Improving I/O Performance of All Peripheral Devices on Paravirtualized Platforms

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 guest page tables. According to our observations, the creation, destruction and 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. Until now, researchers and developers are not aware of the existence of this dependence and do not consequently adjust the design of the paravirtualized hypervisor and the guest operating systems.

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. To minimize the impact, we propose IOSUP (I/O Speed-UP), a novel software-only approach for decreasing the IOTLB misses, as well as retaining the security of hypervisor. We also implement an prototype on Xen and Linux kernel. We do small modifications of Xen (xxx SLoC) and Linux kernel version 3.2.0 (xxx SLoC). We also evaluate the I/O performance in both micro and macro ways. The micro experiment results indicate that the new algorithm is able to effectively reduce the miss rate of IOTLB with even less CPU usage, 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.

1 Introduction

The para-virtualization technology [2, 5] is able to defend against the attacks from the software within guest Virtual Machines (VMs), but it is not armed with efficient protection approach to prevent DMA attacks [4]. To fix this gap, Intel and AMD propose the I/O virtualization (AMD-Vi [1] and Intel VT-d [3]) technology, which introduces a new Input/Output Memory Management Unit (IOMMU) to restrict DMA accesses on the specific physical memory addresses. Intuitively, the hypervisor could leverage IOMMU to protect itself by setting its occupied memory regions inaccessible for all DMA accesses. However, it is not true for the paravirtualized hypervisors (e.g., Xen). In fact, in paravirtualized environment, there are many security-critical data structures, like Global Descriptor Table (GDT) and page tables, shared by both hypervisor and guest VMs [**?**]. In such situations, only protecting the hypervisor's memory regions is far enough, as the adversary could launch DMA requests to illicitly modify those shared data structures (e.g., page tables). Once these security restrictions are broken, the malicious VM's software would be able to levearge these vulnerabilities to compromise the hypervisor.

I/O devices generate interrupts to asynchronously communicate to the CPU the completion of I/O operations. In virtualized settings, each device interrupt triggers a costly exit [2, 9, 26], causing the guest to be suspended and the host to be resumed, regardless of whether or not the device is assigned. Many previous studies that aim to improve device I/O performance While our work is different since it 1) aim to improve I/O performance for all I/O peripheral devices.

Our approach rests on the observation that the high interrupt rates experienced by a core running an I/O-intensive guest are mostly generated by devices assigned to the guest.

This revolutionary trend also urges us to significantly reshape modern operating systems to keep up the pace. Unfortunately, the current designs and implementations of modern operating systems lag behind the requirements. 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.

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 **??** and Section **??**. In Section 5, 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 has led to a lot of IOTLB misses, and this is related to the security policies that para-virtualized hypervisor enforces. Specifically, hypervisor is responsible for building isolated address spaces for itself as well as guest OSes to prevent illicit accesses from malicious guest OSes and I/O devices. So in this section, we mainly introduce how Xen hypervisor uses x86 paravirtualised (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 (see figure 1):

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.

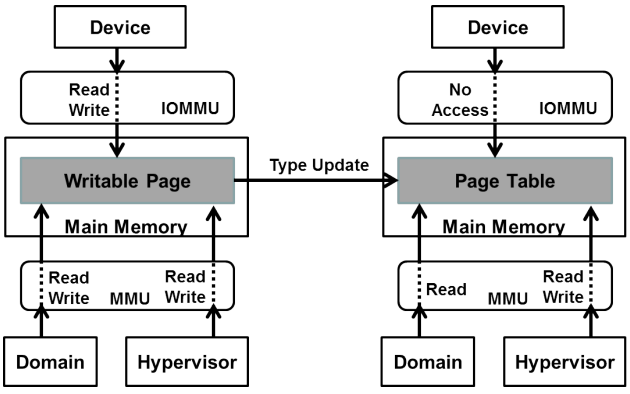


Figure 1: A translation from logical address to linear address. N.B., the translation using LDT is the same process as GDT.

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. Figure 2 illustrates the translation process when processor works in Physical Address Extension(PAE) Mode with 4K-size page.

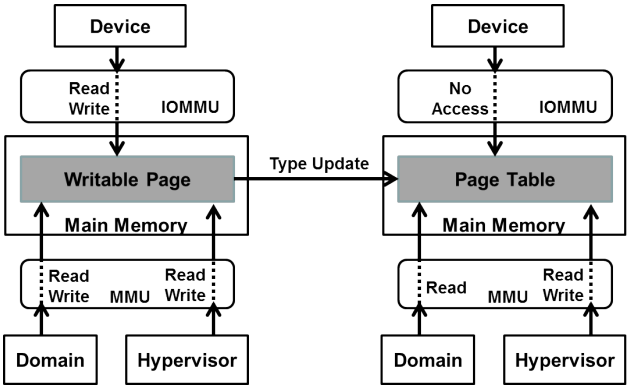


Figure 2: A translation from linear address to physical address. N.B., MMU need traverse the whole page table to find the physical address.

