Small Change Big Achievements: Improving I/O Perfromance of All Peripheral Devices Using Page-Table Cache

Your N. Here
Your Institution

Second Name
Second Institution

Abstract

IOMMUs have been pervasively deployed on paravirtualized systems for the protection of the hypervisor and the security-critical data structures, e.g., the shared page tables. According to our observations, certain updates of the guest VM's page tables that are supposed to be orthogonal to the device I/O performance, would surprisingly lead to a large numbers of IOTLB misses. It implies that the I/O performances of all peripheral devices will be affected by the seemingly unrelated guest page table updates. Unfortunately, no existing work uncovers this dependece and adjusts the design of the paravirtualized hypervisor and the guest operating system.

In this paper, we are the first one to deeply demonstrate the impact upon the device I/O performance due to the page table updates. Then we propose IOSUP (I/O Speed-UP), a novel software-only approach for minimizing the negative impacts, without sacrificing the security of the hypervisor. By maintaining a page-table cache, IOSUP not only successfully decreases the IOTLB flushes, but also accelarates the creations and exits of user processes. We implement an prototype on Xen with Linux as the guest VM's kernel. We do small modifications of Xen (xxx SLoC) and Linux kernel version 3.2.0 (xxx SLoC). We evaluate the I/O performance in both micro and macro ways. The micro experiment results indicate that IOSUP is able to effectively reduce the flush rate of IOTLB from xxx to zero with less CPU usage, even when the page tables are frequently updated. The macro benchmarks shows that the I/O devices always produce better (or the same) performance, even when the system frequently generate many temporal processes.

1 Introduction

In paravirtualization [**?**, **?**], the kernel of each guest Virtual Machine (VM) and the hypervisor share the same virtual space and many security-critical data structures, like page tables and Global Descriptor Table (GDT). All these data structures are manageed by the guest VM, but their updates will be intercepted and validated by the hypervisor [**?**] with the purporse of preventing malicious accesses from the guest software. However, the paravirtualization technology itself is not armed with efficient protection to prevent DMA attacks [**?**]. To fix this gap, the hypervisor has to resort to the I/O virtualization (AMD-Vi [**?**] or Intel VT-d [**?**]) technology, which introduces a new Input/Output Memory Management Unit (IOMMU) to restrict DMA accesses on the physical memory addresses occupied by the hypervisor and the shared security-critical data structures. Leveraging the combination of the paravirtualization and I/O virtualization, the hypervisor prevent all malicious accesses from processor and hardware perpherial devices.

Problem. Although the combination of the paravirtualization and I/O virtualization brings strong security protection, but it also lead to the side effects on I/O performance, particularly on IOTLB flushes. Specifically, we surperisingly find out that certian updates on the guest page tables that are supposed to only affects the memory access from CPU have impacts on the address translation process of DMA access due to the additional IOTLB flushes. In addition, these kinds of updates are often triggered during the whole life cycle of a running system. As a consequence, the IOTLB flushing events are frequently triggered, which inevitably affect the I/O performance of all peripheral I/O devices.

This new depdency issue between guest page table and the DMA transferring urges us to reshape the design of the guest operating system and the hypervisor to minimize the effects on the I/O performance. Unfortunately, no one is aware this issue and also no existing work sloves this problem. In this paper, we aim to uncover this issue, and propose a corresponding solution to

Our approach rests on the observation that the high IOTLB flush rates experienced by a guest VM running are mainly generated by the creations and exits of page

tables.

In this paper, we focus on the improvement of I/O performance. Specifically, we aim to adopt guest OS kernel to further improve the I/O performance of all peripheral devices without sacrificing the security of the paravirtualized platforms. By deeply analyzing modern Xen hypervisor and Linux kernel, we surprisingly notice that the page table updates of guest OS could cause IOMMU to flush IOTLB. These flushes are necessary for the sake of the security of Xen hypervisor, but it inevitably increases the miss rate of IOTLB, and consequently reduces I/O performance, especially for the high-speed devices. Note that we are the first one to uncover this dependence between the security of paravirtualized (Xen) hypervisor and I/O performance. Based on this observation, we propose a novel algorithm that decreases the miss rate of IOTLB by carefully managing the guest page table updates, as well as retaining the security of paravirtualized hypervisor. We implement our algorithm with no modification of Xen and small customizations of Linux kernel version 3.2.0 by only adding xxx SLoC, and evaluate the I/O performance in micro and macro ways. The micro experiment results indicate that the new algorithm is able to effectively reduce the miss rate of IOTLB, especially when the page tables are frequently updated. The macro benchmarks shows that the I/O devices always produce better (or the same) performance, especially when the system frequently generate many temporal processes.

In particular, we make the following contributions:

1. We are the first to identify the
2. .
3. .

The rest of the paper is structured as follows: In Section **??** and Section 4, we briefly describe the background knowledge, and highlight our goal and the thread model. In Section **??** we discuss the design rationale. Then we describe the system overview and implementation in Section 5 and Section **??**. In Section 7, we evaluate the security and performance of the system, and discuss several attacks and possible extension in Section **??**. At last, we discuss the related work in Section **??**, and conclude the whole paper in Section **??**.

2 Background

As stated above, updating page types of guest OS leads to a number of IOTLB misses, and this is related to the security policies that paravirtualized (PV) hypervisor enforces. In the PV setting, the hypervisor is responsible for protecting and updating security-sensitive data structures so as to prevent illicit accesses from guest OSes and I/O devices. As Xen is typical and widely used today, this section mainly illustrates how Xen hypervisor [**?**] uses a x86 PV MMU model [**?**] to restrict OS-access while configures Intel IOMMU to restrict DMA-access, after which our motivation will be pointed out.

Since the model requires a familiarity with X86 address translation and related concepts, we need to understand the translation techniques first.

2.1 understanding of address translation

There are two address translation stages: 1) segmentation mechanism: logical address to linear address translation, which is related to GDT/LDT, and 2) paging mechanism: linear address to physical address translation, which is related to page table. In the following, we will describe the details of each stage.

