization.

The main contributions of this paper are as follows:

- We propose a cross-ISA memory virtualization approach named BTMMU, which addresses most known limitations of previous studies.
- We implement several optimizations to improve efficiency, including low overhead exception based translation path, multi-SPT for multitasking guests, MMIO retranslation, etc.

- We evaluate the prototype on SPEC CINT2006 benchmarks suite and some real-world applications. BTMMU achieves on average 1.40x and 1.36x speedup on IA32-to-MIPS64 and X86_64-to-MIPS64 configurations respectively when compared with the baseline QEMU.
- We reimplement HSPT on the same QEMU version, compare it with BTMMU, and make some in-depth analysis.

The rest of this paper is organized as follows. Section 2 gives background and motivation. The rationale and details of BTMMU are presented in Section 3. Section 4 gives experimental evaluations of BTMMU and a brief discussion. Section 5 presents related works. And section 6 concludes the above.

2 Background and Motivation

2.1 Virtualized Address Translation

Formally, address translation is the mapping of addresses in virtual address space to addresses in physical address space [4]. In the virtualization system, address translation is more complicated. A guest memory emulation requires multi-level translations before finally reaching the host physical memory [5]. In general, there are three-level address translations:

$GVA \Rightarrow GPA$: from guest virtual address to guest physical address via guest page table (GPT) in guest OS.

$GPA \Rightarrow HVA$: from guest physical address to host virtual address via a memory mapping table (MMT) in VMM. MMT is used to redirect GPA to a piece of host virtual memory that is used to emulate guest physical memory.

$HVA \Rightarrow HPA$: from host virtual address to host physical address via host page table (HPT) in host OS.

2.2 Address Translation in Full System DBT

Dynamic binary translation (DBT) [14] dynamically translates a block of guest binary code into a semantically equivalent native instruction sequence for later execution. Full system DBT emulates the whole machine states, including the guest CPU, memory and necessary peripheral devices. There are lots of full system DBT research prototypes and products, such as Embra [24], VMware workstation [15], and QEMU [3]. Address translation in full system DBT is much more complex than a user mode DBT.

Figure 3 takes QEMU-i386-softmmu (the guest is x86) as an example and depicts its address translation flow. QEMU uses a software data structure (VCPU) to record guest CPU states and emulates guest CPU behavior via accesses to the VCPU. And it uses some continuous host virtual memory blocks to emulate guest physical memory.

In the VCPU, the emulated guest register CR3 (GCR3) is a page table base register pointing to the guest page table (GPT). Access to guest memory is GVA memory access, and access to guest CPU status is redirected to VCPU (HVA memory access). Both GVA and HVA memory accesses will be generated when the translated native code is executed.

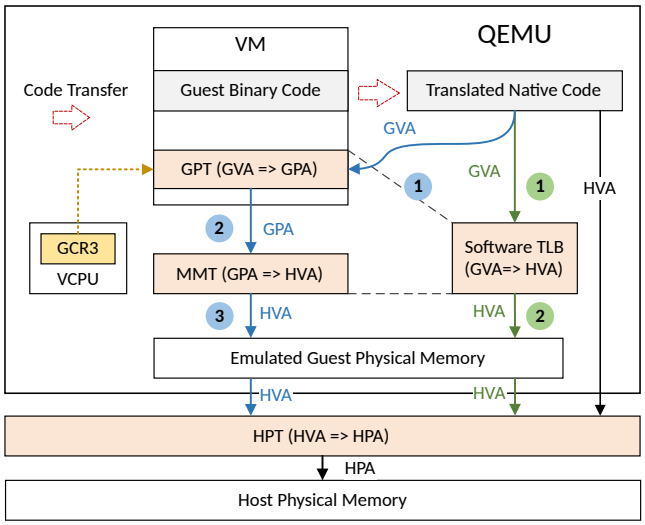


Figure 3. Address translation in QEMU-i386-softmmu

A GVA memory access has to go through three steps. First, it needs to walk the GPT to translate GVA into GPA. Second, it needs to lookup the memory mapping table (MMT) with GPA to get HVA. MMT is maintained by QEMU and used to redirect GPA to a host virtual address block that is used to emulate guest physical memory. Third, it needs to walk the host page table (HPT) to get HPA. Then the host CPU can use HPA to access the host's physical memory.

Such a complex address translation procedure for every GVA access will inevitably lead to substantial performance loss. QEMU uses a platform-independent software Translation Lookaside Buffer (SoftTLB) to cache $GVA \Rightarrow HVA$ mappings. The generated code will first try to read GVA to HVA mapping from SoftTLB. A complete address translation is only needed when SoftTLB fails to provide the mapping (that is called an access miss).

2.3 Software-Based Optimizations

Even though SoftTLB trims three-level address translations to two-level, it is still not efficient enough for the following reasons: (1) SoftTLB needs about ten instructions to lookup the SoftTLB even when the access hits. (2) SoftTLB is designed as a direct mapping structure to reduce the hit path, limiting its hit rate. (3) SoftTLB cannot access HPT directly (stored in host kernel) to further trim the address translation to one level.

To improve the performance of SoftTLB, Tong et al. [19] and Hong et al. [11] quantitatively analyze where time is spent in memory emulation of QEMU and propose a number of optimizations for SoftTLB. The optimizations include

making SoftTLB dynamically-sized and using a small fully-associated victim TLB. They help to achieve a higher hit rate so that time spent on the miss path is reduced. However, the hit path overhead is not reduced.

2.4 Hardware-Assisted Optimizations

As mentioned, previous work such as ESPT, HSPT, Captive and Faravelon's work tried to utilize existing hardware to make improvement. By embedding guest virtual address space into host address space, GVA can be directly accessed in the same way as HVA. This effectively reduces the access path to one level. But the emulator needs to maintain GVA to HPA mappings. ESPT and HSPT use the host page table to store the mapping. Each time a valid GVA is accessed, if there is no mapping for it in host TLB, the GVA to HPA mapping will be decided by host OS and QEMU, then loaded into host TLB and stored in the host page table. ESPT uses a kernel module to operate on host page tables directly. HSPT uses *mmap* syscall to manipulate HPT indirectly.

Captive and one of Faravelon's work (he implemented both ESPT like solution and Captive like solution) utilized the virtualization hardware. Emulators run in guest kernel mode, so they have full control over the virtualized bare metal machine. With direct access to page tables of themselves, these emulators can setup direct GVA to HPA mapping and reduce the memory access to one level. Note that the HPA seen by emulators here is not the real physical memory address of the host, but the second level address translation mechanism will translate it transparently.