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.

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 ensure safety, 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. The type change of a given page will fail unless the count drops to zero. With the page type and its count, Xen is able to validate various updates. Among all the updates, we are concerned about page type updates especially caused by creating and destroying a page table, which is related to DMA-access.

2.3 DMA address translation

Intel IOMMU (i.e., Intel Virtualization Technology for Directed I/O) [xxx] mainly provides hardware support for DMA Remapping and Interrupt Remapping. DMA remapping is made use of by Xen to restrict access to par-

ticular 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 page table) for a domain, and then a DMA address (also called guest physical address in Xen) is transformed by the page table to its corresponding machine address.

For a give request identifier, two tables are used to translate it into a corresponding multi-level page table, shown in figure **??**.

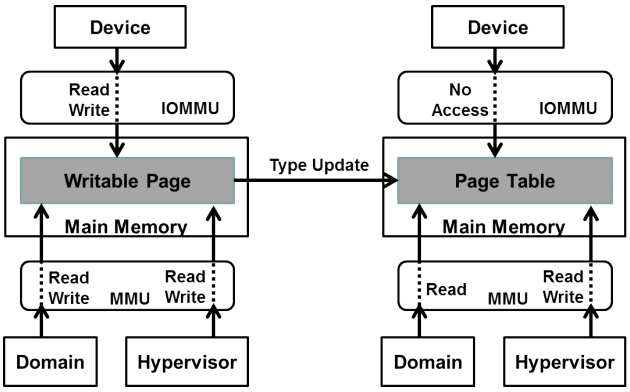


Figure 3: A mapping from the request identifier to the multi-level page table

The request identifier has three parts. The top 8-bit representing its PCI bus number specifies an entry in the root-entry table. The device number (middle 4-bit) and function number (bottom 4-bit) are used together to index into the context-entry table, which in turn points to the base address of multi-level page table. To minimize the overhead of fetching the table entries from memory, root-entry and context-entry are cached in hardware, called the context-cache.

Multi-level, page-table-base structures translates a given DMA address to a corresponding machine address. Figure **??** describes the process with a 4KB granularity of machine page, working like figure 2.

Also, frequently used page tables known as the I/O translation look-aside buffer (IOTLB) are cached in hardware. Together with the context-cache, DMA remapping hardware manages them and provides cache invalidation interfaces for software. In our PV setting, Xen is responsible for explicitly invalidating corresponding caches when modifying those tables. Behaviors of invalidating IOTLB are introduced below.

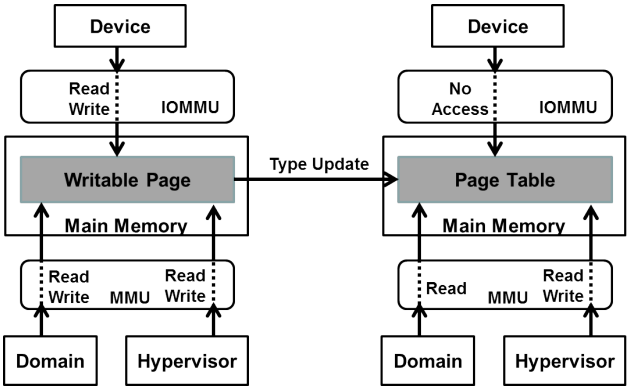


Figure 4: A mapping from DMA address to machine address

According to [xxx], IOMMU provides three types of IOTLB invalidation requests, i.e., global invalidation, domain-selective invalidation, page-selective invalidation. Intuitively, when requested entries in the IOTLB that correspond to specified DMA addresses need 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.

After a brief introduction to DMA remapping, we mainly talk about how Xen utilize it to protect guest page tables from I/O devices. During the boot-up stage of a guest OS, Xen will allocate a subset of machine pages and set every writable-page in the I/O page tables as writable. Thus, both guest OS and its assigned I/O devices have write accesses to the writable pages. To prevent illicit DMA access, Xen configures the multi-level page table to make guest page-tables as well as segment descriptor tables inaccessible to devices.

As a result, when validating the page type updates from guest OS, Xen no more allows the assigned I/O device to access those to-be-page-table pages. This is where our motivation lies in.