2.1.1 logical to linear address translation

Generally, in the X86 architecture using segmentation [**?**], an instruction operand that refers to a memory location includes a value that identifies a segment and an offset within that segment. Each segment is represented by a Segment Descriptor that describes the segment characteristics, including segment base address, limitation of segment length and other meta-data. Segment Descriptors are stored in the Global Descriptor Table (GDT) or Local Descriptor Tables (LDT). Each logical address consists of a *segment* selector and *offset*.

To translate a logical address into a linear address, the processor does the following:

1. Uses the the segment selector to locate the segment descriptor for the segment in the *GDT/LDT* and reads it into the processor. 2. Examines the segment descriptor to check the access rights and range of the segment to ensure that the segment is accessible and that the offset is within the limits of the segment. If the offset within the segment is beyond the range specified by the *limit* field of the segment, the logical to linear address translation will stop as a hardware exception raise. 3. Adds the base address of the segment from the segment descriptor to the offset to form a linear address.

2.1.2 linear to physical address translation

To determine the physical address corresponding to a given linear address, the appropriate page table, and the correct entry within that page table must be located.

Since guest kenel is working in PAE mode on the 32-bit system in the model, we use PAE-based page table to describe paging mechanism. PAE mode has 3 levels of page tables. L1 is the bottom level, L2 is the middle level and L3 is the top level. A slot in L1 is Page-Table-Entry (PTE) slot, a slot in L1 is Page-Middle-Directory

(PMD) slot, and a slot in L3 is Page-Global-Directory (PGD) slot.

A given linear address is divided into 4 parts, b_0 through b_3. The b_3 bits (PGD slot offset) specify an entry in the PGD page, whose base address is stored in control register CR3. The b_1 bits (PTE slot offset) specify an entry in the PTE page, whose location is determined by the bits from b_2 bits, which specify an entry in the PMD page respectively. Finally, processor finds the physical address by adding the offset b_0 and the base address of the data/code page.

So far, the whole two-stage address translation is completed by hardware and thus physical memory is accessible to software. Also, to facilitate linear address translation speed, a Translation Lookaside Buffer (TLB) is used by MMU as a cache of page table entries.

Figure 1 illustrates the translation process when CPU accesses physical memory.

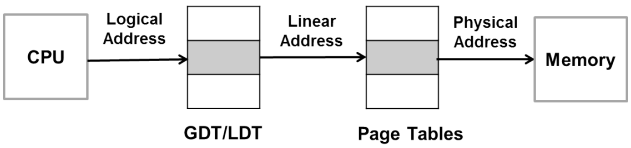


Figure 1: Address Translation

2.2 paravirtualised MMU model

In order to prevent guest OS from subverting system, Xen PV model sets GDT/LDT and page tables be read-only. To achieve this, WP bit in the CR0 register is set, and RW bit in the PTE slots is cleared, which point to GDT/LDT and page tables. Besides, x86 architecture supports four privilege levels in hardware, ranging from ring-zero to ring-three. Typically, OS runs in ring 0, the most privileged level to execute privileged instructions, while applications executes in ring 3, the least privileged level. In the PV model, since Xen is in ring-0, it needs to modify the OS to execute in ring 1. This prevents the guest OS from executing privileged instructions to remove the read-only permissions by updating CR0.WP and RW bit in PTEs.

On top of that, Xen still applies the address translation mentioned above to the de-privileged OS. Specifically, Xen allows a direct registration of guest page tables within MMU, so that the OS has direct access to machine memory by its own page tables as well as TLB, the so called direct-paging. In the meantime, Xen must also be involved in the management of updating guest page tables to prevent OS from arbitrarily modifying its page tables, otherwise OS will have a chance to access the machine memory space of Xen since they are sharing the same virtual memory space. Also, OS cannot update the GDT/LDT as well as the Interrupt Descriptor Table (IDT) directly. Therefore, Xen limits OS to read-only access in order to protect the critical structures, provides hypercalls for OS to explicitly submit all the update requests and then validates all the requests. Note that the OS is modified to be aware that a mapping between guest physical addresses and machine addresses (P2M table) so that it can writes new page tables using machine addresses before Xen validates them.

To aid validation, Xen associates each page with a page type and a type reference count. Types are defined as following: PGT_l1_page_table (used as an L1 page table), PGT_l2_page_table (used as an L2 page table), PGT_l3_page_table (used as an L3 page table), PGT_l4_page_table (only used as an L4 page table for 64-bit x86 system), PGT_seg_desc_page (used as an global descriptor table / local descriptor table), PGT_writable_page (a writable page). To ensure that every type is mutually exclusive and thus a given page could only have one type at every moment, the type reference count is maintained. That is to say, if the count does not drop to zero, the page type of a given page cannot be changed. Xen allows OSes to allocate and manage their own critical structures while it is only involved in updating them. Note that Xen maintains the IDT directly and provides OSes with a virtual one that is not associated with any special type like GDT/LDT.

Xen validates the update operations in a number of ways, such as no entry of a guest page table pointing to the hypervisor's private space. Updating entries in a critical table is a sub-operation of upating the whole table that involves a page type update. As can be seen in figure 2, every page type update is between writable and non-writable type. During the operation of page type update, Xen also needs to protect the critical guest table structures from being attacked by I/O devices.

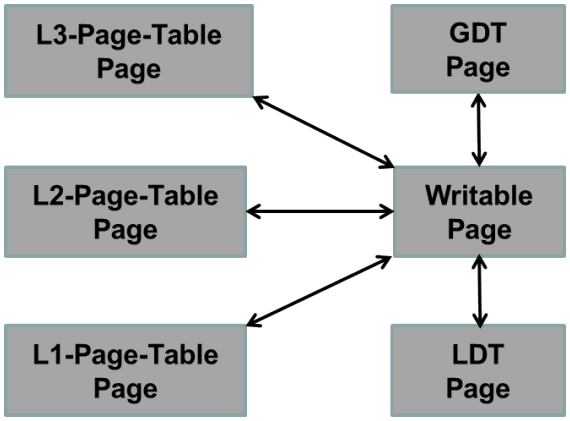
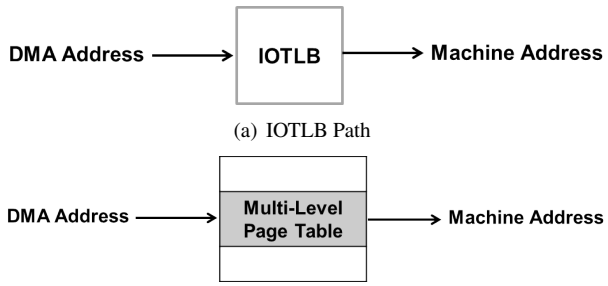