Because existing hardware is not designed with cross-ISA in mind, these solutions have to bear with some severe limitations. Guest virtual address space has to be smaller than the host to enable space embedding. Guest with different page sizes will be hard to support. And it is not easy to handle privileged accesses.

2.5 Our Motivation

Since some limitations cannot be easily solved with current software and hardware, we start to look at the hardware extensions. Dual-TLB [25] proposed by Wang et al. well addressed this issue. Dual-TLB extends the host's virtual address space by adding a two bits Machine IDentifier (MID) in each TLB entry. The page size challenge can be handled by the software managed multiple page size enabled MIPS MMU. Dual-TLB provides dedicate exception entrance for extended address spaces. It can be used to reduce the overhead of guest memory exception handling. However, only simulated processor was available at that time, so the original paper performed only limited evaluations. Today Dual-TLB feature is available in several commercial processors, so we can perform further studies. Our work implements a more complete system on real hardware, resolves several blocking issues for real deployments, and discusses possible further improvements.

3 BTMMU Design

BTMMU composes of a low-cost extension to host MMU, a kernel module performing privileged operations and a modified QEMU. The design details are described in this section. Section 3.1 gives a brief description of the hardware extension. Section 3.2 presents the software implementation of BTMMU. Several optimizations are discussed in section 3.3.

3.1 BTMMU Hardware Extension

We adopt the Loongson 3A4000 as our experimental platform, which is a MIPS64 compatible processor. The CPU cores of 3A4000 are implemented with GS464v [12] architecture, which implements the hardware extension (Dual-TLB) proposed by Wang et al. [25]. Readers can refer to the paper for detailed rationale and implementation descriptions. Here we give a short summary:

1. Based upon the standard MIPS64 MMU (with 64-entry full associated VTLB and 2048-entry 8-way associated FTLB), each TLB entry is extended with two bits, which is called Machine IDentifier (MID). The processor maintains a MID context for each memory access. A TLB entry is only hit when the MID of current access matches the MID context. MID effectively provides four concurrent virtual address spaces for processes. MID context defaults to zero so existing software can run without any modification. MID value 0 is used for native host virtual space, value 1 is reserved for homogeneous virtualization spaces, and value 2 and 3 can be used as heterogeneous virtual spaces. We use 2 for non-privileged GVA and 3 for privileged GVA.
2. Two ways are added to provide control of MID context. The first one is a new instruction named SETMID. SETMID acts as a prefix instruction, memory access next to it will use the MID value set by it. The second one is a global MID context field, which provides a default context if current memory access is not prefixed by SETMID instruction.
3. Dedicated exception entrances are provided for MMU exceptions with MID value 2 or 3.

With the help of such a hardware extension, we can solve the mentioned limitations of previous studies. The virtual address extension allows up to 48bit effective guest virtual address, which is enough in common situations. Software-managed MMU and dedicated entrances can be used to implement the different policies for different virtual address spaces. The virtual address segmentation and page size can be different for each MID space.

Figure 4 depicts the address translation in BTMMU. The hardware TLB stores both the page mappings for guest and host pages. Although TLB entries are physically shared, with dedicated exception entrances and different handling policies, we can think that there are logically separated guest

TLB(GTLB) and a host TLB(HTLB). GTLB stores both non-privileged guest address mappings $(2 \parallel GVA) \Rightarrow HPA$ and privileged guest address mappings $(3 \parallel GVA) \Rightarrow HPA$. HTLB stores address mappings $(0 \parallel HVA) \Rightarrow HPA$.

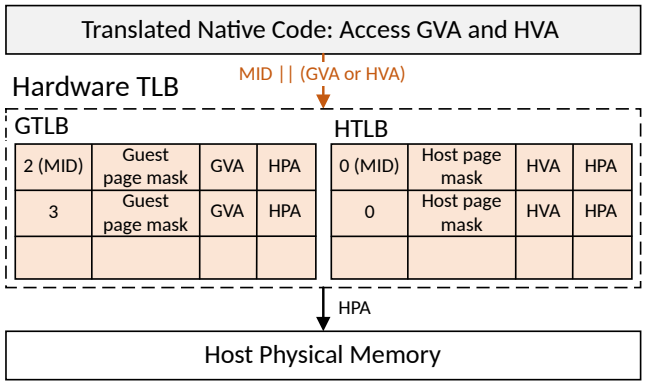


Figure 4. Address translation in BTMMU.

3.2 Software Implementation

The overall architecture of BTMMU is similar to that of ESPT. QEMU maintains GVA to GPA and GPA to HVA mappings while OS kernel maintains HVA to HPA mappings, so it is natural to let them cooperate to deduce GVA to HPA mapping, which will be stored in the shadow page table and TLB. A kernel module is used to handle necessary privileged operations, and it will cooperate with QEMU to reduce the memory access path to one level.

BTMMU is different from ESPT in the following ways:

1. BTMMU does not rely on the signal mechanism. Instead, it utilizes the dedicated exceptions for guest virtual address spaces. In this way, it can reduce the overhead of setting up mappings.
2. BTMMU has a customized shadow page table structure in its kernel module in order to be complete independent of host page table handling. While ESPT embedded shadow page table into host page table, which limited its applicable range.
3. BTMMU generate an instruction pair for each guest memory access, as shown in Figure 5. Because GVA and HVA accesses are closely interleaved in translated code, we use SETMID instruction to switch MID context for the next memory access.
4. BTMMU handles both privileged and non-privileged memory accesses while ESPT deals with only the latter.

There are three kinds of TLB-related exceptions in MIPS architecture: TLB miss exception, TLB invalid exception and TLB modify exception [18]. In our design, the host OS will handle HVA related exceptions, and the BTMMU kernel module will handle GTLB related exceptions.

Guest Instruction	Host Instruction
Non-privileged GVA Memory Access E.g. MOV %EBX, 0(%EAX); ; GVA in %EAX	SETMID 2; Load T1, 0(T0); ; GVA in T0
Privileged GVA Memory Access E.g. MOV 0(%EBX), %EAX; ; GVA in %EBX	SETMID 3; Store T0, 0(T1); ; GVA in T1

Figure 5. Instruction translation in BTMMU.

In the following sections, we name GVA related exceptions as GTLB exceptions for convenience and present how BTMMU handles them.