3 Motivation

Both guest OS and I/O devices relinquish their write-access to guest page tables implies the relationship between page type updates and I/O TLB flush. We use a new page table creation as an example to illustrate the relationship in detail.

Every time a guest OS launches a new process, it creates a new page table. Consequently, it allocates multiple pages from the buddy system and initializes them. At the time, OS fills up the pages of writable-type with entries mapping virtual addresses to the machines addresses and submits update requests, which will be examined by 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 submit to IOMMU the invalidation requests of flushing specified IOTLB entries. To achieve high performance while ensure safety, it is enough for Xen to require several page-based invalidations using queued invalidation interface. Figure 5 describes the process with a 4KB granularity of machine page.

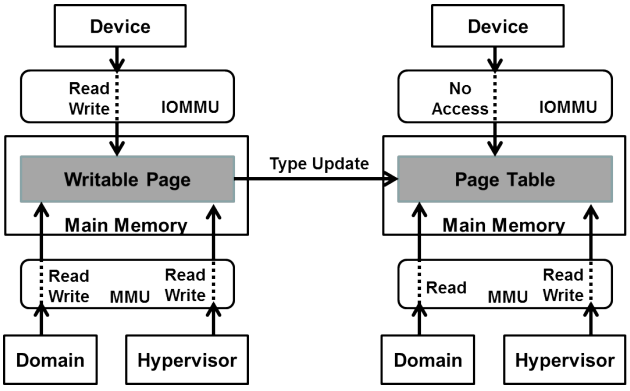


Figure 5: a page type update from writable to page-table

In a nutshell, page type updates as frequently occurs during processes creation/destruction, will force Xen to unmap/map the page from I/O page table and thus causing many IOTLB-flushes. As an IOTLB-miss requires a new page-walk for translation that can result in several consecutive accesses to memory, until the translation is done and the physical host address is resolved, a DMA transaction cannot be completed, which thereby may lead to a bad effect on I/O performance. It can be concluded that Xen has sacrificed I/O performance due to the safety reason. Because of that, we are motivated to reduce as many IOTLB misses as possible while retain the safety.

4 Problem Definition

4.1 Page Table Update in Bare-Metal Environments

5 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 while 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.

5.1 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.

5.2 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.

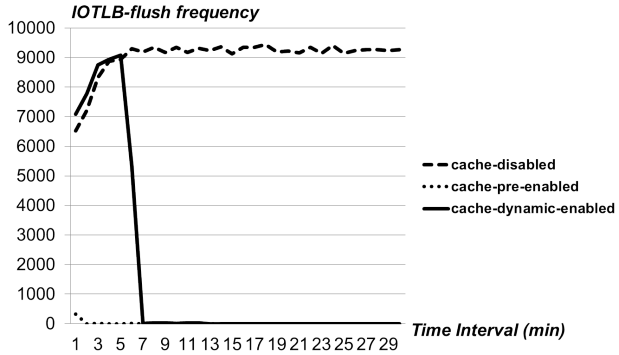


Figure 6: Frequency of IOTLB-flush

As can be seen from Figure 6. 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 7, 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.

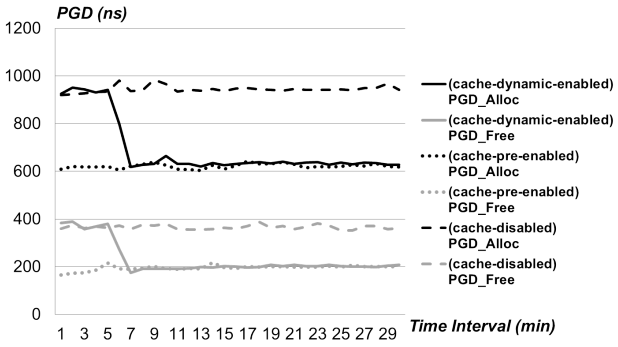
Besides CPU usage, the algorithm is evaluated in the aspect of memory size since three levels of cache pools have been built to support a fast process creation/termination. Cache pools in the cache-pre-enabled group from Figure 8 takes up 250 pages(i.e., (1000K $= 250 * 4K$) $<$ 1M) at most in the long time run, only 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: Δnum_to_free $=$ num_in_pool $-$ num_in_use.