Figure 2: Page Type Updates



(a) IOTLB Path

(b) Page-Table Path

Figure 3: DMA Address Translation

2.3 DMA address translation

Intel IOMMU (i.e., Intel Virtualization Technology for Directed I/O) [**?**] mainly provides hardware support for DMA Remapping and Interrupt Remapping. DMA remapping is made use of by Xen to restrict access to particular memory area from all I/O devices. DMA remapping supports independent DMA address translation, indicating that if a specific device wants to access the machine memory through DMA, the access is via I/O page tables. Xen can utilize them to do a strict examination, such as checking if the access is permitted, working like the PV MMU model. In this case, if a device is allowed to directly access specified machine memory that belongs to a guest OS, then the device is referred to as the OS's assigned devices.

More specifically, in the case of a PCI device, a DMA request is composed of two parts: a request identifier that is used to index into a specific address translation structures (i.e., multi-level I/O page table) for a domain, and then a DMA address (regarded as a guest physical address in Xen) is transformed by the page table to its corresponding machine address. Here we are concerned about how DMA address is restricted. By configuring *read* and *write* fields in the I/O page table, Xen is able to enforce access-control such as mapping DMA addresses to writable pages while permitting read-only access to critical structures.

To facilitate translation speed of DMA addresses, frequently accessed page table entries known as the I/O translation look-aside buffer (IOTLB) are cached in hardware. If IOTLB hit occurs, a machine address is accessed quickly by IOTLB-path. If not, then the DMA-access has to go along the page-table path, as can be seen in figure 1. The page-table path definitely costs more time. As a result, IOTLB benefits the DMA address translation.

IOTLB entries are not static and Xen is responsible for explicitly flushes them when necessary. According to [**?**], IOMMU provides three types of IOTLB invalidation, i.e., global invalidation, domain-selective invalidation, page-selective invalidation, which differ in granularity. Intuitively, when a requested entry that corresponds to a specified DMA address needs to be invalidated, a page-selective invalidation is the best choice for the sake of performance. Besides that, IOMMU also supports two kinds of invalidation interfaces: register based invalidation and queued invalidation interface, between which queued invalidation performs better.

If IOTLB flushes frequently in whatever granularity, it will largely increases the probability of IOTLB misses, thereby badly affecting I/O performance. And this is where our motivation lies.

3 Motivation

Yueqiang says: Key Observation 1: Page type updates have impacts on the IOTLB miss, and thereby affect I/O performance Yueqiang says: To serve the first observation, we need 7 page types and their update graph. Any update needs a corresponding validation $->$ IOTLB flush.

Yueqiang says: Key Observation 2: The main source of the page type update is from writable-to-pagetable page update. Yueqiang says: To serve the second observation, we need 7 page types and their update graph. All other updates are only once in the whole life cycle, while the writable-pagetable page updates are frequent.

By analysing how Xen validates the update of page type, we have uncovered the relationship between page type update and I/O TLB flush . Here We use a specific update from writable to page-table to illustrate the relationship in detail.

Every time a guest OS launches a new process, it creates a new multi-level page table, multiple pages of which are allocated from the buddy system and initialized by OS. At the time, OS fills up the pages of writable-type with entries mapping virtual addresses to the machines addresses and submits update requests to Xen.

Since the pages are writable and their type reference counts are not zero at the moment, Xen firstly reduces the count to zero and then vets every entry in every level of page tables through a page table walk. If there exists an entry pointing to a machine page beyond the OS, then the update fails. If not, Xen will set and pin every page to its corresponding page-table type. The pinning mechanism is used to avoid performance cost. Specifically, each time Xen installs a new page table base pointer to the control register CR3 (i.e., context switch), it does not have to validate the page tables if they are pinned.

In the meantime, Xen must clear *read* and *write* permission fields in the I/O page tables corresponding to the machine frames of guest page tables and flush specified IOTLB entries. To achieve high performance while ensure safety, Xen requires multiple page-selective invalidations utilizing queued invalidation interface. Figure

4 describes the page type updates of machine pages between these two page types.

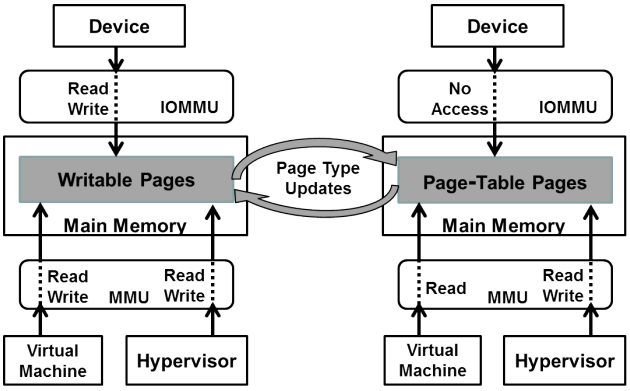


Figure 4: Page Type Updates Between Writable and Page-Table

From the description above, we have two key observations as follows:

1. Every page type update in figure **??** will force Xen to map or unmap a specific page from I/O page table and thus flush a corresponding IOTLB entry, which may have a bad impact on I/O performance.

2. Among all the page type updates, the update between writable and page-table is the main source of causing IOTLB-flush, since page table creation/destruction is very common during runtime while GDT/LDT creation/destruction is rare.

Based on the observations, we are motivated to propose IOSUP in order to reduce as many IOTLB flushes as possible for a maximum possible use of the IOTLB-path while retain the safety.