3.2.1 Regular Exception Handler: Using Hardware Exception Mechanism. The original Dual-TLB prototype did not cache page mappings in kernel. As shown in Figure 6, when a GTLB miss happens, BTMMU will do the following to handle it:

1. saves the exception context in GTLB miss exception handler and redirect non-privileged exception handler (NPEH) in QEMU. We achieve the control flow redirection by saving the non-privileged exception handler's address to Exception Program Counter (EPC) and then issuing Exception RETurn (ERET) instruction. The using of EPC and ERET is the regular exception handling procedure in MIPS compatible architecture [18].
2. NPEH walks the GPT to get GPA for current GVA and then lookups memory mapping table (MMT) to get HVA. Then NPEH informs the $(GMID \parallel GVA) \Rightarrow HVA$ mapping to host kernel via *ioctl*.
3. The kernel module gets HPA for this HVA by walking the host page table. Then it refills $(GMID \parallel GVA) \Rightarrow HVA$ mapping entry into GTLB.
4. Restores exception context and re-executes the instruction that triggered this exception.

This procedure applies to both non-privileged and privileged memory accesses because exception handlers can tell the type of GVA from the MID context. As we mentioned before, MID 2 is used for non-privileged GVA and 3 is for privileged GVA.

3.2.2 Fast Exception Handler: Using Shadow Page Table. Due to the limited number of TLB entries, GTLB miss will happen very frequently, therefore, the overheads from exception handlers might offset the benefits of one-level mapping.

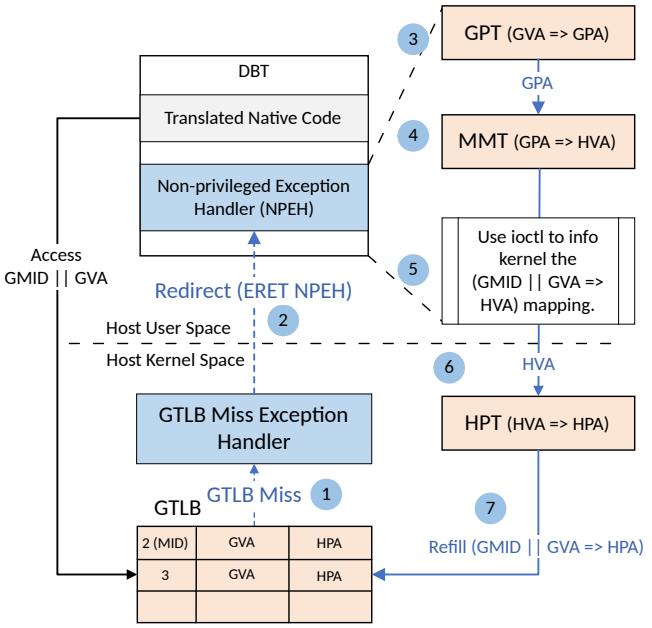


Figure 6. BTMMU regular exception handler

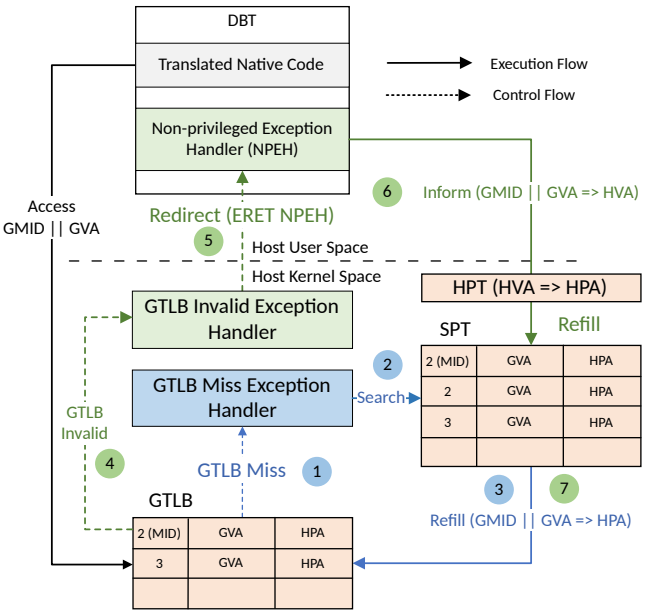


Figure 7. BTMMU fast exception handler

Caching the mappings in kernel is a natural way to optimize the process. The cache is implemented as a shadow page table (SPT). SPT stores $(GMID \parallel GVA) \Rightarrow HPA$ mapping entries in the host kernel. A new mapping will be stored in GTLB as well as SPT. When a GTLB miss happens, the handler will first try to refill from SPT.

Only when SPT does not have valid mapping, it will trigger GTLB invalid exception and redirect control to NPEH. Figure 7 depicts the fast exception handling procedure with SPT.

3.2.3 SPT Design. SPT can be flexibly configured to accommodate different kinds of emulations. Figure 8 shows the SPT design of BTMMU for a real configuration, where the guest is X86_64 (48-bit virtual address space and 4k page size), and the host is MIPS64 (48-bit virtual address space and 16k page size). Since the guest's page size is 4k, we use the lowest 12-bit of the guest address as the guest virtual page offset (GVPO). Each SPT entry needs 8 bytes (on the 64-bit host), so a 16k host's page can accommodate 2^{11} entries. Thus, we need three 11-bits of guest virtual address as the guest virtual page number (GVPN) to index guest page, and the left 3-bit of the guest virtual address is used as page global directory (PGD) index.

For each page, we set up the mapping in SPT only when it is accessed and reclaim it when SPT is flushed. This strategy can avoid unnecessary memory usage.

Since all guest processes share the same address space identifier (ASID) with QEMU, their address mappings are indistinguishable for the GTLB hardware. Therefore, we need to flush GTLB and SPT upon the guest process switch. SPT

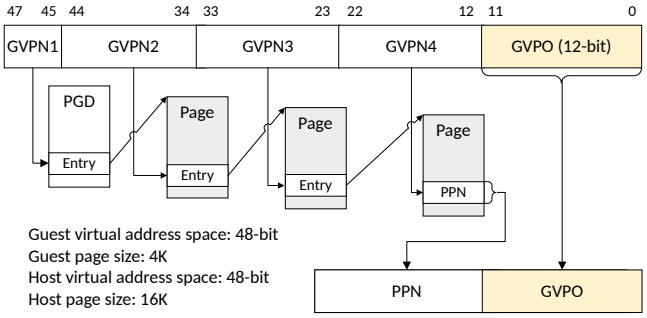


Figure 8. Design of SPT for x86_64 linux guest

flush can be avoided, and we will discuss the optimization shortly after.

For correctness, SPT should be consistent with GPT. The coherency between SPT and GPT is a classical problem. One solution to synchronization between SPT and GPT is memory tracing [1, 5]. The main idea is monitoring all writes to guest page tables by write-protect them. However, this scheme can itself be a source of overheads.