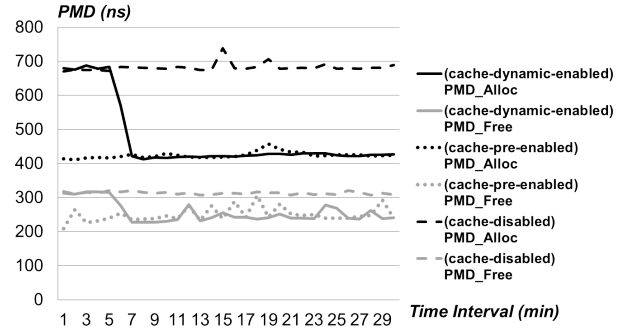
Since pre-enabling the cache is not flexible enough, we also provide another interface for users to activate the cache mechanism in an on-demand way. For instance, system has been in a busy setting for a while and then cache is enabled manually. Users may make use of this feature to better improve system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

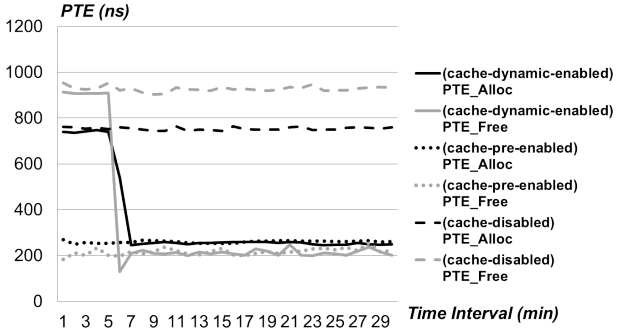
From Figure 6 and 7, both frequency of IOTLB-flush and CPU usage in this group have a very similar trend with that of the cache-disabled group in the first five min-



(a) PGD

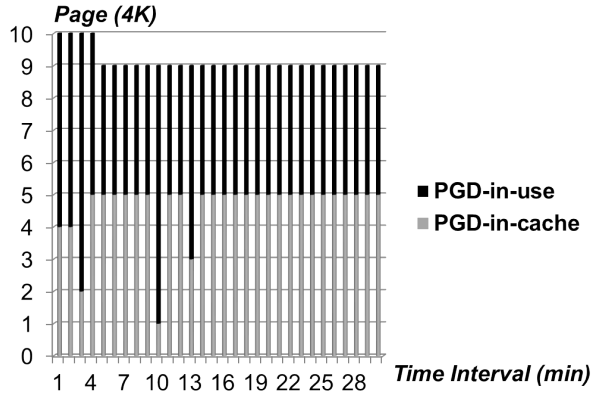


(b) PMD

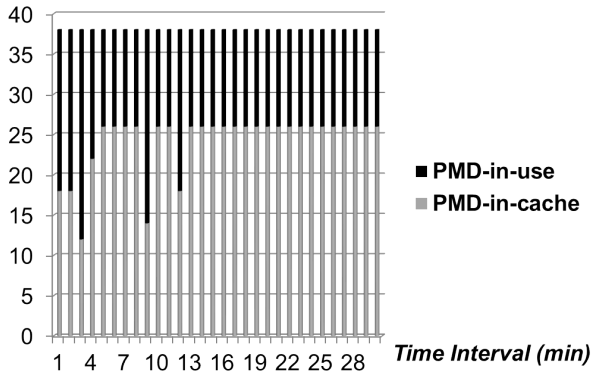


(c) PTE

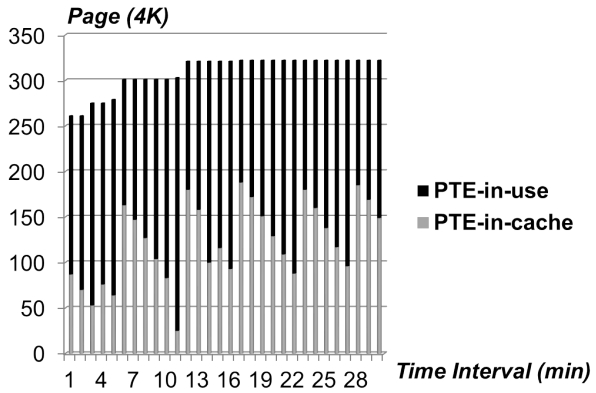
Figure 7: CPU Usage for Each Level of Page Table



