HSPT: Practical Implementation and Efficient Management of Embedded Shadow Page Tables for Cross-ISA System Virtual Machines

Zhe Wang[1,2], Jianjun Li[1], Chenggang Wu[1] *, Dongyan Yang[3] †, Zhenjiang Wang[1], Wei-Chung Hsu[4], Bin Li[5] ‡, Yong Guan[6]

[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, [2]University of Chinese Academy of Sciences, [3]China Systems and Technology Lab, IBM, Beijing, China, [4]Dept. Computer Science & Information Engineering, National Taiwan University, [5]Netease, Hangzhou, China, [6]College of information Engineering, Capital Normal University, Beijing, China

[1]{wangzhe12, lijianjun, wucg, wangzhenjiang}@ict.ac.cn, [3]dyyang@cn.ibm.com, [4]hsuwc@csie.ntu.edu.tw, [5]richardustc@gmail.com, [6]guanyong@mail.cnu.edu.cn

Abstract

Cross-ISA (Instruction Set Architecture) system-level virtual machine has a significant research and practical value. For example, several recently announced virtual smart phones for iOS which run smart phone applications on x86 based PCs are deployed on cross-ISA system level virtual machines. Also, for mobile device application development, by emulating the Android/ARM environment on the more powerful x86-64 platform, application development and debugging become more convenient and productive. However, the virtualization layer often incurs high performance overhead. The key overhead comes from memory virtualization where a guest virtual address (GVA) must go through multi-level address translation to become a host physical address (HPA). The Embedded Shadow Page Table (ESPT) approach has been proposed to effectively decrease this address translation cost. ESPT directly maps GVA to HPA, thus avoid the lengthy guest virtual to guest physical, guest physical to host virtual, and host virtual to host physical address translation.

However, the original ESPT work has a few drawbacks. For example, its implementation relies on a loadable kernel module (LKM) to manage the shadow page table. Using LKMs is less desirable for system virtual machines due to portability, security and maintainability concerns. Our work proposes a different, yet more practical, implementation to address the shortcomings. Instead of relying on using LKMs, our approach adopts a shared memory mapping scheme to maintain the shadow page table (SPT) using only "**mmap**" system call. Furthermore, this work studies the support of SPT for multi-processing in greater details. It devices three different SPT organizations and evaluates their strength and weakness with standard and real Android applications on the system virtual machine which emulates the Android/ARM platform on x86-64 systems.

Categories and Subject Descriptors C.0 [*General*]: System Architectures; D.4.2 [*Operating Systems*]: Storage Management—main memory, virtual memory

General Terms Management, Measurement, Performance, Design, Experimentation, Security

Keywords memory virtualization; cross-ISA virtualization; Embedded Shadow Page Table; HSPT; Hosted Shadow Page Table; practical implementation; loadable kernel module; Security; Portability

* To whom correspondence should be addressed.

† This work was done when Dongyan Yang attended Institute of Computing Technology, CAS.

‡ This work was done when Bin Li attended Institute of Computing Technology, CAS.

VEE '15, March 14–15, 2015, Istanbul, Turkey.
Copyright © 2015 ACM 978-1-4503-3450-1/15/03. . . $15.00.
http://dx.doi.org/10.1145/2731186.2731188

1. Introduction

System virtualization has regained its popularity in recent years and has been widely used for cloud computing [15, 17]. It allows multiple guest operating systems to run simultaneously on one physical machine. The guest operating systems (OS) on such virtual machines are agnostic about the host OS and hardware platforms. System virtualization can be divided into same-ISA and cross-ISA categories depending on whether the guest and the host are of different instruction-set architecture (ISA). Same-ISA system virtualization is commonly used for server consolidation, for example, VMware Workstation [13], VMware ESX Server [23], XEN [8], KVM [25] and VirtualBox [6] support same-ISA system virtual machines. Cross-ISA system virtualization is also important and commonplace. For example, applications and OSes compiled for one ISA can run on platforms with another ISA. Recently announced virtual smart phones for iOS which run Apple iPhone applications on x86 based PCs are based on cross-ISA system level virtual machines. The Android Emulator [2] emulates the Android/ARM environment on the x86-64 platforms is yet another example, it offers great convenience in development and debugging to Android application developers. This paper focuses on the performance of cross-ISA system virtual machines.

QEMU [5] is a very commonly used cross-ISA virtual machine. Since the ISA of the guest (e.g. ARM) is different from that of the host (i.e. x86 based PC). Dynamic Binary Translation (DBT) is often used to speed up emulation [18]. One challenge in system virtualization is the large memory virtualization overhead where the virtual address of the guest must be mapped to the guest physical address, then the guest physical address be mapped to the host physical address during program execution (as shown in Figure 1(a)). For example, on average QEMU spends 23%$\sim$43% of the total execution time in memory translation for SPEC CINT2006 benchmarks when running in system mode emulation [14]. So optimizations to minimize such memory virtualization overhead are the key to enhance the performance of the system-level emulator.

Hardware-assisted memory virtualizations, such as Intel Extended Page Tables [19] and AMD Nested Paging [9], are effective ways to reduce this overhead for same-ISA virtual machines. But they do not support cross-ISA system virtualization. Software-based memory virtualization (e.g. Software MMU) has been used in existing system virtual machines, such as QEMU. However, software MMU is one major contributor to sluggish simulation. Some recent approaches, such as the Embedded Shadow Page Table (ESPT) [14] exploits a combination of software MMU and hardware MMU to significantly cut down the memory virtualization cost. ESPT utilizes the larger address space on modern 64-bit processors and creates a loadable kernel module (LKM) to embed the shadow page entries into the host page table. Those shadow page table (SPT) entries are used to store the mapping between guest virtual address and host physical address. This table can be used by the hardware walker to resolve TLB misses. In [14], ESPT has achieved significant speed up ($>$50%).

However, the original ESPT work has a few drawbacks. For example, its implementation relies on a LKM to manage SPTs. Using LKMs is less desirable for system virtual machines due to portability, security and maintainability concerns. For instance, 1) most of LKMs use the internal kernel interface and different kernel versions may have different interfaces. For example, there have been 47 Linux kernel versions after version 2.0 and 12 of them had updated MMU-related modules. If the LKM approach is adopted to manage ESPT, we would have to update the kernel modules of ESPT management for 10 out of the 12 kernel versions. So it would be difficult to have one LKM supporting all kernel versions. 2) To enforce security, modern OS only allows the user who has root privilege to load LKMs. So the original ESPT can only be used by the privilege users. We believe it is important to allow all users to have access to system virtual machines. This is the case for many Android application developers. 3) Using LKMs, the kernel would be less secure. For example, for the Linux kernel, many kernel exploits have been reported, and often these exploits attack LKMs instead of the core kernel [16]. CVE is a list of information security vulnerabilities and exposures provided for public [3]. In the 306 Linux kernel vulnerabilities listed on CVE from January 2013 to November 2014, 112 vulnerabilities are located in the core kernel, while 194 vulnerabilities are in LKMs or drivers.

The main reason for using LKMs in ESPT is to operate on the SPT which is created and maintained at privileged level. We divided the operations on SPT into three types: creating SPT, synchronizing with guest page table (GPT) and switching SPT for different processes. The application scenario of the first type is that when the guest creates a new process, we should create a new SPT accordingly. The second is that because SPT is the shadow of the GPT, we must synchronize the SPT with the GPT to ensure consistency. The third is that when the guest switches process, the emulator should also switch to its corresponding SPT. Based on these three types of operations, we have come up with a different implementation to manage SPTs without using LKMs. To distinguish our new implementation from the original ESPT, we call our approach “Hosted Shadow Page Table” (HSPT). HSPT uses three methods to accomplish these operations with no LKMs. First, it uses a portion of the host page table as SPT to accomplish the operation of creating SPT. Second, it uses the shared memory mapping scheme where multiple virtual pages can be mapped to the same physical page to synchronize the SPT with GPT. Third, it uses Shared SPT to handle multi-processing in guest OSes. As for Shared SPT, it has also investigated on three variations for performance improvements.