A more efficient method is intercepting some guest TLB flushing operations and update SPT accordingly. The interception strategy is feasible because almost all architectures (like x86 and arm) explicitly call out that they have no coherency guarantees between the processor's page table and hardware TLB. Alternatively, they use certain privileged instructions to flush TLB (e.g., *invlpg*, *mov %cr3* in x86) [5].

3.3 Optimizations

A series of optimizations are implemented in BTMMU. Besides the usual software techniques, we find that the hardware support enables more optimizations.

3.3.1 Multi-SPT. In multitasking OSes, context-switch of guest processes occurs periodically. Context-switch of guest processes might lead to GTLB and SPT flush because their address mappings are not distinguishable for TLB hardware as aforementioned. Fortunately, if we can maintain one SPT for each guest process, then we only need to switch SPT rather than flush SPT. BTMMU uses the guest process's page table base address as an identifier to distinguish different SPT. Provided the process is alive, its page table base address should be unique. We have to track the guest process state and only keep *running* processes' SPT. To track which guest process is in *running* state, we monitor the process termination related syscalls (such as *exit* and *exit_group* in Linux) issued by the guest process in QEMU. Accordingly, our multi-SPT optimization does not rely on guest system's hardware ASID.

HSPT also implements multi-SPT, but it relies on the guest system's ASID support, and each SPT will consume more host virtual address space, which will make embedding more difficult. In contrast, BTMMU's multi-SPT does not consume the host virtual address space.

Multi-SPT needs a special treatment to support privileged address space. In common guest OSes all guest processes share privileged virtual address space, so their privileged page table entries are identical. Storing them in each process's SPT will not only waste memory, but also leads to synchronization issue. We will have to copy all modifications made to privileged space of one SPT to all other SPTs.

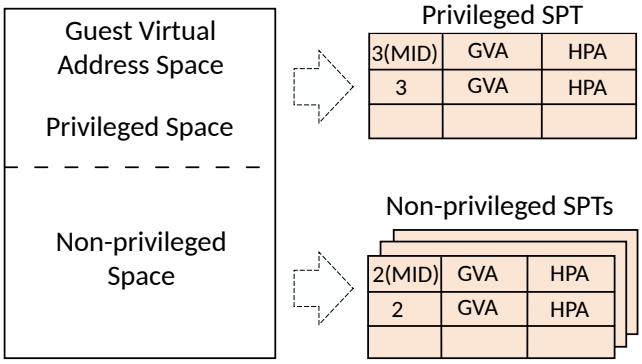


Figure 9. Optimize privileged memory emulation with Multi-SPT optimization

To address this, we maintain a special SPT for the privileged virtual address space of guest processes. Figure 9 shows the design. BTMMU uses MID value 3 for privileged GVA and 2 for non-privileged GVA. Accordingly, privileged GVA accesses will use the unique privileged SPT, while non-privileged GVA accesses will use the current process's SPT. Upon guest context-switch, only non-privileged SPT is switched.

3.3.2 MMIO Retranslation. There are some difficulties to overcome for accelerating privileged memory accesses. Dual-TLB authors mentioned that due to the inefficiency memory-mapped IO(MMIO) handling prevented them from achieve higher speedup for linux kernel boot. Several previous studies did not support privileged guest memory accesses at all, they will fall back to QEMU's softMMU for such accesses. The reasons might include: 1) For computational applications, privileged accesses are not so relevant for their performance because they are typically much less than non-privileged ones. 2) MMIO might greatly reduce the performance benefit.

MMIO access is often not backed by physical memory pages. When GVA is translated to GPA, the emulator will find out it is an IO, and emulate the IO by calling related device emulation code. So, when guest MMIO triggers a GTLB miss, no GVA to GPA mapping can be set up and used in later access. Without special treatment, every MMIO will trigger GTLB miss. All the exception handling work is useless, and only leads to more overhead.

Algorithm 1: MMIO Retranslation

Input: Translation block (TB) to be executed
Output: None

```
if TB is marked as MMIO involved and not retranslated then
    invalid TB;
    retranslate TB without BTMMU support;
    mark TB as retranslated;
end
execute TB;
if TB is MMIO involved and has not been marked then
    mark TB as MMIO-involved;
    unlink TB;
end
```

BTMMU implements an optimization to avoid the overhead. Algorithm 1 presents the scheme. At first, all blocks are translated in BTMMU enable mode. When MMIO is emulated, the corresponding translation block (TB) will be marked and unlink from the TB chain. The next execution of the TB will try to look it up from the code cache. At that time, we will retranslate it without BTMMU support (fallback to SoftMMU). In this way, we can enable BTMMU for privileged accesses, and don't suffer the overhead for MMIO accesses. In this way, we can enable BTMMU for privileged accesses, and don't suffer the overhead for MMIO accesses.

4 Evaluation

In this section, we evaluate the effectiveness of BTMMU on both applicability and efficiency. First, we show that BTMMU is able to support emulations of the 32-bit guest on the 64-bit host (with different page sizes) and 64-bit guest on the 64-bit host (with different page sizes). Then we test and verify the effect of SPT, multi-SPT and MMIO retranslation optimizations. Finally, we compare BTMMU to previous works and make some discussions.

4.1 Experimental Setup

Our host experimental platforms are Loongson 3A4000 and X86 described in Table 1 and Table 2 separately. Loongson 3A4000 host is used for most experimental results. And X86 host is used to verify the implementation of HSPT. We select SPEC CINT2006, some real-world applications, and IOZone as our benchmarks running in the guest system. The guest platforms that we choose to virtualize are x86_32 (running Ubuntu 16.04.6 LTS (i686)) and x86_64 (running Ubuntu 16.04.1 LTS (x86_64)), which are fully supported by QEMU.

Table 1. Host Experimental Platform

System TL630-V001	
Architecture	GS464V
Model	Loongson 3A4000
Cores/Threads	4/4
Frequency	1.8 GHz
VTLB (used as host TLB)	64-entry fully associative
FTLB (used as guest TLB)	2048-entry, 8-way associative
OS	Loongnix-1.0.2003, kernel version 4.19.73
Memory	8 GB
VMM	QEMU version 4.2.1

Table 2. x86 Host Experimental Platform for HSPT

System LENOVO 81J0	
Architecture	X86_64
Model	Intel® Core™ i7-8565U
Cores/Threads	4/8
Frequency	1.8 GHz
Shared 2nd-Level TLB	6-way associative, 1536 entries. Plus, 1-GB pages, 4-way, 16 entries
OS	Linux kernel version 5.8.0
Memory	8 GB
VM	QEMU version 4.2.1

4.1.1 SPEC CINT2006 Benchmarks. SPEC CINT2006 is a CPU intensive benchmark suite, stressing processor and memory subsystem. For most experiments, we use the *reference* input set. However, the *train* input dataset is used in comparison with HSPT because the original HSPT paper used that.