4 Problem Definition

4.1 Page Table Update in Bare-Metal Environments

5 IOSUP Overview

Yueqiang says: requirements: 1) minimum modifications (SLoC); 2) keeping security (we need a short subsection to discuss the security) 3) minimum effects on page table allocation/deallocation in terms of memeory and CPU usage Yueqiang says: mechanism: life cycle of pt cache (allocation, free) pt cache(mechanism) enable and disable Before we illustrate the design rationale of IOSUP, we first describe the desired requirements that IOSUP meets as follows:

R1) Retaining security. IOSUP aims at reducing IOTLB flush while guaranteeing existing security strength. R2) Achieving as good as possible performance. While benefitting IOTLB performance, IOSUP is supposed to achieve positive results in the aspects of CPU and memory usage. R3) Slightest possible modifications to Xen and Linux kernel for the sake of compatibility.

5.1 Design Rationale

As can be seen in figure 5, IOSUP introduces a cache flag to enforce fine-grained access control in order to ensure security without flushing IOTLB. Also, IOSUP proposes a novel cache mechanism to support the access control while facilitate the speed of every level of page table allocation/deallocation, saving CPU time while causing small impacts on memory usage.

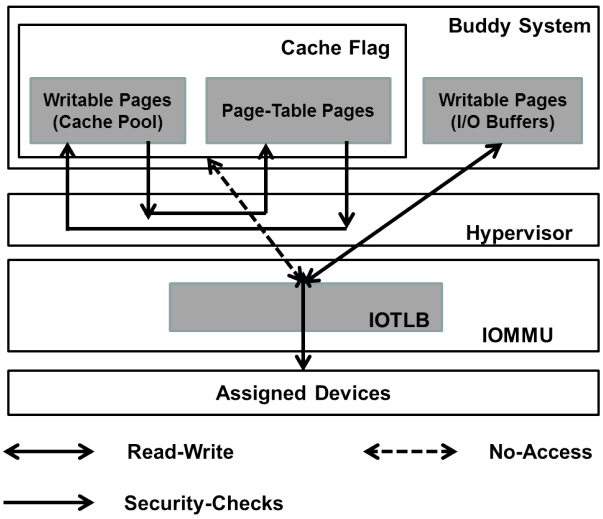


Figure 5: Overview

5.1.1 Fine-Grained Access Control

As revealed in previous sections, access control to writable pages is at a coarse granularity. Xen allows write-access both for guest OS and assigned I/O devices. Instead, IOSUP caches a certain number of writable pages, prohibits DMA access for the cached writable pages while OS still has its write-permission. If they are updated to be page tables, Xen only limits OS to read-only permission (software protection) without modifying I/O page tables as well as IOTLB. Specifically, Xen maintains a new flag called **cache** with a machine page (a cached page), indicating that the corresponding page is cached by guest OS and free from DMA-access. Whatever page type a machine page is, if it owns the flag, Xen neither maps nor unmaps it from I/O page tables, thus

avoiding an IOTLB-flush. Note that IOSUP only considers page type updates between writable and page-table.

For instance, if guest OS creates a new page-table, Xen firstly reuses the validation process to enforce software protection, and then checks if the page has the **cache** flag. If so, the only thing that Xen needs to do is to update the page to be a page-table. If not, Xen sets the page with the new flag, clears *read* and *write* permission fields in I/O page tables and flushes IOTLB. As for the guest page table destruction, Xen also reuses existing security checks to remove software protection while maintains DMA prevention, after which the corresponding page is (if without the flag) set with the **cache** flag and updated to be writable. Figure 6 describes the process.

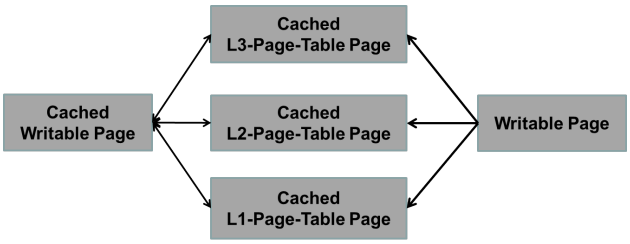


Figure 6: Page Type Updates with Cache Flag

As can be seen in Figure 6, every time page type updates between writable and page-table occur, every related page will own the flag, inaccessible to devices and all Xen needs to is to validate guest OS, having protected guest page tables and improved IOTLB performance, which satifies R1,2.

As writable pages with **cache** flag freed by guest OS cannot be mixed with ordinary pages in the buddy system, IOSUP builds up cache pools to manage freed writable pages in the OS space to function properly and serve as caches of page tables.

5.2 Cache Mechanism

If guest OS directly frees pages with cache flag to the buddy system, which may allocate them for DMA transactions. This will lead to unacceptable access faults. As a result, IOSUP enables OS to cache the writable pages for page-tables. Specifically, IOSUP defines new cache allocating and freeing functions for every level of page table to manage cache and whichever existing page-table function will invoke the corresponding cache function. By this approach, IOSUP reserves page-table caches. When dealing with process creation/destruction, OS allocates pages from or frees them into the cache pool instead of the buddy system, reducing time cost, thus meeting R2.

Cache mechanism begins its life-cycle either automatically with system booting up or dynamically during system runtime. IOSUP provides an interface for users to activate dynamically. Utilizing the feature, users can improve system performance in an on-demand way (e.g., too many process-creation at some time). Basically, if the mechanism is activated dynamically, there may exists a few writable pages without cache flag and this does not ruin the security while flush IOTLB only once. Also note that when every cache pool is empty initially or OS asks for more pages (e.g., many processes creations) than cache pool could supply, cache allocating function always requests pages from the buddy system, which also affect IOTLB once.