The main contributions of this paper are as follows:

1. Proposed a practical implementation of ESPT to speed up cross-ISA system virtual machines without using loadable kernel modules thus avoid the accompanying portability, usability and security problems.
2. Proposed an efficient synchronization mechanism between SPT and GPT based on shared memory mapping methods.

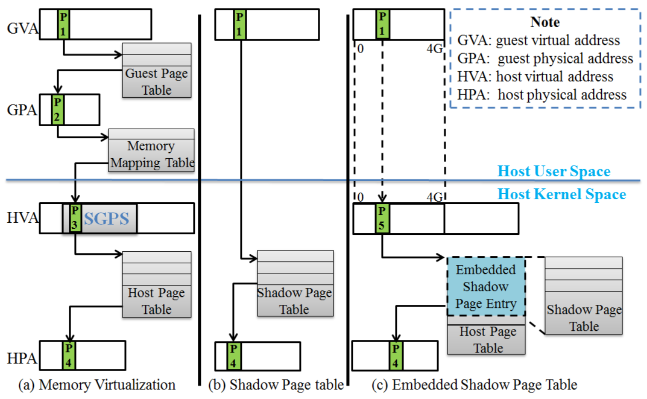
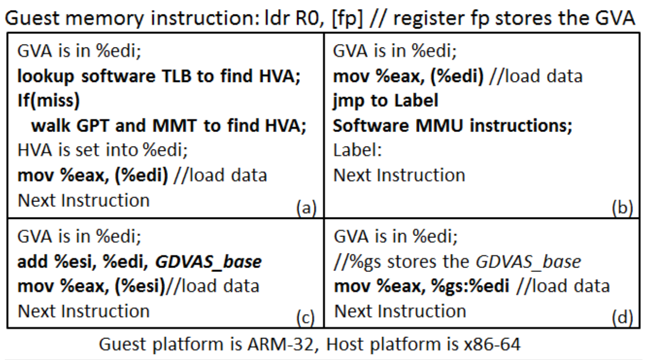

GVA
GPA
HVA
HPA
P1
P2
P3
P4
P5
Guest Page Table
Memory Mapping Table
SGPS
Host Page Table
Shadow Page Table
Embedded Shadow Page Entry
Host Page Table
0
4G
Note
GVA: guest virtual address
GPA: guest physical address
HVA: host virtual address
HPA: host physical address
Host User Space
Host Kernel Space
(a) Memory Virtualization
(b) Shadow Page table
(c) Embedded Shadow Page Table

Figure 1. Address translation of (a) Traditional memory virtualization, (b) Shadow Page Table and (c) Embedded Shadow Page Table


Guest memory instruction: ldr R0, [fp] // register fp stores the GVA

GVA is in %edi;
lookup software TLB to find HVA;
If(miss)
  walk GPT and MMT to find HVA;
HVA is set into %edi;
mov %eax, (%edi) //load data
Next Instruction (a)
	GVA is in %edi;
	mov %eax, (%edi) //load data
	jmp to Label
	Software MMU instructions;
	Label:
	Next Instruction (b)

GVA is in %edi; **add %esi, %edi, *GDVAS_base*** **mov %eax, (%esi)**//load data Next Instruction (c)	GVA is in %edi; //%gs stores the *GDVAS_base* **mov %eax, %gs:%edi** //load data Next Instruction (d)

Guest platform is ARM-32, Host platform is x86-64

Figure 2. Target translation codes of guest memory access instruction with (a) Software MMU, (b) ESPT and HSPT. Target translation codes in (c) HSPT are generated by general translation method. Target translation codes in (d) HSPT are generated by optimized translation method.

3. Proposed and evaluated three SPT organizations, including Shared, Private and Group Shared SPT to handle multi-processing in guest OSes. A guideline on how to select each variation is provided.
4. All schemes proposed and studied have been carefully implemented and evaluated on a system emulator which emulates the Android/ARM environment on the x86-64 platform. Experiments with SPEC CINT2006 benchmarks and multiple practical Android applications show that our technology can achieve comparable performance as the original ESPT while making the system emulator more portable, secure and maintainable. With our new SPT organization varations, our approach can efficiently handle applications with multi-processing.

The rest of the paper is organized as follows: Section 2 gives our motivation; Section 3 focuses on the framework and details of HSPT; Section 4 presents the settings of our experiment and results. Section 5 briefly discusses related work and Section 6 concludes this paper.

2. Motivation

Memory virtualization is an important part of system virtualization. In a hosted system virtual machine, such as KVM [25] based virtual machines, multiple guest OSes and the host OS share the same machine physical memory and are isolated from each other. Consider the x86 platform, as an example, if running as a standalone machine, the OS needs provide the page table which maps the Virtual Address (VA) to the Physical Address (PA). The hardware MMU uses TLB to speed up page table lookup. When a TLB miss occurs, a hardware table walker searches the page table to find the page table entry, and insert it into the TLB. If running on a virtual machine, the PA of a guest machine is not the true machine PA. It is actually mapped to the virtual address of the host machine, so another level of translation is required. Figure 1(a) shows the address translation of traditional memory virtualization. A virtual machine usually allocates a large chunk of virtual space to simulate the guest physical memory, we call this space "Simulated Guest Physical Space" (SGPS) (as indicated in the figure). When the guest accesses the machine memory, it goes through three-level of address translation. It works as follow: When the guest code accesses a guest virtual page P1, it goes through the guest page table (GPT) to obtain the guest physical page P2 and then look up the internal memory mapping table (MMT) which is created for recording the location of guest physical memory at the host virtual address to find the corresponding host virtual page P3. Next, it searches the host page table (HPT) to obtain the host physical page P4. After these three steps, the guest can finally access the machine memory.

For same-ISA system virtual machines, a Shadow Page Table (SPT) [10] is often created, which maps the guest VA to the machine physical address. When a guest process is running, the hypervisor switches the HPT to this SPT so that only one level of address translation is performed. Figure 1(b) shows the traditional SPT. When running the guest code, the page table base pointer, such as the CR3 register in x86, will be changed to the SPT and then it can directly use hardware MMU to accomplish the address translation.

For cross-ISA system virtual machines, such memory translation process is simulated in software, so it is often called "software MMU". QEMU [5] uses software MMU to translate a guest virtual address (GVA) to the host virtual address (HVA), and let the host machine to handle the HVA translation afterwards. In QEMU, a software-TLB contains the recently used map from GVA to HVA. Each memory access instruction of the guest is translated into several host instructions by the internal dynamic binary translator [5] of QEMU. Suppose ARM-32 is the guest and x86-64 is the host, an ARM memory load instruction "ldr R0,[fp]" is translated into several host instructions with software MMU, as shown in Figure 2(a). These generated host instructions are used to search the software-TLB and can get the target

HVA if hit. If miss, the GVA will go through the aforementioned three-level of address translation (i.e. P1 to P2, P2 to P3 and P3 to P4). In other words, to emulate a single memory access instruction of the guest, 10 to 20 native instructions will be executed when the search hits in the software-TLB or hundreds of instructions to execute when it misses. However, the advantage of using software MMU is platform and OS independent, thus more portable.