4.1.2 Real-world Applications. We select some realistic applications to further verify the applicability of BTMMU. The chosen applications include `Apache`, `Memcached`, `PHP`, and `SQLite`. Apache is a single-threaded command line web server [2]; Memcached is a multi-threaded distributed memory object caching system [13]; PHP test uses a simple script to test basic PHP functionalities; SQLite is a C program that provides a lightweight disk-based database.

4.1.3 IOZone. IOZone [6] is a benchmark program that evaluates a computer system's ability to handle I/O operations involving files. We choose IOZone to evaluate VM's I/O performance.

4.2 Applicability of BTMMU

We prototype BTMMU on IA32-to-MIPS64 QEMU and X86_64-to-MIPS64 QEMU to evaluate the applicability of BTMMU. We choose these two configurations because their guests and hosts have representative different virtual address spaces. IA32 has a 32-bit virtual address space with a 4KB page size; X86_64 has a 48-bit virtual address space with a 4KB page size; Loongson 3A4000 has a 48-bit virtual address space with default 16KB page size (which can be changed). These two configurations show that BTMMU is versatile enough to handle emulations where previous works have difficulty to deal with. Figure 10 presents the performance of BTMMU.

Figure 10(a) and figure 10(c) present the relative speedup of BTMMU in IA32-to-MIPS64 QEMU and X86_64-to-MIPS64 QEMU compared to baseline QEMU; the x-axis lists the benchmarks (SPEC CINT2006 and four real-world applications), and the y-axis represents relative speedups. Figure 10(b) and figure 10(d) present each benchmark's absolute runtime of baseline QEMU and BTMMU in seconds; the x-axis lists benchmarks, and the y-axis represents the execution time in seconds. Results show that BTMMU gains on average 1.40x speedup on IA32-to-MIPS64 configuration and 1.36x on X86_64-to-MIPS64 configuration over the baseline QEMU. Among all benchmarks, 456.hmmer gains the highest performance improvement of 1.89x and 1.81x. The results of IA32-to-MIPS64 and X86_64-to-MIPS64 emulations are almost consistent, indicating that the performance of BTMMU is stable.

4.3 The Effect of SPT

To evaluate the effect of SPT, we have a comparison between BTMMU without SPT support and BTMMU with only one SPT (single-SPT) support. Figure 11 presents the relative speedups of SPEC CINT2006. The baseline is IA32-to-MIPS64

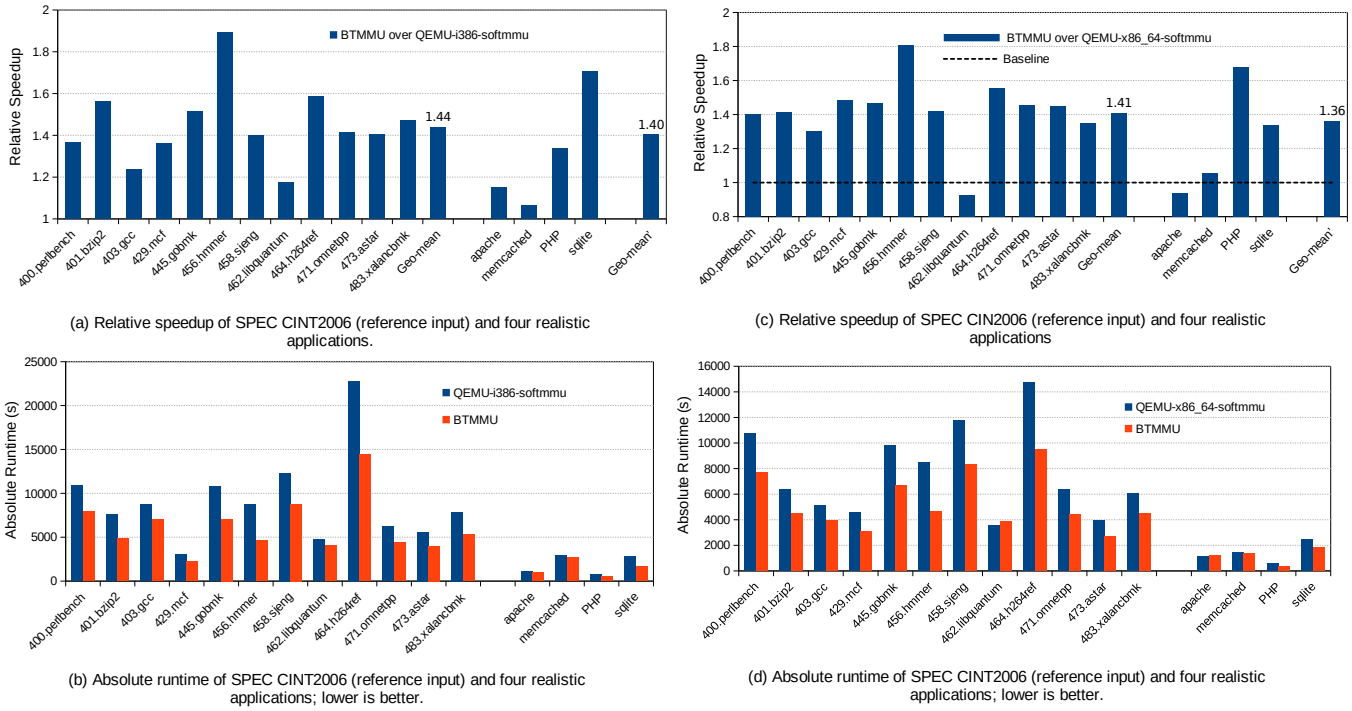


Figure 10. BTMMU vs. baseline QEMU in IA32-to-MIPS64 and x86_64-to-MIPS64 emulations.

QEMU. BTMMU with single SPT and BTMMU without SPT are compared with the baseline. The x-axis represents benchmarks, and the y-axis represents relative speedups. Results show that BTMMU without SPT support only gains 0.18x performance of baseline QEMU, while BTMMU with single-SPT support outperforms baseline QEMU with on average 1.36x speedup. Due to the high frequency of GTLB refill exceptions, their overhead can bring significant performance loss. The result proves the importance of controlling the overhead of GTLB exception handling.