During the runtime of the cache, IOSUP also needs to control the memory size that caches take up. If the cache size becomes larger, it is necessary for cache freeing function to free pages from the pool into the buddy system. When to free pages in pool depends on two factors, namely 1) a proportion between pages in use and in pool, and 2) a total number of pages in use and in pool. If both factors exceed specific thresholds, respectively, corresponding cache flags must be cleared before the pages are freed. Thus, IOSUP defines a new hypercall for OS. When handling the hypercall, Xen mainly clears the flag of specified pages, maps freed pages in the I/O page tables and flushes IOTLB. It can be concluded that inappropriate thresholds will lead to the flushes of IOTLB. Because of that, IOSUP offers an interface for users to modify the default values of both thresholds in order to reduce IOTLB-flush or adjust cache size (meets R2). The default thresholds are determined by conducting experiments where no pages are freed to the buddy system (see details in section 7), and the page number to free is stated in the equation below: $\Delta num_to_free = num_in_pool - num_in_use$. If users have only a few available memory in extreme cases, they are provided with another interface to manually free all pages in cache into the buddy system, thus relieving system's pressure while badly affecting IOTLB. Using the two interfaces, users have full control of the page-table cache.

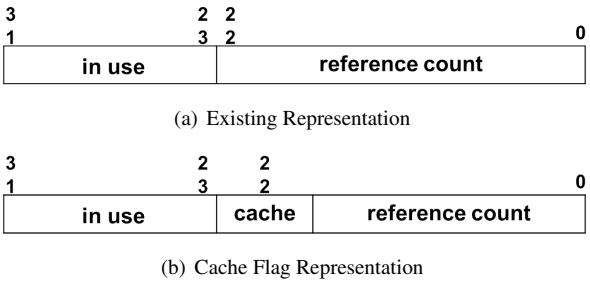
This is the life-cycle of a cached page, originating from the buddy system, freed into the cache, then maybe allocated by the cache and finally freed into the buddy system when necessary.

Section 6 will demonstrate its small modifications in details.

6 Implementation

Yueqiang says: implementations: 1) hooks for on-demand enable 2) locks for mechanism 3) data structures in mechanism 4) threshold selection for cache free? or this discussed in design 5)

Based on Xen version 4.2.1 and guest kernel version 3.2.0-rc1, IOSUP implements the access control related



(a) Existing Representation

(b) Cache Flag Representation

Figure 7: Field of Page Type Info

to cache flag as well as management of page-table cache.

6.1 Cache Flag

From a 32-bit field related to page type information, IOSUP was planning to use a redundant bit to represent the cache flag in order to be compatible with its original implementation. However, each bit in the upper 9-bit of the field has its specific use and the lower 23-bit serves as the reference count of current page type, representing at most $(2^{23} - 1)$ reference counts of the page type (see figure 11(a)). IOSUP enforces a stricter rule to prevent count overflow, as cache flag occupies the top bit of the 23-bit and the maximal reference count is limited to $(2^{22} - 1)$, which affects Xen little (see figure 11(b)). Actually, Xen hypervisor is functioning well with the flag bit since the reference count of a page type during runtime cannot reach at $(2^{22} - 1)$. On top of that, __get_page_type() is the critical function to be customized in order to check if a machine page has the flag bit.

6.2 Management of Page Table Cache

In the PV setting, guest OS is required to work in PAE (i.e., Physical Address Extension) mode. As a result, IOSUP maintains three levels of cache pools, each of which is essentially a structured single-linked list caching the guest physical addresses of pages. For the top two levels used for caching PGD (i.e., Page Global Directory) and PMD (i.e., Page Middle Directory), IOSUP obtains the physical addresses directly from their linear addresses. While the bottom PT (i.e., Page Table) is located in the High Memory, the mapping between its linear and physical addresses is not stable. IOSUP acquires the physical addresses of cached PT pages from the corresponding page-info structures.

Every list supports operations of both removal and insertion. Removal exists within the cache allocating function, which fetches a cached page from the top of the list. Insertion is located within the cache freeing function, which inserts a cached page also onto the top. Also, IOSUP maintains two global operation counters, namely num_in_use and num_in_pool, each logging the invocation times of removal and insertion, respectively. They are required when cache freeing function needs to free cached pages into the buddy system. Every pair of cache allocating and freeing functions is used to hook existing page-table related functions. By this way, whenever OS creates or destroys any process, it always interacts with the caches. Also note that every cache pool and operation counter are shared data resources and their related operation code are critical sections. IOSUP makes use of locks to ensure exclusive resource-access in the multi-processor setting. More specifically, Every level of page-table cache and all its operation counters share a spin lock and irrelevant operations should be removed out of the critical section, especially those that are time-consuming.

IOSUP implements a new system call to provide an interface for users to activate the cache in an on-demand way. In the system call, a global boolean variant is defined, which is initialized as false, indicating that cache is not enabled. And users can assign the variant as true to enable the cache. Anther two user interfaces related to modification of default thresholds and freeing all pages in cache are also implemented through system calls. For the modification, new thresholds of the proportion and total page number are passed as agruments while users can frees all cached pages by resetting a global boolean variant, which is initialized as true.

Within the cache freeing function, a hypercall is implemented with a single-linked list of cached addresses of pages as the parameter. The hypercall is a uniform interface for every level of cache freeing function, simplifying the modifications both to guest OS and Xen. As a reply to the hypercall, Xen mainly zeroes the flag bit of 32-bit field and invokes the important function interface intel_iommu_map_page to create entries in the I/O page tables as well as flush IOTLB.

7 Evaluation

We have implemented the proposed novel algorithm demonstrated in previous sections. The implementation adds 350 SLoC to Linux kernel and 166 SLoc to Xen hypervisor, of which 2 SLoc in the hypervisor are modified, which aims to build a cache pool for guest page tables so as to avoid unnecessary IOTLB flushes.

This section evaluates the performance of our algorithm by running both micro- and macro-benchmark kits.

7.1 Experimental Setup

Our experimental platform is a LENOVO QiTianM4390 PC with Intel Core i5-3470 running at 3.20 GHz, four

CPU cores available to the system. We enable VT-d feature in the BIOS menu, which supports page-selective invalidation and queue-based invalidation interface. Xen version 4.2.1 is used as the hypervisor while domain0 uses the Ubuntu version 12.04 and kernel version 3.2.0-rc1. In addition, domain 0 as the testing system configures its grub.conf to turn on I/O address translation for itself and to print log information to a serial port in debug mode.

