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 updates of the guest 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 [**?**, **?**] 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 [**?**]. To fix this gap, Intel and AMD propose the I/O virtualization (AMD-Vi [**?**] and Intel VT-d [**?**]) technology, which introduces a new Input/Output Memory Management Unit (IOMMU) to restrict DMA accesses on the specific physical memory addresses. Thus, the hypervisor could leverage IOMMU to protect itself by setting its occupied memory regions inaccessible for all DMA accesses. It is true for the hypervisor in the full virtualization (e.g., Hardware Virtual Machines (HVMs) and Xen [**?**]), since the memory boundary between the hypervisor and guest VMs are clear and statically fixed.

If the hypervisor and the guest VMs work in full virtualization (e.g., Hardware Virtual Machines (HVMs) and Xen [**?**]), the above solution is clean and secure to defend against both software and DMA attacks from guest VMs. However, in paravirtualized environment, there are many security-critical data structures, like Global Descriptor Table (GDT) and page tables, that are inevitably used by both the hypervisor and the guest VM [**?**, **?**]. In such situations, the adversary could launch DMA requests to illicitly modify those shared data structures to open the door for the malicious guest software to bypass the security restrictions set by 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 2, 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 3, 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 Problem Definition

2.1 Page Table Update in Bare-Metal Environments

3 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.

3.1 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.

3.2 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.

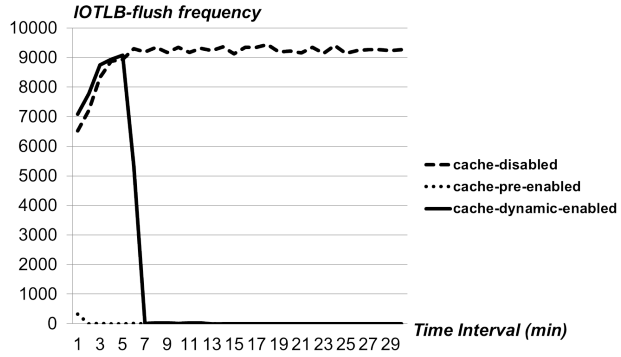
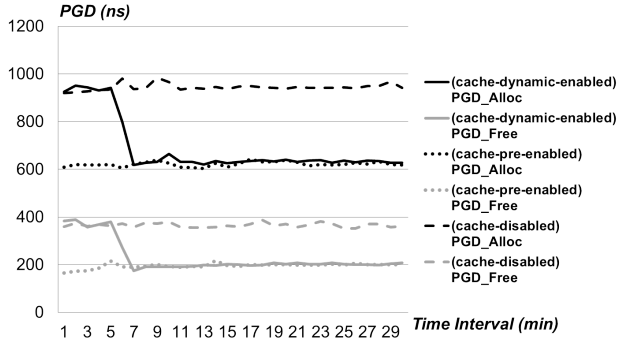


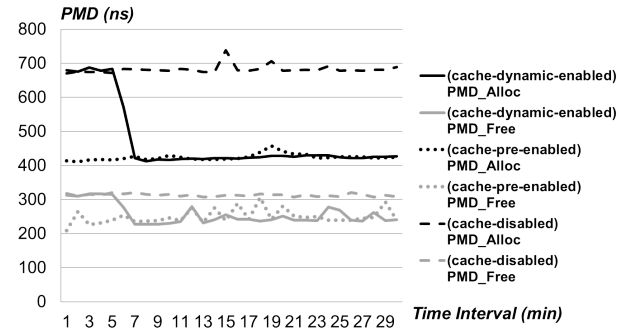
Figure 1: Frequency of IOTLB-flush

As can be seen from Figure 1. 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.