(a) PGD

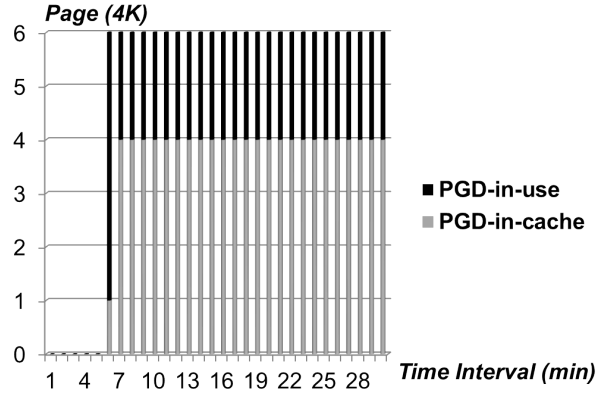


(b) PMD

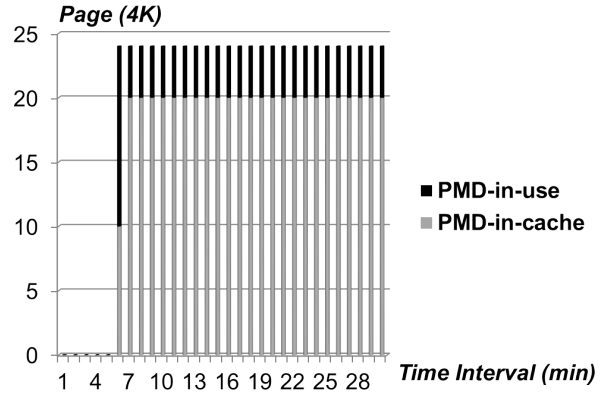


(c) PTE

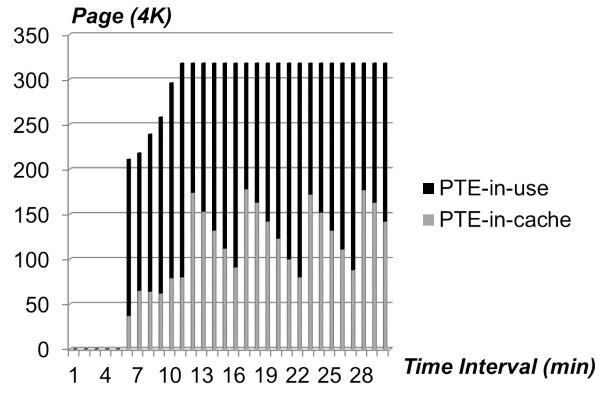
Figure 8: Cache Pools Size for Cache-Pre-Enabled Group



(a) PGD



(b) PMD



(c) PTE

Figure 9: Cache Pools Size for Cache-Dynamic-Enabled Group

utes, 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 9 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.

5.3 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.

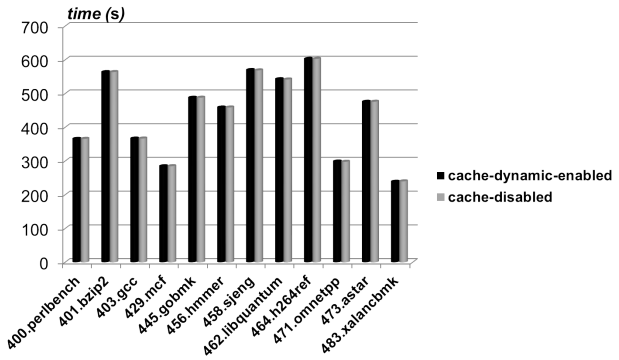


Figure 10: SPECint

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 10. 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.

Lmbench is used to measure CPU time that processes cost (i.e., fork+exit, fork+execve, fork+/bin/sh -c), shown in Figure 11. The configuration parameters are selected by default, except parameters of processor MHz, a range of memory and mail result, since CPU

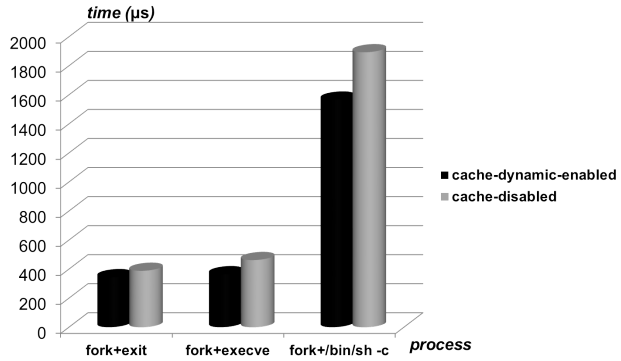


Figure 11: Lmbench

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 12: 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 12, 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.

References

[1] AMD. Secure virtual machine architecture reference manual. December 2005.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[3] Intel. Intel 64 and IA-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c. October 2011.

[4] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving xen security through disaggregation. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 151–160, New York, NY, USA, 2008. ACM.

[5] Andrew Whitaker, Marianne Shaw, Steven D Gribble, et al. Denali: Lightweight virtual machines for distributed and networked applications. Technical report, Citeseer, 2002.