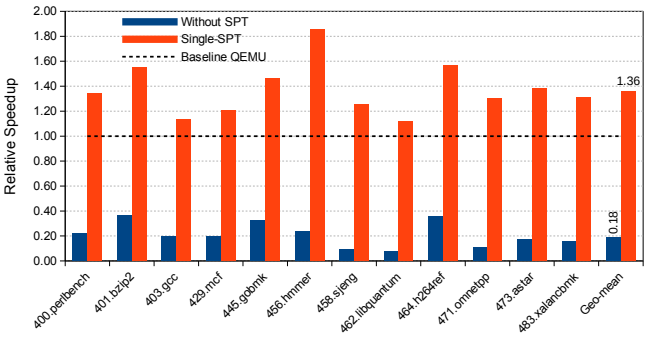


Figure 11. Speedup comparison of no-SPT and single-SPT.

4.4 The Effect of Multi-SPT

To evaluate the multi-SPT optimization, we have a comparison between BTMMU with single-SPT and BTMMU with multi-SPT support. Figure 12 presents the relative speedups of benchmarks on multi-SPT BTMMU over single-SPT BTMMU on baseline IA32-to-MIPS64 and X86_64-to-MIPS64 QEMU. The x-axis represents the benchmarks, and the y-axis represents the relative speedups.

Results show that multi-SPT BTMMU outperforms single-SPT BTMMU with on average 1.09x and 1.06x speedup on IA32-to-MIPS64 QEMU and X86_64-to-MIPS64 configuration respectively. The results show that multi-SPT surpasses single-SPT in most cases. 403.gcc, 458.sjeng 471.omnetpp and 483.xalancb-mk gain relative high speedup, while others have no significant performance improvement. Moreover, real-world applications apache and memcached have more obvious performance improvements. This is reasonable since they are multi-threaded applications, so they switch more often than other single-threaded applications.

To better understand the performance numbers, we gather several data, including the SPT flush times, the GTLB miss times and the GTLB invalid times while running *reference* SPEC CINT2006. Table 3 presents the comparisons between single-SPT BTMMU and multi-SPT BTMMU. The results show that multi-SPT reduces SPT flush times nearly by half, and GTLB invalid times by ten times. In contrast, GTLB miss times is almost unchanged. This is also reasonable because the GTLB miss rate is mainly affected by the TLB hardware and program behavior.

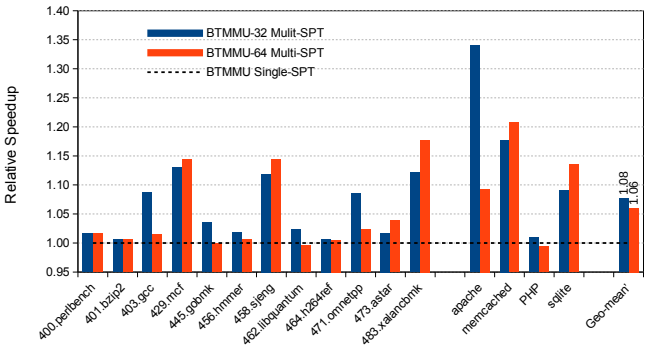


Figure 12. Speedup comparison of multi-SPT and single-SPT.

Table 3. Comparison between Single-SPT and Multi-SPT

	SPT flush	GTLB miss	GTLB invalid
Single-SPT	64132	7182031638	285879115
Multi-SPT	38067	7182491083	20214384

4.5 The Effect of MMIO Retranslation

We use IOZone to evaluate guest I/O performance. Figure 13 shows IOZone comparison results of BTMMU with and without MMIO retranslation support over baseline QEMU. Results show that BTMMU without MMIO retranslation support achieves only 0.79x performance of baseline QEMU, while BTMMU with MMIO retranslation support outperforms baseline QEMU on average 1.04x. The results show that reducing the MMIO-related memory emulation overheads is necessary, and the optimization works as expected.

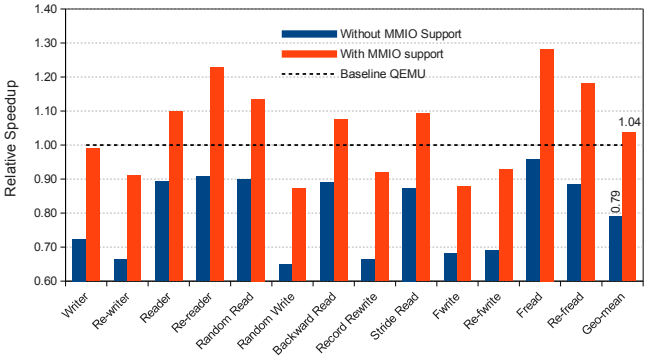
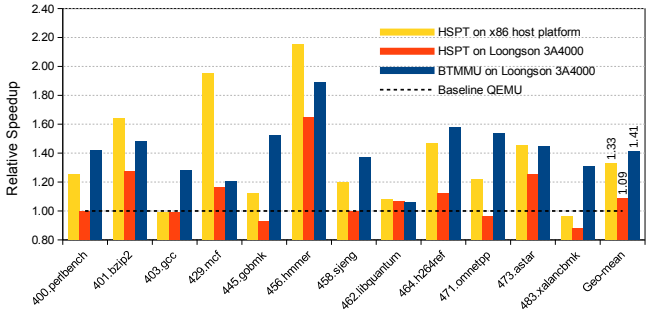


Figure 13. Performance comparison of BTMMU with and without MMIO support.

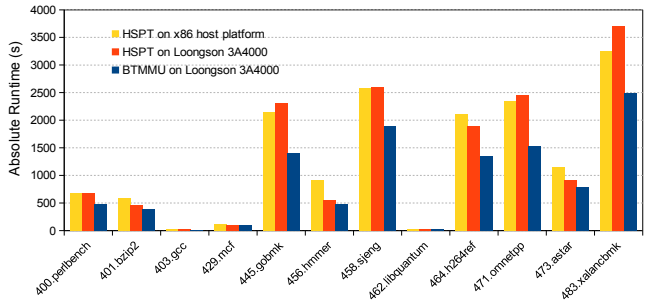
4.6 Comparison between Reproduced HSPT and BTMMU

Although HSPT presented its experimental results, they were produced on a much older version of QEMU and a different host platform, making direct comparison invalid. SoftTLB of QEMU version 4.2.1 has been improved a lot by works [11, 19] since 2015. Therefore, to have an effective comparison, we reimplement HSPT on the same version of QEMU and collect results on the same host platform as BTMMU.