Besides that, we have been aware that creating/terminating a process will give rise to many page table updates (e.g., from a page of Writable to a page of Page Global Directory), upon which function iotlb_flush_qi() will be invoked to flush corresponding IOTLB entries, and this is how a process-related operation affects IOTLB-flush. Thus, a global counter is placed into the function body to log invocation times of the function and then an average counter per minute is calculated which is called a frequency of IOTLB-flush. When the logged average counter drops to zero, it means that IOTLB does not be flushed any more, indicating that no process is created or terminated then.

Because of that, we define two different settings, classified by the frequency of IOTLB-flush.

Idle-setting: Actually, when system boots up and logins into graphical desktop, lots of system processes are created, causing many IOTLB flushes. But as time goes by, the frequency of IOTLB-flush reduces rapidly and stays stable to zero level ten minutes later, and we think that system starts to be in an idle setting, where no process creation/termination occurs and existing system daemons are still maintained.

Busy-setting: We launch a stress tool emulating an update-intensive workload to transfer the system from an idle setting to a busy one. Specifically, the tool is busy periodically launching a default browser(e.g., Mozilla Firefox 31.0 in the experiment), opening new tabs one by one and then closing the browser gracefully in an infinite loop, so as to constantly create/terminate a large number of Firefox processes, thus giving rise to frequent updates of page tables. More precisely, one iteration of the loop costs five minutes and thus the frequency of process creation/termination are 542.14 per minute and 542.07 per minute, respectively. Besides that, memory usage on an average iteration of the loop is 284.1 MB. Since the frequency of IOTLB-flush will become in a stable level five minutes after the tool starts to run, execution time length of one iteration is also set to the time interval.

Since page tables do not update in the idle setting, our algorithm can not play a big role in system performance. Both micro- and macro-benchmark kits are performed under the busy setting, in which micro-tests are utilized to evaluate the frequency of IOTLB-flush, CPU usage and memory size while macro-benchmarks give an assessment on overall system performance.

7.2 Micro-Benchmarks

To begin with, micro-experiments are conducted in two groups. In one group called cache-disabled, the "idle" system enters into the busy setting without the cache pool enabled. On the contrary, system state changes in another cache-pre-enabled group where the tool is invoked when the cache is already enabled since system begins to run.

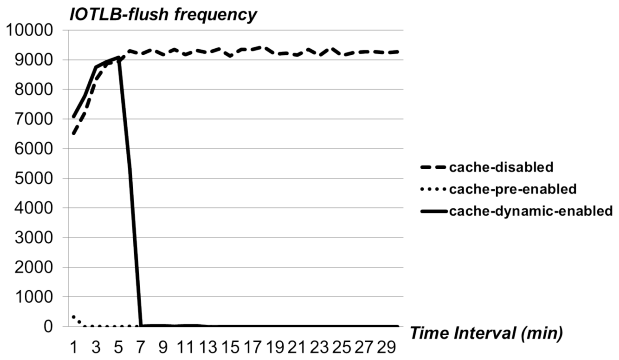
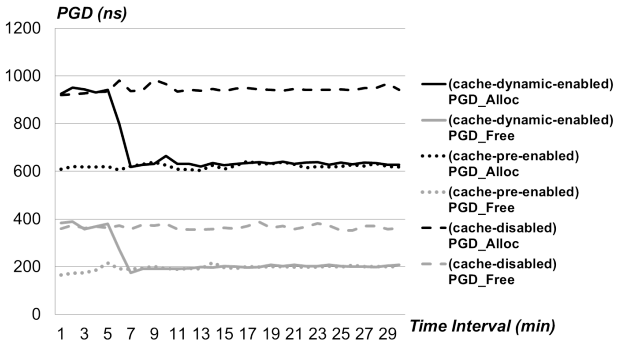


Figure 8: Frequency of IOTLB-flush

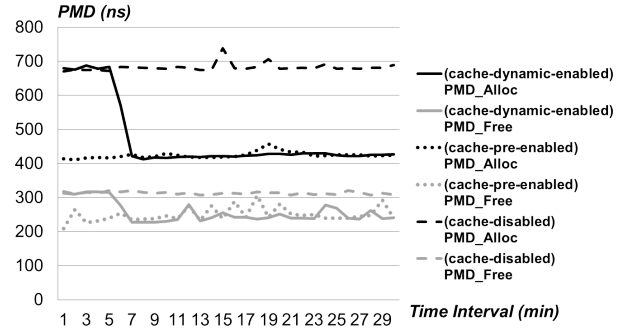
As can be seen from Figure 8. Y-axis represents the frequency of IOTLB-flush, corresponding to the time interval (i.e., one minute) of x-axis for the first thirty minutes that the running tool has taken up. From this figure, frequency in the cache-disabled group increases rapidly and remains stable five minutes later. By contrast, frequency in the cache-pre-enabled group drops to zero in a very short time and keeps zero level from then on. It can be safely concluded that our proposed algorithm does have a positive effect on reducing IOTLB frequency to zero quickly.

Now lets move to CPU usage that each group will take up. Specifically, each level of page table has its allocation functions and free functions, e.g., pgd_alloc() and pgd_free() and the execution time that every related function is calculated per minute. As a result, in Figure 9, allocation and free functions in three levels of page tables in the cache-pre-enabled group consumes 45.1% less and 70.9% less CPU time in nanoseconds, respectively, compared with that of the cache-disabled group, indicating that a process interacting with the pool has an advantage in saving time over one interacting with the buddy system.