Embedded Shadow Page Table (ESPT) proposes to adopt the SPT approach from same-ISA virtual machine. Adopting SPT directly in cross-ISA machines is much more difficult than it initially looks. This is because the guest architecture is emulated. For example a guest register is emulated as a host memory location, so when running the guest code, it needs to access both the guest addresses and the host addresses in the same translated code block, each requires a different page table. So frequently switching back and forth between SPT and HPT is needed. This excessive overhead of page table switching is intolerable, therefore, ESPT proposes to embed the SPT into the HPT to avoid such table switching [14]. In ESPT, LKMs are used to create and manage embedded shadow page entries in HPT (as shown in Figure 1(c)). The consistency between GPT and SPT is also maintained by using LKMs. ESPT first sets all the 4G memory space dedicated for the shadow page entries as protected, when certain pages are accessed, SIGSEGV will be triggered to invoke a registered signal handler. In the signal handler, ESPT will use the fault GVA to go through the aforementioned three-level of address translation to find HPA and then create the mapping from GVA to HPA into the SPT. Finally, it resumes from the fault instruction. ESPT maintains a SPT for each guest process, when the guest process switches, ESPT will use LKMs to set the host directory page table base pointer for the lower 4G space to the targeted SPT. For example, Figure 1(c) shows that "Embedded Shadow Page Entry" points to a SPT, when the guest process switches, ESPT will set "Embedded Shadow Page Entry" to point to the guest's new SPT.

Taking ARM-32 as guest and x86-64 as host, as shown in Figure 2(b), we can see that ESPT will translate one guest access memory instruction to a 'mov', a 'jmp' and several Software MMU instructions. There are two types of page faults in this approach: one is shadow page fault and another is guest page fault. Shadow page fault occurs when the requested GVA is not in the SPT. It is handled by LKMs. A guest page fault is handled by Software MMU. Those instructions work as follow: the 'mov' instruction will first try to access the GVA by using SPT. If hit in SPT, the obtained HVA will go through the hardware MMU to execute and then jump to execute the next guest instruction. If miss in SPT, LKMs will be invoked to fill the SPT and resume execution, and if guest page fault occurs, ESPT will replace the 'jmp' with 'nop' and execute Software MMU instructions to fill the GPT.

ESPT can significantly reduce the address translation overhead. SPEC CINT2006 benchmark results indicate that ESPT achieves an average speed up of 1.51X in system mode when emulating ARM-32 on x86-64 and a 1.59X for emulating IA32 on x86-64 against the original software MMU [14].

ESPT and Software MMU both have their own strength and weakness. To avoid the shortcomings of relying on LKMs, we have come up with a different way to incorporate the idea of SPT. It is non-trivial to avoid using LKMs for two reasons: 1) Our new approach must be able to create and maintain SPT in kernel space without using LKMs; 2) Multi-processing emulation with the same address space must be supported by our method. The technical details of HSPT will be described in the following section.

3. The Framework of HSPT for Cross-ISA System Emulation

Similar to ESPT, our HSPT focuses on the widespread scenario where the virtual space of the host is larger than the guest, such as ARM 32bit to x86 64bit. This scenario is very common to system virtualization since the 64bit architecture is more powerful in both functionality and performance, making it an ideal host machine. Taking advantage of the larger host's address space, we can use a portion of the host page table as our SPT to avoid frequent switching between the shadow and the host page table. We also believe our approach could be applied to the scenario where the guest is an ARM 64bit architecture. The user address space of x86-64 is 256TB (48-bit). Although ARM 64bit also supports 64-bit virtual address, the current AArch64 OS only uses 39-bit of virtual address (512GB), so there are room left for embedding SPTs.

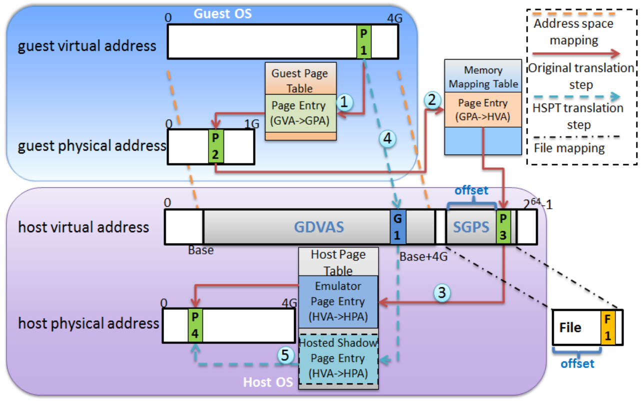


Figure 3. Address translation and the method of creating shadow page mapping of HSPT

Figure 3 shows the address translation process under HSPT. We set aside a fixed virtual space called "Guest-Dedicated Virtual Address Space" (GDVAS) from the host virtual space (as indicated in the figure). When we set aside GDVAS, each page in the guest virtual space is mapped to a page in the GDVAS. For example, the memory accesses of guest virtual page P1 would be translated to the accesses of host virtual page G1 in GDVAS (labelled as step 4). A traditional virtual machine would transform the accesses of P1 to the accesses of host physical page P4 after going through GPT, MMT and HPT (step 1, 2, 3) translation steps. In our approach, P1 is mapped to G1 with an offset which is also the base address of the allocated GDVAS (step 4). By mapping G1 to P4 which was mapped from P3 (step 5), the guest can also access data from G1. Thus we can convert the basic translation process from P1$\xrightarrow{GPT}$P2$\xrightarrow{MMT}$P3$\xrightarrow{HPT}$P4 to P1$\xrightarrow{+offset}$G1$\xrightarrow{SPT}$P4. The translation from P1 to G1 does not need table lookup, they only differ by a fixed offset. Compared with the basic memory translation, our approach can reduce three rounds of table lookup down to only once.

Figure 2(c, d) show the general and optimized translation where the optimized translation is only applicable to the platform with segment registers. Since there is only a constant offset from P1 to G1, when translating a guest memory access instruction, we just add the GVA (e.g. P1) to the base of GDVAS. As shown in Figure 2(c), an ARM memory instruction 'ldr R0,[fp]' is translated into 'add %esi, %edi, *GDVAS_base*' and 'mov %eax, (%esi)', where *GDVAS_base* represents the base of GDVAS. Moreover, when the host has the support of segment registers, we can utilize one segment register to hold the base address of GDVAS and optimize the translation. As shown in Figure 2(d), the segment register 'gs' is used to finish the first-layer address translation from P1 to G1. After that, hardware MMU will finish the translation from G1 to P4 automatically. HSPT based translation can reduce the address translation process from 10 to 20 native instructions down to only one or two, thus eliminating most of the address translation overhead.

So far, we have introduced the framework of HSPT, but there are still several issues to address:

1. We had described the memory accesses to guest virtual page P1 can be transformed to accessing the host virtual page G1. Section 3.1 will discuss how to create the mapping from G1 to P4.
2. SPT keeps the direct maps from guest virtual address to host physical address, so it must keep pace with the guest page table. Section 3.2 will discuss a mechanism to maintain the consistency between SPT and GPT.
3. When a guest process switch happens, both the traditional SPT and ESPT must also change the SPT. Section 3.3 will discuss three new SPT variations in our HSPT to support the guest multi-process.