Figure 14 shows the performance comparison of HSPT and BTMMU in two experiment hosts. The selected benchmark is SPEC CINT2006 with *train* dataset. And the emulated guest is IA32 because HSPT can only handle 32 guests. On x86 host, reproduced HSPT outperforms baseline QEMU by an average speedup of 1.33x. While on Loongson host, reproduced HSPT outperforms baseline QEMU by an average speedup of 1.09x. It is much lower than BTMMU's 1.41x. The host kernel page size configuration might be a reason for HSPT's low efficiency on Loongson. To accommodate the use of *mmap* in HSPT, we have to use 4KB page size for the host kernel, while the default page size is 16KB. 4KB page size is uncommon setting for Loongson, the OS might not operate in optimal way.



(a) Comparison of relative speedup of SPEC CINT2006 (with train input); higher is better.



(b) Comparison of absolute runtime of SPEC CINT2006 (with train input) in seconds; lower is better.

Figure 14. Performance comparison of HSPT and BTMMU on X86 and Loongson 3A4000 host.

4.6.1 In-Depth Analysis. To have an in-depth analysis, we choose the best performer *456.hmmer* and other five long runtime benchmarks *445.gobmk*, *458.sjeng*, *464.h2-64ref*, *471.omnetpp* and *483.xalancbmk* (all of their running time exceed 1000 seconds), then collects and compare their slow path memory emulation exception rates.

Table 4 presents the collected slow path memory exception types of QEMU, HSPT and BTMMU. When the exception

happens, they will go through the same slow path of memory emulation. In such case a new GVA to HPA mapping have to be setup via walking the GPT. So these rates directly relate to memory emulation overhead.

Table 4. Memory emulation exception in baseline QEMU, HSPT and BTMMU

Approaches	Exception Types
QEMU	SoftTLB miss
HSPT	Segmentation fault
BTMMU	GTLB invalid exception

Table 5 shows the slow path exception rates of published older QEMU, baseline QEMU, reproduced-HSPT and BTMMU. All results are produced on the host platform described in Table 1. The TLB miss rate of older version QEMU is extracted from the original HSPT paper. The exception rate of older version QEMU is on average 2.79%; our baseline QEMU is on average 0.1630%; reproduced-HSPT is on average 0.0104%; BTMMU is on average 0.0026%. Results of older version QEMU and our baseline QEMU reveal that QEMU has achieved great progress, which explains why the speedup of reproduced HSPT and BTMMU is lower than that of published HSPT.

Reproduced HSPT's exception rate has a ten-fold reduction compared with baseline QEMU, matching published HSPT results. BTMMU reduces the slow path exception rate from 0.1630% to 0.0026%, which is a 62.7x reduction. BTMMU has lower exception rate than HSPT, which can explains in part the performance advantage. BTMMU promotes not only non-privileged memory emulation but also privileged memory emulation. This might be the main reason for its lower exception rate. What's more, BTMMU should have lower exception handling overhead due to dedicated exception entrance. The lower exception rate and lower exception overhead lead to better performance.

Table 5. Memory exception rates comparison

	Old-QEMU	Base-QEMU	HSPT	BTMMU
445	3.17%	0.0407%	0.0168%	0.0028%
456	2.20%	0.3759%	0.0034%	0.0012%
458	4.90%	0.0393%	0.0112%	0.0024%
464	1.62%	0.0366%	0.0053%	0.0015%
471	2.24%	1.8128%	0.0158%	0.0038%
483	3.84%	0.4720%	0.0239%	0.0061%
Mean	2.79%	0.1630%	0.0104%	0.0026%

4.7 Discussion

The experimental results show that BTMMU can efficiently handle cross-ISA memory virtualization of different configurations. BTMMU is more versatile than previous approaches by removing limitations on virtual address space size and page size. Utilizing dedicated hardware exceptions and SPT further improves the efficiency of BTMMU. What's more, BTMMU's multi-SPT optimization consumes no host virtual address space, does not rely on hardware ASID, and supports both non-privileged and privileged memory emulations. These features are important for real deployments.

We make a direct performance comparison with HSPT, which is representative. Analysis shows that only 0.0026% of memory accesses in BTMMU needs to go through the slow path emulation, which is already quite low.

To achieve near native memory emulation, we have to control the fast path emulation overheads too. For QEMU's softMMU, the fast path has more than ten instructions including several memory accesses. For BTMMU, we have two instructions. The extra SETMID instruction will bring some overhead. Current hardware does not support a setting without SETMID. To evaluate the overhead, we evaluate the result of one more SETMID (use two SETMID for each access). As shown in Figure 15, the additional SETMID does bring nontrival performance loss. So we can expect a similar speedup if we can avoid the SETMID instruction.

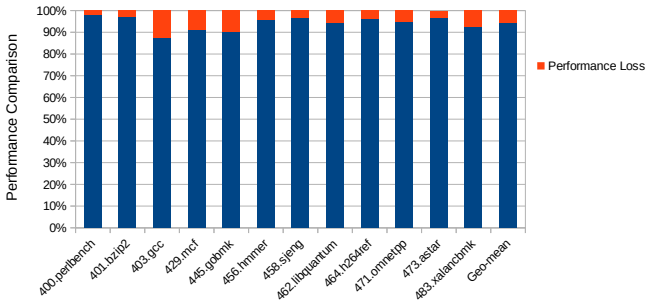


Figure 15. Performance Loss caused by Extra SETMID.

In fact, SETMID can be removed by embedding MID bits into virtual address. Current processors have less than 64 bit effective virtual address bits, we can use some reserved bits to act as MID to avoid an extra instruction. We leave it as a future work.

Overall, Real-world applications have lower speedup than SPEC CINT2006 applications. We think the reason is that TLB locality is lower for the former. In the future work we will have more investigation on this.

BTMMU can not handle the emulation where the guest's virtual address space is larger than the host's. Currently, there are few systems with virtual address spaces larger than 48bits. New hosts can extend their virtual address spaces if necessary.

5 Related Work

Memory virtualization is one of the essential parts of system virtualization. Some effective techniques are developed to improve the efficiency of multiple level address translation of memory virtualization. Shadow paging [22] is software improvement. It establishes a 'shadow' of guest page table in OS kernel to record mapping from GVA to HPA. So multiple level translations are reduced to one level, and the efficiency can be greatly improved. But shadow paging wastes memory to store extra page tables and needs costly synchronization with GPT. Hardware support of second level address translation [1] is adopted by mainstream processors, such as Intel VT-x's Extended Page Tables (EPT) [9], AMD SVM's Nested Page Tables (NPT) [21] and IBM Power's Logical to Real Rddress Translation (LRAT) [8].

The above techniques work well for same-ISA system virtualization. However, they are not directly applicable to cross-ISA virtualization. Cross-ISA memory virtualization remains a vital performance bottleneck of cross-ISA virtualization systems.