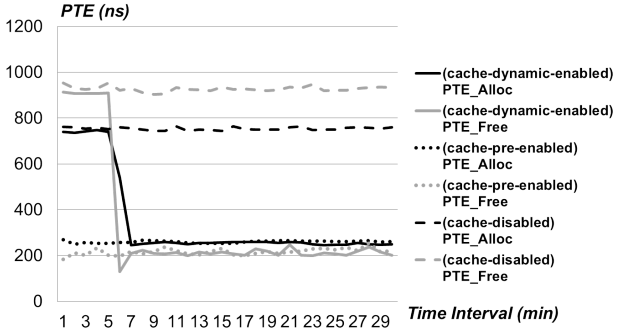
Besides CPU usage, the algorithm is evaluated in the aspect of memory size since three levels of cache pools have been built to support a fast process creation/termination. Cache pools in the cache-pre-enabled group from Figure 10 takes up 250 pages(i.e., (1000K $= 250 * 4K) < 1$M) at most in the long time run, only



(a) PGD

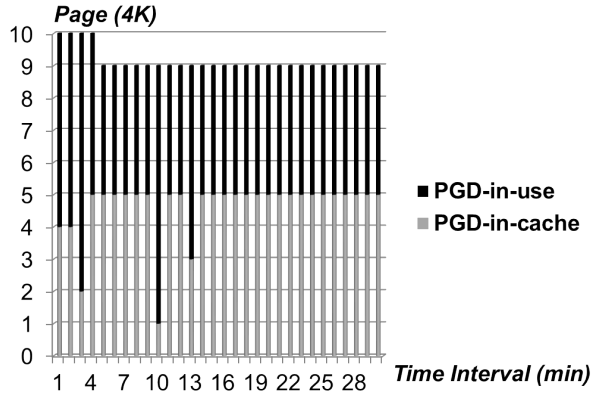


(b) PMD

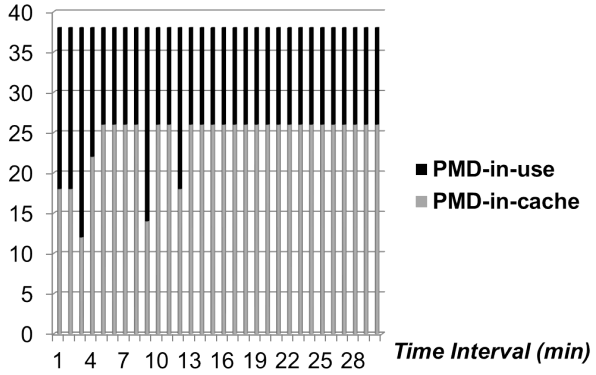


(c) PTE

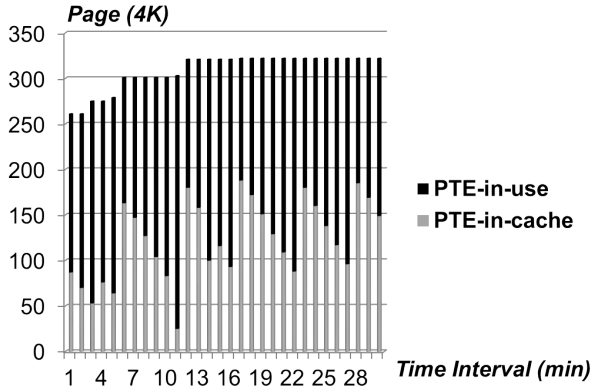
Figure 9: CPU Usage for Each Level of Page Table



(a) PGD



(b) PMD



(c) PTE

Figure 10: Cache Pools Size for Cache-Pre-Enabled Group

0.35 percentage of the tool's consumption, which is an insignificant usage and reaches a satisfying tradeoff between CPU time costs and space size.

But what if the memory percentage is too high? it is necessary to free pages from the cache pool to the buddy system. Pages in pool will be freed if 1) a proportion between pages in use and in pool, and 2) a total number of pages in use and in pool are greater. And data from group of cache-pre-enabled by default is referred to quantify the proportion and the total number. Actually, users can modify the two factors to adjust the cache pool size through an interface. On top of that, page number beyond the proportion is freed, stated in an equation below: $\Delta num_to_free = num_in_pool - num_in_use$.

Since pre-enabling the cache is not flexible enough, we also provide another interface for users to activate the cache mechanism in an on-demand way. For instance, system has been in a busy setting for a while and then cache is enabled manually. Users may make use of this feature to better improve system performance dynamically.

Next, in a group of cache-dynamic-enabled, cache is enabled when IOTLB-flush becomes stable while the freeing mechanism is added so as to check if this group behaves like the cache-pre-enabled group, i.e., cache-dynamic-enabled group could achieve a stable and low enough level in certain aspects, namely, frequency of IOTLB-flush, CPU usage as well as memory size, and it reaches to the level quickly.

From Figure 8 and 9, both frequency of IOTLB-flush and CPU usage in this group have a very similar trend with that of the cache-disabled group in the first five minutes, but reduces to zero quickly and stays stable, much like that of the cache-pre-enabled group. In addition, memory size that the cache-dynamic-enabled group in Figure 11 takes up is only 210 pages at most, also consuming a small percentage. And it is reasonable that the cache-dynamic-enabled group has less pages in the pool since a certain amount of page tables has been freed to the buddy system before the cache pool is put to use.

Also note that the expected results in group of cache-dynamic-enabled is obtained after sever trials by slightly modifying default values of the proportion and the total number. Since these two factors heavily relies on a specific scenario, they may not work for all. How to decide the factors is further discussed in future work.