3.1 Creating Shadow Page Mapping

As shown in Figure 3, creating shadow page mapping means creating the mapping from G1 to P4 into the SPT. After basic translation process, we know that P3 is mapped to P4. What we need to do is to make G1 to share P4 mapped from P3. HSPT uses a mechanism which maps two or more virtual pages to the same physical page to accomplish this shared operation. In the virtual machine initialization phase, the virtual machine allocates a host virtual space which is the so-called "SGPS" space (as indicated in the figure) and used as the guest physical space. When doing this, we use the "**mmap**" system call with 'MAP_SHARED' flag to map this space to a file. Then when the virtual machine starts to run and needs to create the SPT entry for G1, what we need to do is to map G1 with proper protection to the same host physical page P4. This is done by using the "**mmap**" system call and map G1 to the same offset F1 of the target file with P3. After this, the host OS will automatically map G1 and P3 to the same host physical page P4 with isolated page protection.

3.2 Consistency between HSPT and GPT

To make sure each GPT has a correct SPT and the entries in SPT are not outdated, we need to capture the creation and modification of GPT and then set the respective SPT mappings. The main reason for GPT changes include: 1) creation of a new GPT by guest OS due to initiating a new process; 2) modification of GPT by guest OS. These two scenarios are discussed respectively in the following sections.

3.2.1 Guest Page Table Created by Guest OS

When the guest OS creates a process, we need to correspondingly prepare a SPT for it. Since there are so many entries in each guest page table and only a small portion of these entries are actually used, it would be inefficient to synchronize every entry at this time. Therefore, we adopt a lazy synchronization approach which uses the SIGSEGV signal to inform which entry needs update and is similar to ESPT.

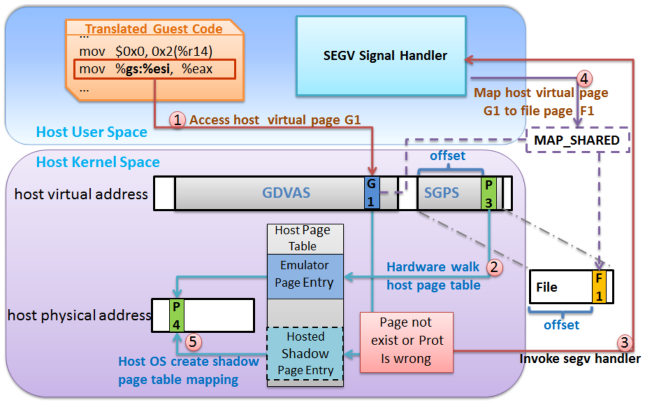

```
...
mov  $0x0, 0x2(%r14)
mov  %gs:%esi,  %eax
...
```

Figure 4. Signal notification mechanism for creating HSPT and synchronizing shadow page entries

Signal based Lazy Synchronization When a new SPT is created, we do not create mappings for all the entries. Instead, we only set the page protection value of all the entries as 'NONE'. Thus when a shadow page entry is accessed at the first time, the SIGSEGV exception will be raised and we

Algorithm 1 SEGV signal handler

```
Input:
    Faulted host virtual address, $SegvAddress$;
    The base of current GDVAS, $BaseAddress$;
    The current guest page table, $GPT$;
    The Memory Mapping Table, $MMT$;
Output:
    None;
 1: Host virtual page $G1 = SegvAddress\&PAGE\_MASK$;
 2: Guest virtual page $P1 = G1 - BaseAddress$;
 3: if Is guest page fault then
 4:     Jump to execute guest exception handler code;
 5: else
 6:     Walk the GPT and look up the MMT to find host
        virtual page $P3$;
 7:     Find $P3$'s corresponding file page $F1$;
 8:     Map $G1$ to the same file page $F1$;
 9:     Return to restart the faulted insruction;
10: end if
```

have the chance to synchronize for this entry. The execution flow of the SIGSEGV signal handle is shown in Figure 4 and the algorithm of signal handler is shown in Algorithm 1. When the guest first accesses the host virtual page G1 (labelled as step 1), a SIGSEGV signal will be raised since the page is protected in SPT (step 2). The host kernel will throw a signal to the virtual machine process and the control goes into our previously registered SIGSEGV signal handler (step 3). The handler will first compute the guest virtual page P1 (indicated in signal handler) and then walk the GPT to decide whether it is caused by the guest page fault or the inconsistency of SPT. If it is caused by the guest page fault, the handler will jump to execute the guest exception related handler. Otherwise, this exception is caused by the inconsistency of SPT and we should use the synchronization method mentioned in Section 3.1 to map G1 (Step 4) and the host OS will update the SPT entry automatically (Step 5). After this, the signal handler returns and resume the fault instruction.

3.2.2 Guest Page Table Modified by Guest OS

When the guest OS is running, all the page tables of guest processes could be modified at any time. If we monitor the whole guest page tables' memory space, the overhead would be excessive. Therefore, we intercept TLB-invalidation instructions to capture which entry requires synchronization. This approach is similar to the mechanism used in ESPT.

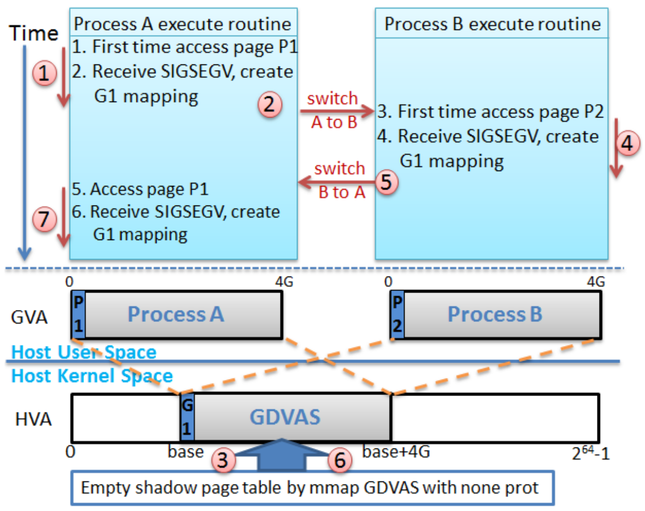
Intercepting the Guest TLB-invalidation instructions When the guest OS updates a page table entry, it should inform TLB to invalid the corresponding entry to ensure the consistency between the GPT and the TLB. Therefore, we can intercept TLB-invalidation instructions to capture the GPT updates indirectly. When we intercept these instructions, we do not modify the shadow page entry to the newest mapping, instead, we just clear the outdated mapping in SPT by using "**mmap**" system call with 'PROT_NONE' flag. When such pages are accessed, a SIGSEGV signal will be raised and we could synchronize at that time.

3.3 Support for Multi-Process

Multi-processing is very common in real Android apps. Without proper support for multi-processing, the cross-ISA system virtual machine using HSPT will have inadequate performance. We investigate three variations of SPT organizations: Shared SPT which makes all the guest process sharing one single SPT, Private SPT which provides a separate SPT for each guest process and Group Shared SPT which makes a group of the guest processes sharing a SPT. These three methods will be detailed in the following subsections.

3.3.1 Shared SPT

Shared SPT represents that all the guest processes uses the same SPT. It works as follows: When the emulator detects a guest process is switching out (this can be detected by monitoring the page table pointer), we should clear the SPT by using "**mmap**" system call with 'PROT_NONE' flag to get them ready for the next switched-in process. Figure 5 gives the illustration of this process. When guest process A first accesses a page, a SIGSEGV signal will be raised due to the inconsistency and we will update the corresponding entry in SPT(labelled as step 1). Then when the guest OS switches process A out and B in(step 2), we clear the SPT entries belong to process A to get ready for the new process B(step 3). After some time, process B would also be switched out(step 4) and process A back to active(step 5). Similarly, we clear the SPT entries of process B(step 6). When process A first accesses its SPT, SIGSEGV signals will be raised and SPT will be properly synchronized.