Cross-ISA memory virtualization can be implemented in software. QEMU [3] is a popular open-source full system emulator. It maintains a software MMU (SoftMMU) to map GVA into HVA and generate a code sequence for each guest memory access to perform the address conversion. Tong et al. [19] made quantitative measures where time is spent in QEMU and proposed several optimizations, including dynamic-sized TLB and victim TLB. Hong et al. [11] also made some efforts on optimizing software-TLB of QEMU. These works managed to reduce the miss rate of QEMU's software TLB and improve performance. But they have not addressed the overhead of software TLB query.

Several approaches devote to utilize the host's existing hardware to achieve efficient memory virtualization. ESPT [7], proposed by Chang et al., try to embed guest virtual address space into host virtual address space so that that memory emulation can utilize host hardware MMU. This effectively reduces the three-level address translation into one level, and results are promising. HSPT [23], proposed by Wang et al., followed a similar idea with ESPT. But it used *mmap* syscall instead of a kernel module to set up direct address mapping and thus more portable. Spink et al. [16, 17] and Faravelon et al. [10] take advantage of the host's hardware memory virtualization resources to achieve high-performance cross-ISA memory virtualization from a bare-metal perspective. However, all these works requires that the host system has a larger virtual address space than the guest. Besides, the hardware they tried to utilize is not designed with cross-ISA in mind, so there are other hard-to-overcome limitations. For example, if the guest has a different page size than the host, it will be very hard to handle.

Wang et al. [25] proposed a low-cost extension to host MMU to enable efficient cross-ISA memory virtualization. With the hardware support for extending virtual address space, multiple page sizes and software customizable paging policy, it can avoid many limitations of the above approaches. But only a rough prototype was implemented, and limited evaluations were performed. Our work utilizes the proposed hardware, solves known limitations of previous studies and makes in-depth evaluations.

6 Conclusion

This paper identifies the challenges faced in cross-ISA memory virtualization and proposes an efficient and versatile solution named BTMMU to address them. With low-cost hardware support, BTMMU properly accommodates both 32-bit and 64-bit guest virtual address space with different page sizes. BTMMU improves both non-privileged and privileged memory accesses. Hardware exception support and shadow page table are utilized to reduce exception handing overhead. BTMMU proposes multi-SPT optimization for multitasking guests, which does not consume the host's virtual address space, is independent of hardware ASID, and supports privileged memory access.

Experimental results show that BTMMU is capable of handling known challenges in cross-ISA memory virtualization efficiently and adequately. SPEC CINT2006 *reference* benchmarks and real-world applications show that BTMMU achieves up to an average 1.40x speedup on IA32-to-MIPS64 configuration and an average 1.36x speedup on X86_64-to-MIPS64 configuration. By comparing with reimplemented HSPT we show that BTMMU has better performance than HSPT. Further analysis shows that BTMMU has lower slow path memory emulation exception rate. And BTMMU can be further improved by removing the extra instruction in fast path.

Acknowledgments

We would like to thank our anonymous reviewers for their insightful comments and feedback on the paper. This work is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDC05020100.

References

[1] Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices* 41, 11 (2006), 2–13.

[2] Apache 2014. *ab - Apache HTTP server benchmarking tool.* Retrieved Oct 9, 2014 from http://httpd.apache.org/docs/current/programs/ab.html

[3] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[4] Randal E Bryant and David R O'Hallaron. 2015. *Computer Systems.* Pearson Education Canada.

[5] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture* 12, 1 (2017), 1–206.

[6] Don Capps and William Norcott. 2008. IOzone filesystem benchmark.

[7] 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. *SIGPLAN Not.* 49, 7 (March 2014), 117–128. https://doi.org/10.1145/2674025.2576201

[8] Xiaotao Chang, Hubertus Franke, Yi Ge, Tao Liu, Kun Wang, Jimi Xenidis, Fei Chen, and Yu Zhang. 2013. Improving virtualization in the presence of software managed translation lookaside buffers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 120–129.

[9] Intel Corporation. 2007. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*. Intel Corporation.

[10] A. Faravelon, O. Gruber, and F. Pétrot. 2017. Optimizing Memory Access Performance Using Hardware Assisted Virtualization in Retargetable Dynamic Binary Translation. In *2017 Euromicro Conference on Digital System Design (DSD)*. 40–46. https://doi.org/10.1109/DSD.2017.41

[11] 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 (Dec. 2015), 24 pages. https://doi.org/10.1145/2837027

[12] Weiwu Hu and Yunji Chen. 2010. GS464V: A high-performance low-power XPU with 512-bit vector extension. In *Proc. Symp. High Performance Chips*. 22–24.

[13] Memcached 2014. *Memcached - a distributed memory object caching system*. Retrieved Dec 15, 2020 from https://memcached.org/

[14] Mark Probst. 2002. Dynamic binary translation. In *UKUUG Linux Developer's Conference*, Vol. 2002.

[15] Mendel Rosenblum. 1999. Vmwares virtual platform. In *Proceedings of hot chips*, Vol. 1999. 185–196.

[16] Tom Spink, Harry Wagstaff, and Björn Franke. 2016. Hardware-Accelerated Cross-Architecture Full-System Virtualization. *ACM Trans. Archit. Code Optim.* 13, 4, Article 36 (Oct. 2016), 25 pages. https://doi.org/10.1145/2996798

[17] Tom Spink, Harry Wagstaff, and Björn Franke. 2019. A Retargetable System-Level {DBT} Hypervisor. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 505–520.

[18] Dominic Sweetman. 2010. *See MIPS run*. Elsevier.

[19] 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 (TACO)* 11, 4 (2015), 1–24.

[20] T2 UltraSPARC. 2006. Supplement to the UltraSPARC architecture 2007.

[21] AMD Virtualization. 2008. Amd-v nested paging. *White paper. [Online] Available: http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx* (2008).

[22] Carl A. Waldspurger. 2003. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2003), 181–194. https://doi.org/10.1145/844128.844146

[23] Zhe Wang, Jianjun Li, Chenggang Wu, Dongyan Yang, Zhenjiang Wang, Wei-Chung Hsu, Bin Li, and Yong Guan. 2015. HSPT: Practical Implementation and Efficient Management of Embedded Shadow Page Tables for Cross-ISA System Virtual Machines. *SIGPLAN Not.* 50, 7 (March 2015), 53–64. https://doi.org/10.1145/2817817.2731188

[24] 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*. 68–79.

[25] Wang Zhen-hua, Jin Guo-jie, and Wang Wen-xiang. 2015. A Dual-TLB Method for MIPS Heterogeneous Virtualization. In *The International Symposium on Code Generation and Optimization (CGO) 2015 (AMAS-BT workshop)*.