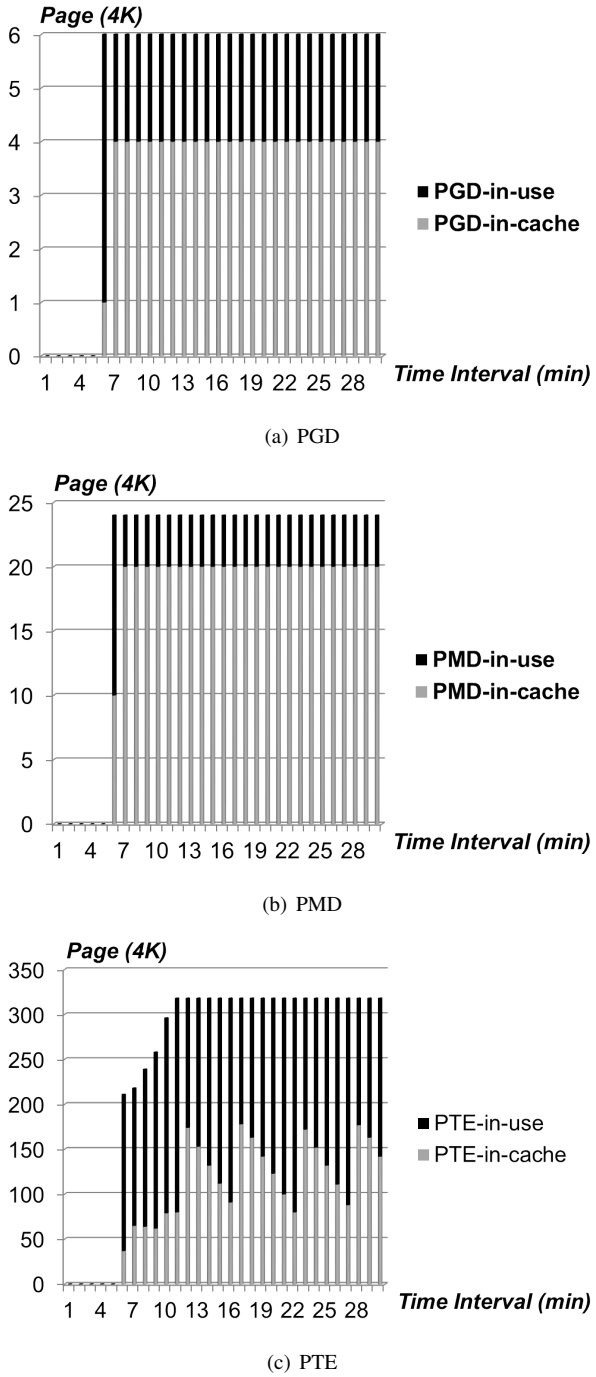


Figure 11: Cache Pools Size for Cache-Dynamic-Enabled Group

7.3 Macro-Benchmarks

Different micro tests have shown optimizations in three aspects for the algorithm while macro-benchmarks are made use of to evaluate its effects on overall system performance. Since cache-disabled group does not apply to real case, macro tests are conducted in two groups, i.e.,

cache-disabled group and cache-dynamic-enabled group.

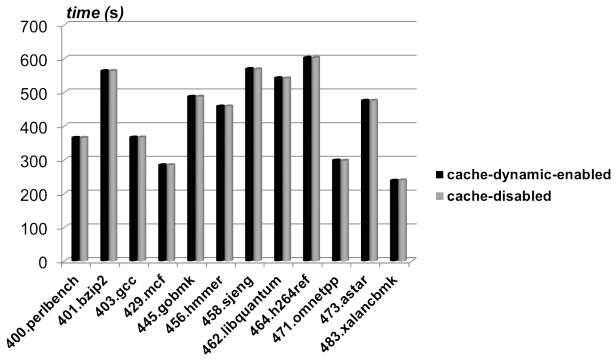


Figure 12: SPECint

SPECint_2006v1.2 has 12 benchmarks in total and they are all invoked with EXAMPLE-linux64-ia32-gcc43+.cfg for integer computation, results of which produce Figure 12. 483.xalancbmk in cache-dynamic-enabled group costs 239 seconds, 0.42% less than that of the cache-disabled group and this is the biggest difference among all benchmarks. As a result, little difference between the two groups exists, which indicates that the algorithm does not have any bad effect on system performance.

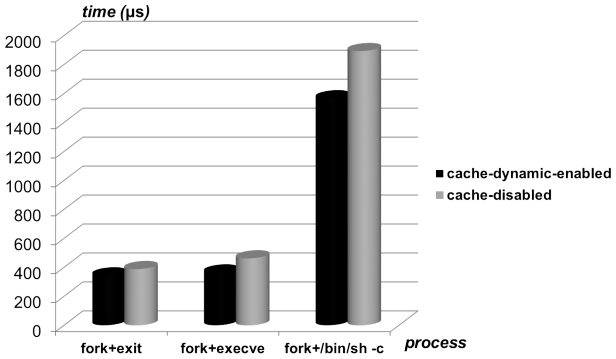


Figure 13: Lmbench

Lmbench is used to measure CPU time that processes cost (i.e., fork+exit, fork+execve, fork+/bin/sh -c), shown in Figure 13. The configuration parameters are selected by default, except parameters of processor MHz, a range of memory and mail result, since CPU mhz of our test machine is 3.2 GHz rather than the default one, memory range uses 1024 MB to save time that Lmbench-run costs and we need no results mailed. As can be seen from the figures, command of fork+exit in cache-dynamic-enabled group costs 344 microseconds, 11% less than that of the cache-disabled group. and other two commands also perform well. Undoubtedly, the algorithm has reduced CPU cycles.

group / 10^6 bits / second	cache_dynamic_enabled	cache_disabled
average_throughput	87.927	87.903
throughput_range	87.880~88.010	87.880~87.950

Figure 14: Netperf

As for I/O performance, we use netperf to evaluate the performance of network-intensive workloads. To overcome the adverse effect caused by real network jitter, we physically connect the testing machine directly to a tester machine by a network cable, and then the tester machine as a client measures a network throughput by sending a bulk of TCP packets to the testing machine being a server. Specifically, the client connects to the tested server by building a single TCP connection. Test type is TCP_STREAM, sending buffer size is $16KB$ and connection lasts 60 seconds. On top of that, the TCP_STREAM test of netperf is conducted for 30 times to obtain an average value of throughput. Throughput in cache-dynamic-enabled group is 87.93×10^6 bits per second, 0.02% more than that of the cache-disabled group, shown in Figure 14, and this makes no difference. Seemingly, the results indicate that the algorithm has no contribution to the performance improvement, contradicting with the results from micro experiments.

Actually, Nadav Amit [xxx] demonstrates that the virtual I/O memory map and unmap operations consume more CPU cycles than that of the corresponding DMA transaction so that the IOTLB has not been observed to be a bottleneck under regular circumstances. Thus, only when the cost of frequent mapping and unmapping of IOMMU buffers is sufficiently reduced, the guest physical address resolution mechanism becomes the main bottleneck. Furthermore, he proposes the so-called pseudo pass-through mode and utilizes a high-speed I/O device (i.e., Intels I/O Acceleration Technology) to reduce time required by DMA map and unmap operations so that IOTLB becomes the dominant factor. As a result, it is quite reasonable that netperf results with/without the algorithm are almost the same.