Time
Process A execute routine
1. First time access page P1
2. Receive SIGSEGV, create G1 mapping
switch A to B
Process B execute routine
3. First time access page P2
4. Receive SIGSEGV, create G1 mapping
switch B to A
5. Access page P1
6. Receive SIGSEGV, create G1 mapping
1
2
3
4
5
6
7
0
4G
GVA
P1
Process A
P2
Process B
Host User Space
Host Kernel Space
HVA
G1
GDVAS
base
base+4G
2^64-1
Empty shadow page table by mmap GDVAS with none prot

Figure 5. Shared SPT for handling guest multi-process

From this illustration, we can see that in the Shared SPT strategy, when a process is switched back again, the information of the filled SPT entries in the last timeslot is lost and the SPT of the switched-in process has to be warmed up again. Such repeated filling would result in a significant increase in the number of SIGSEGV and the runtime overhead. To reduce such repeated signal, we explored an optimization strategy called "Prefill Optimization".

Prefill Optimization Consider temporal locality that the entries accessed in the past may likely be accessed again in the future, we choose to prefill SPT entries when a process is switched in. Because the page table entries could be changed during the time of switched out, we only record which entries were accessed but not with full detailed information. Moreover, the number of filled entries will accumulate as the process keeps executing and only a small part of the entries filled may be accessed in this timeslot, so if we prefill all the past entries at the resumption of each process, the overhead incurred may exceed the benefit. Based on this observation, we set a window for each guest process. Each time a new SPT entry is synchronized, we'll record the index of this entry in the window. When the window is full, it is flushed to restart recording. After done this, when a certain process is switched in, we'll first synchronize all the entries recorded in this processs window. The impact of the window size to performance will be evaluated in the experiment section.

3.3.2 Private SPT

Prefill optimization can reduce the overhead of warming up in Shared SPT during frequent guest process switches. However, the prefill operation itself could be expensive. If the host machine has enough virtual memory space, the virtual machine could be benefit from Private SPT to avoid page table refill. As mentioned before, each SPT is bound to a separate GDVAS. Private SPT would consume a greate amount of virtual address space.

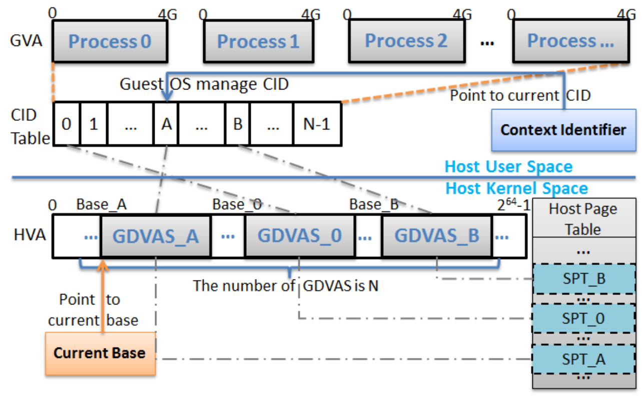


Figure 6. Private SPT for handling guest multi-process

Setting write-protection to the switched-out GPTs is a common but expensive method to monitor the modification of GPT [24]. To reduce this overhead, we again intercept the TLB-invalidation instructions to identify modified page entries. Consider x86 and ARM, as an example, they use PCID (Process Context Identifier) [4] and ASID (Address Space Identifier) [1, 20] respectively to identify TLB entries for each process. We call this kind of identifier as "Context Identifier" (CID). Same virtual address of different processes can be distinguished in the TLB due to CID. Based on this, when the process switching happens, there is no need to flush the whole TLB.

Principles of TLB structure with CID Under this structure, OS must obey the following principles: 1) At the time of process switching, CID register which contains the CID of the current active process should be modified. 2) When the OS modifies the page table, TLB must be informed with the CID and the address. 3) There are a limited number of available CIDs. We assume this number is N. So when the number of processes exceeds N, the sharing of CID between new processes and certain old processes would happen.

Management of Private SPTs Based on the principles above, we can tell which entry of a process, and the process CID is modified from the TLB-invalidation instructions. Therefore, when setting up each private SPT, we choose to bind each SPT with a CID (as shown in Figure 6) rather than the process ID so that we can use the TLB-invalidation instructions to maintain consistency between SPT and the corresponding guest page table.

Switching SPT As mentioned above, the CID register would be updated when the guest process switch happens. Since SPT is correlated with CID, we should also switch SPT at this time. Before describing activities involved in SPT switching, there is an issue needs to be addressed about the instruction translation which is shown in Figure 2(c). Since the translated instructions may be shared by different guest processes, when each process has a different SPT, in order to make each process access their own data through the same translated native instructions, we need to load the GDVAS base address of the current executing process before adding it to GVA. This base address can be stored in a global memory area and switched when the guest process is switched. Similarly, when the host has the segment register support, there is no need for this load operation and we can simply use the segment register to contain this base and modified this register when the guest process switches in. Based on the above, when the guest process switches what need to be done is only modifying the active GDVAS base address which can be kept in a global memory area or in the segment register on certain host platforms.

3.3.3 Group Shared SPT

Although Private SPT can avoid frequent SPT-clear problem of Shared SPT, it consumes too much host virtual space. Taking ARM as the guest, a virtual space size of 256*4G=1TB (upto 256 different processes allowed) would be needed. There may not be enough host virtual space. To address this problem, we proposed Group Shared SPT that the number of SPTs do not based on the available CID. Instead, we set up a fix number of SPTs depending on the size of host virtual space. Each time a process switches in, if this process already has a corresponded SPT, then we'll just use this SPT. Other-

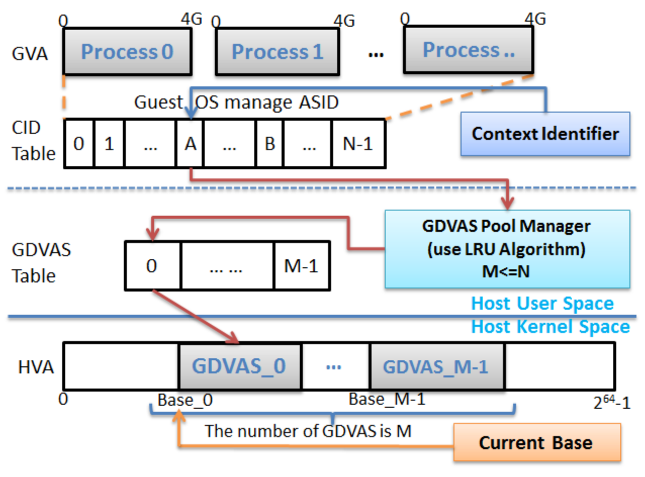


Figure 7. Group Shared SPT for handling guest multi-process

wise, if there's still an available SPT, it will be allocated. If not, we'll pick up one from the SPT pool based on certain strategies. This specific strategy is the LRU (Least Recently Used) algorithm which chooses the process which is least recently scheduled and clear its SPT for this new process. In Group Shared SPT, a group of processes share the same SPT. It is a compromise between Shared SPT and Private SPT. As long as the virtual space is sufficient, it works as Private SPT, when the virtual address space is short, it works like Shared SPT for a small number of processes.

Figure 7 shows the framework of Group Shared SPT. We can see that SPT is allocated from a GDVAS Table rather than correlated with CID as used in Private SPT. When the new switched-in process has no corresponding SPT and need to allocate a new one from the GDVAS table, we also adopt the prefill optimization used in Shared SPT. The performance under different number of SPTs will be evaluated in the experiment section.

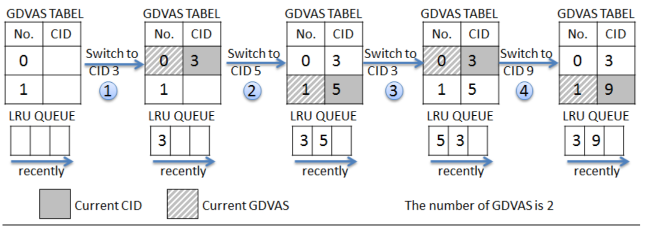


Figure 8. Example of managing shadow page tables with Group Shared SPT

Figure 8 shows an example of how Group Shared SPTs are managed during process switching. In this example, we assume the virtual machine only allocates two GDVASs from the host. When the guest runs the first process with CID 3, GDVAS_0 will be allocated to this process. The current base is set to Base_0 and the LRU queue is updated accordingly (labeled as step 1). Next, when guest switches to the process with CID 5, GDVAS_1 will be allocated to this process. The current base is set to Base_1 and the LRU queue is updated accordingly (step 2). Next, when guest switches back to the process with CID 3, the current base is set back to Base_0 and the LRU queue updated (step 3). Next, when the guest switches to another process with CID 9, no free GDVAS are available, so the virtual machine must select one GDVAS as victim. In this case, GDVAS_1 will be selected since it is least recently scheduled according to the LRU queue. Hence, GDVAS_1 will be allocated to this newly switched-in process (step 4).

4. Experimental Evaluation

4.1 Experimental setup

We implement HSPT based memory virtualization framework on an Android Emulator which uses a modified QEMU to emulate the Android/ARM environment on the x86-64 platform. The host experimental platform is Intel E7-4807 machine with 1064MHZ, 15G RAM, and Ubuntu 12.04.3 LTS(x86-64). We use Android 4.4(Kernel: 3.4.0-gd853d22nnk) as our guest and run test cases in guest VM.

Two suites of programs we selected to run as the guest applications are SPEC CINT2006 benchmarks and Android applications.

SPEC CINT2006 benchmarks SPEC CINT2006 is a CPU-intensive benchmark suite, stressing system's processor and memory subsystem. The source code of these benchmarks uses the GNU standard lib, but Android uses its own native lib. So in our experiments, we use static linked benchmarks. We only test the train input dataset instead of ref due to the limitation of available memory. Android Emulator uses QEMU which can only provide a maximum of 1GB of memory for guest when emulating the ARM platform, some benchmarks of CINT2006 with ref input require more than 1GB of memory during execution.

Android applications In order to test whether HSPT can enhance the actual user experience, we also test the Android system boot and applications start-up time. We select some generally used apps, including Clock, Calendar, Messages, Calculator, Phone and Contacts. The reason why we test the app start-up time is that running apps needs user interaction, it is difficult to test precisely. The startup latency is an important aspect to consider in many applications, especially for embedded systems. At the startup of an Android application, intensive interaction with the OS will occur to request for system resources. Many operations in the startup-phase might also get used afterwards. To avoid averaging with testing outliers, we ran each application test multiple times and take the average of three middle scores as the final result.

4.2 Evaluation of Shared SPT

4.2.1 Shared SPT without prefill optimization

Figure 9(a) shows the performance of the CINT2006 benchmarks. The x-axis is the benchmarks and y-axis is the execution time in seconds. Figure 9(b) shows the speedup. We can

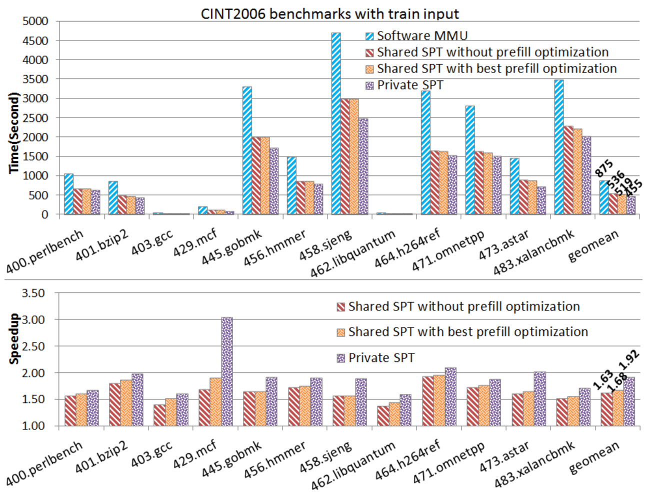


Figure 9. (a) Execution time and (b) Speedup by Private SPT and Shared SPT against Software MMU for SPEC CINT2006 benchmarks

see that our Shared SPT without prefill optimization can enhance the performance of Android Emulator for each benchmark and can achieve a 1.63X speedup on average.

4.2.2 Shared SPT with Prefill Optimization

In order to test the impact of the prefill optimization with different window sizes on the performance, we set ten values (shadow page entries) including 0, 100, 200, 300, 400, 500, 600, 700, 800 and 900 for window size. Figure 10 shows the speedup achieved by prefill optimization with different window sizes against Shared SPT without this optimization. The x-axis represents different window sizes and the y-axis is the speedup against shared SPT without prefill optimization. We can see that with increasing window size, the speedup of Shared SPT firstly keeps going up and then starts to decrease when the window size exceeds 300 entries. At window size 300, we have achieved a maximum speedup of 1.68X (1.033*1.63X).

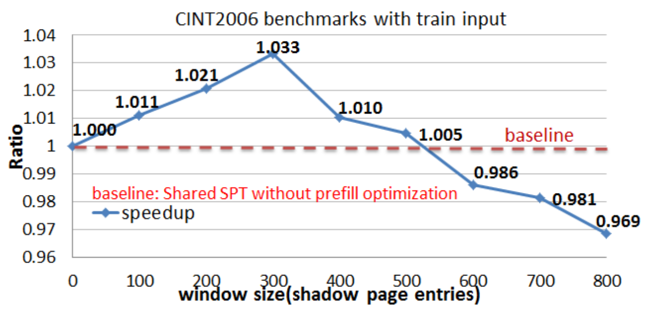


Figure 10. Speedup by prefill optimization with different window size against Shared SPT without prefill optimization

To explain the reason of the drop of performance when the window size exceeds 300, we did a profile of the number of SEGV exceptions for uninitialized page table entries and synchronized SPT entries. Figure 11 shows the result, the x-axis is different window sizes and y-axis is the ratio against window size 0 (that also represents the Shared SPT without prefill optimization). There are two types of SEGVs in HSPT that one is the SEGV due to uninitialized SPT entries and the other is raised by guest page faults. 'Uninitialized SEGV' represents the first type of SEGV and 'Synchronized Entries' represents the number of SPT entries synchronized including activities from both the prefill operation and the SIGSEGV signal handling. The 'Proportional decrease in Uninitialized SEGV' legend is computed by the following formula:

$$\frac{Base\ SEGV\ num\ -\ Current\ SEGV\ num}{Base\ SEGV\ num}$$

, where 'Base SEGV num' and 'Current SEGV num' represent the number of 'Uninitialized SEGV' under window size 0 and the current window size respectively. The legend of 'Proportional increase in Synchronized Entries' is computed similarly. We can see that the two ratios have minuscule differences when window size is less than 300. But when window size exceeds 300, the gap between these two ratios start to diverge which means that many prefilled SPT entries are not used before this process is switched out. Since the prefill comes with overhead, such unused prefilled entries will be a waste. When this overhead exceeds the profit, the performance goes down.

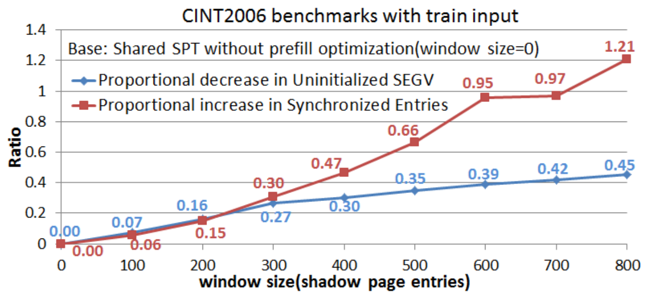


Figure 11. The number of Uninitialized SEGV and synchronized shadow page entries with different window sizes

4.2.3 Shared SPT for Android Practical Applications

Figure 12(a) shows the experimental result of Android system boot and applications start-up time. The x-axis is the Android system and applications, the y-axis is the boot time of system and the start-up time of applications in seconds. Figure 12(b) shows the speedup. The 'Shared SPT with best prefill optimization' is the prefill optimized Shared SPT when window size is 300. From this figure we can see that no matter whether we adopt prefill optimization or not, Shared SPT does not outperform the Software MMU for the system boot time and start-up time of each application. For practical Android apps, what is needed is Private SPT. Nevertheless, prefill optimization for Shared SPT clearly outperforms basic Shared SPT on both CINT2006 and practical apps.

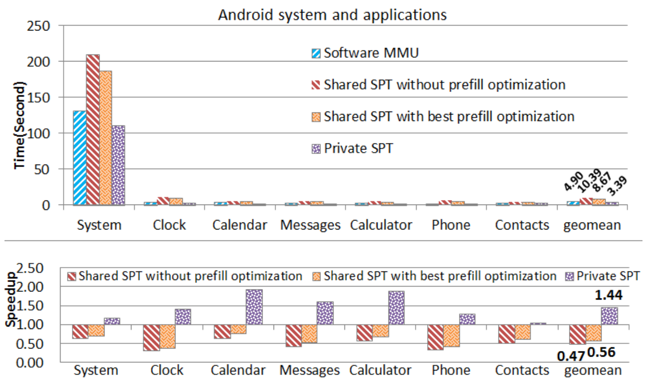


Figure 12. (a) Execution time and (b) Speedup by Private SPT and Shared SPT against Software MMU for Android system boot and applications start-up

The different outcomes between Figure 9 and Figure 12 come from the fact that each CINT2006 benchmark needs only one process but each Android app may need multiple service processes. So the performance of process switching dominates practical apps more than CINT2006 benchmarks. In Shared SPT, frequent process switching can causes frequent SPT-clear operations. So Shared SPT is not ideal for emulating applications with multi-processing.

4.3 Evaluation of Private SPT

4.3.1 Private SPT for SPEC CINT2006 benchmarks

From Figure 9(b), we can see that Private SPT achieves an average speedup of 1.92X against Software MMU. We can also see that compared with other benchmarks, 429.mcf has the largest speedup. In order to explain this, we also did an experiment that profiled the TLB miss rates of Software MMU. Figure 13 shows the result, the y-axis is the TLB-miss rate. We can see that the TLB miss rate of 429.mcf is higher than others. There are two main reasons of TLB miss in Software MMU that one is that software-TLB needs to be cleared each time a guest process switches out (it does not use the ASID feature of ARM TLB) and the other is that software-TLB is not big enough (it has only 256 entries). Because Private SPT does not need to clear the SPT when a guest process switches and each page in guest virtual space is covered in SPT, so it can effectively solve the problem of

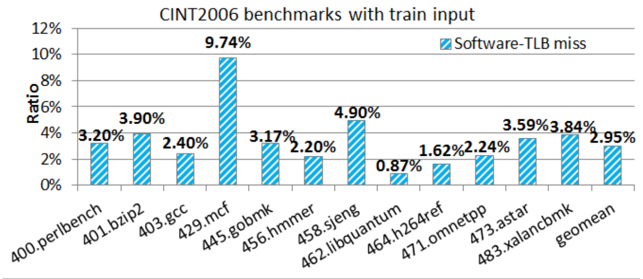


Figure 13. TLB miss ratio of Software MMU

high TLB miss rate and this explains why this benchmark can achieve a greater speedup.

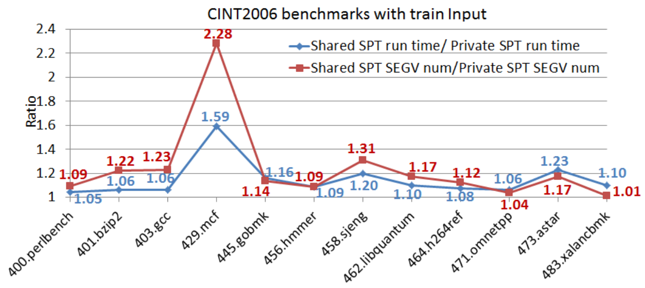


Figure 14. Impact on the performance of the total number of SEGV with uninitialized reason

From Figure 9(b), we can also see that 429.mcf achieves the highest speedup when comparing Private SPT against Shared SPT. To explain the reason, we give both Private SPT and Shared SPT a profile of the number of SEGV due to uninitialized reason. The results are shown in Figure 14. The y-axis is the ratio which represents statistics of Private SPT against Shared SPT with best prefill optimization. As we can see from the figure, the performance improvement tracks the same trend with the proportional decrease in Uninitialized SEGV numbers. The benchmark 429.mcf reduces most of the SEGV caused by uninitialized SPT entries so as to improve the performance the greatest.

4.3.2 Private SPT for Android Practical Applications

From Figure 12(b), we can see that compared with Software MMU, Private SPT has a significant performance boost of Android emulator and can achieve an average of 1.44X speedup for Android system and practical applications.

4.4 Evaluation of Group Shared SPT

We also evaluate the performance of Group Shared SPT with different number of SPTs against Software MMU. Group Shared SPT is implemented with the prefill optimization with 300 as the window size and we test the performance when the number of SPTs is 1, 4, 8, 16 and 32. Note that the more SPT used, the performance would be closer to Private SPT, but the more virtual space will be consumed. The fewer SPT used, the performance will be closer to Shared SPT.

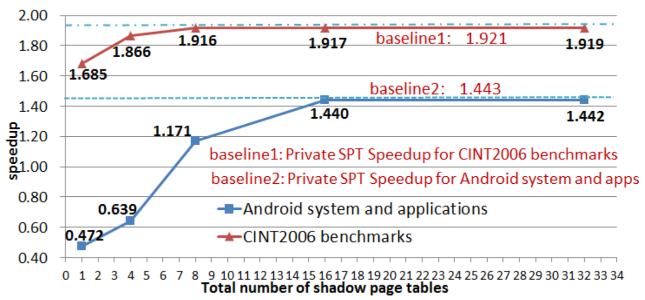


Figure 15. Performance of Group Shared SPT with different number of SPTs

The experimental result is shown in Figure 15. The x-axis represents different number of SPTs and the y-axis is the average speed-up against Software MMU. The performance is compared with the average speedup of Private SPT with 256 SPTs which is shown in the upper baseline. We can see obviously from the figure that the performance of CINT2006 benchmarks keeps improving with the increasing number of SPTs and when the number of SPTs exceeds 8, the performance of Group Shared SPT is very close to Private SPT. For Android system and applications, when the number exceeds 8, Group Shared SPT can perform better than Software M-MU and when the number exceeds 16, Group Shared SPT becomes competitive with Private SPT.

We can see that for CINT2006 benchmarks and Android practical applications, 8 and 16 of SPTs would be sufficient to obtain the performance benefit of Private SPT without suffering insufficient virtual address space. Since the Android apps need more service processes while running, they need a larger number of SPTs than CINT2006 benchmarks.

4.5 Discussion

The observed performance of HSPT on CINT2006 is similar to ESPT [14]. We did not compare the performance of HSPT side-by-side with ESPT since we need to implement the LKMs in order to reproduce the performance numbers of ESPT in an identical experimental setup. This work does not claim HSPT will yield greater performance than ESPT, it is motivated for better platform portability, higher system security, and improved usability for application developers since non-root users can also benefit from HSPT technology. Another major difference is that we noticed the importance of synchronizing SPT for multi-processing in Android environment. While Private SPT works well for multi-processing, it may take too much virtual address space. We come up with a Group Shared SPT approach which preserved most of the performance gain from Private SPT without its downsides.

In this paper, we only introduce the implementation of H-SPT on Linux OS. Actually, our HSPT can be implemented on any host OS with shared memory mapping capability. For example, since Windows has an API "**CreateFileMapping**" [7] to implement the shared memory mapping, HSPT can also be implemented on Windows OS.

5. Related Work

The primary memory virtualization methods for same-ISA system-level virtual machine include the software-based method such as Shadow Page Table (SPT) and the hardware-assisted method such as Intel Extended Page Tables [19] and AMD Nested paging [9]. SPT [10] has been widely used, it maps GVA directly to HPA to avoid levels of virtual address translations. The method to improve the performance of emulator used by hardware-assisted memory virtualization is mainly on two dimensional (2D) page walks [10] which uses hardware to walk both guest and nested page table to accomplish the address translation.

Xiaolin Wang and Jiarui Zang show that neither hardware-assisted method nor SPT can be a definite winner. SPT will result in expensive VM exits whenever there is a page fault that requires synchronization between the guest and shadow page tables. Hardware assists can solve this problem well, but compared with SPT, it has a disadvantage that the page walk yields more memory accesses and thus longer latency to resolve TLB misses [21, 22]. So they propose a dynamic switching mechanism between these two methods [24].

To reduce the overhead of many memory references to resolve TLB misses by 2D page walkers, some architectures use a page walk cache (PWC), which is an extra hardware table to hold intermediate translations [9, 12]. Jeongseob Ahn and Seongwook Jin discover that nested page table sizes do not impose significant overheads on the overall memory usage. So they propose a flat nested page table to reduce unnecessary memory references for nested walks. Further, they also adopt a speculative mechanism to use the SPT to accelerate the address translation [11]. Compared with SPT, they eliminate the shadow page table synchronization overheads.

There are two memory virtualization technologies for cross-ISA system virtualization. One is ESPT [14] and another is Software MMU such as the one used in QEMU [5]. ESPT embeds shadow page entries into the host page table (HPT) to avoid frequent page table switching between SPT and HPT. Software MMU designs a software-TLB which contains the recently used map from guest virtual address to host virtual address. Each memory instruction of the guest is translated into several host instructions to search the software-TLB. If hit, the mapped host virtual address can be used, if miss, the emulator must go through the three-level address translation. The advantage of Software MMU is platform portability. ESPT can reduce the software-TLB search time since the SPT can directly work with the hardware MMU. However, ESPT uses LKMs to manage the embedded SPT, and the use of LKMs decreases platform portability, system usability and introduces possible security concerns [16]. To combine these two methods' advantage, we propose HSPT which exploits all the advantages of ESPT with no LKMs.

6. Conclusion

In this paper, we proposed an practical implementation of Shadow Page Table (SPT) for cross-ISA virtual machines without using Loadable Kernel Modules (LKMs). Our approach uses part of the host page table as SPT and rely on the shared memory mapping schemes to update SPT, thus avoid the use of LKMs. Our implementation and management of the SPT avoids some shortcomings of using LKMs, making the virtual machines more portable, usable and secure. When test the Android emulator, we noticed the importance of reducing SPT update cost. Initially, a Shared SPT was used. We introduced a prefill optimization to cut down the cost of SPT updates during process resumptions. However, the best way to support multi-processing is to use Private SPT. S-ince Private SPT consumes too much virtual address space (VAS), it could fail in cases where there are many processes. So we come up with an elegant compromise, called Group Shared SPT where a group of processes could share the same SPT. With a given number of SPTs, limited by the available VAS supported by the platform, if the number of process-es is relatively small, each process can obtain its own SPT, thus enjoy the full benefit of Private SPT. When the number of processes increases beyond the number of SPTs, some processes must share a SPT. So Group Shared SPT works adaptively to balance between high performance and limit-ed VAS. With sufficient host virtual space, our approach has achieved up to 92% speedup for CINT2006 benchmarks and 44% improvement for the Android system boot and practical applications start-up on the Android emulator.

Acknowledgments

We would like to thank the anonymous reviewiers for their useful feedback. This research is supported by the National High Technology Research and Development Program of China under grant 2012AA010901, the National Natural Science Foundation of China (NSFC) under grants 61303051, 61303052, 61332009 and 60925009, the Innovation Research Group of NSFC under grant 61221062.

References

[1] *ARM®Architecture Reference Manual.*

[2] Android emulator. URL `http://developer.android.com/tools/help/emulator.html`.

[3] Cve. URL `http://www.cvedetails.com`.

[4] *Intel®64andIA-32 ArchitecturesSoftwareDeveloperManual.*

[5] Qemu emulator. URL `http://wiki.qemu.org/Manual`.

[6] Virtualbox. URL `http://www.virtualbox.org/`.

[7] Windowsapi. URL `http://msdn.microsoft.com/en-us/library`.

[8] Xen. URL `http://www.xen.org`.

[9] Amd-v nested paging, 2008.

[10] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGARCH Comput. Archit. News*, 34(5):2–13, Oct. 2006. ISSN 0163-5964.

[11] J. Ahn, S. Jin, and J. Huh. Revisiting hardware-assisted page walks for virtualized systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 476–487, 2012. ISBN 978-1-4503-1642-2.

[12] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. *SIGPLAN Not.*, 43(3):26–35, Mar. 2008. ISSN 0362-1340.

[13] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Trans. Comput. Syst.*, 30 (4):12:1–12:51, Nov. 2012. ISSN 0734-2071.

[14] C.-J. Chang, J.-J. Wu, W.-C. Hsu, P. Liu, and P.-C. Yew. Efficient memory virtualization for cross-isa system mode emulation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 117–128, 2014. ISBN 978-1-4503-2764-0.

[15] H. Chen and B. Zang. A case for secure and scalable hypervisor using safe language. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, 2012.

[16] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, 2011.

[17] J. Park, D. Lee, B. Kim, J. Huh, and S. Maeng. Locality-aware dynamic vm reconfiguration on mapreduce clouds. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 27–36, 2012. ISBN 978-1-4503-0805-2.

[18] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design).* 2005.

[19] R. S. M. A. B. K. L. Uhlig, Neiger and Smith. *Intel virtualization technology.* Computer, 2005.

[20] P. Varanasi and G. Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, 2011.

[21] VMware. Performance evaluation of amd rvi hardware assist. Technical report, Technical report, 2009.

[22] VMware. Performance evaluation of intel ept hardware assist. Technical report, Technical report, 2009.

[23] C. A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, OSDI '02, pages 181–194, 2002. ISBN 978-1-4503-0111-4.

[24] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li. Selective hardware/software memory virtualization. *SIGPLAN Not.*, 46 (7):217–226, Mar. 2011. ISSN 0362-1340.

[25] S. Yang. Extending kvm with new intel®virtualization technology. In *Intel Open Source Technology Center.*, 2008.