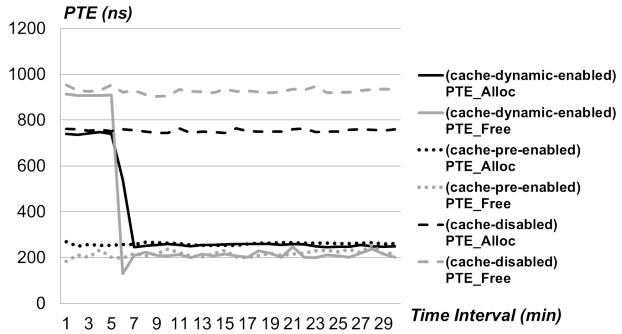
Now lets move to CPU usage that each group will take up. Specifically, each level of page table has its allocation functions and free functions, e.g., pgd_alloc() and pgd_free() and the execution time that every related function is calculated per minute. As a result, in Figure 2, allocation and free 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



(a) PGD



(b) PMD



(c) PTE

Figure 2: CPU Usage for Each Level of Page Table

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 3 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.

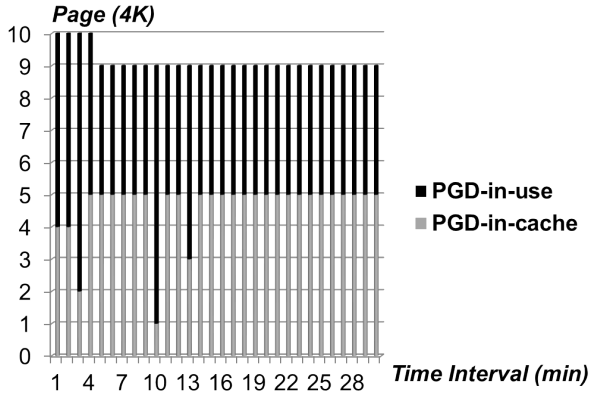
But what if the memory percentage is too high? it is necessary to free pages from the cache pool to the buddy system. Pages in pool will be freed if 1) a proportion between pages in use and in pool, and 2) a total number of pages in use and in pool are greater. And data from group of cache-pre-enabled 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.

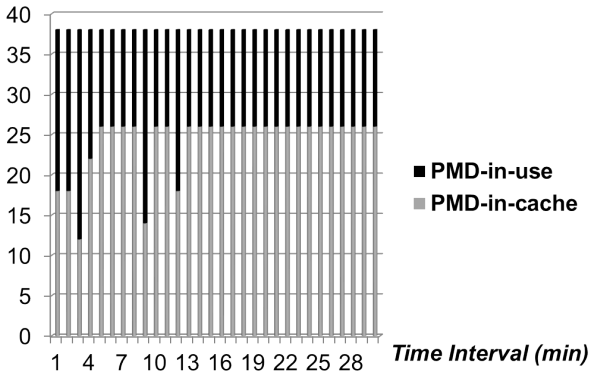
Next, in a group of cache-dynamic-enabled, cache is enabled when IOTLB-flush becomes stable while the freeing mechanism is added so as to check if this group behaves like the cache-pre-enabled group, i.e., cache-dynamic-enabled group could achieve a stable and low enough level in certain aspects, namely,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

From Figure 1 and 2, 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 4 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.

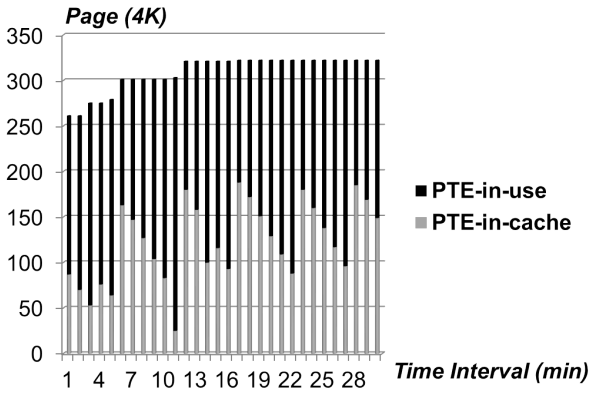
Also note that the expected results in group of cache-dynamic-enabled is obtained after sever trials by slightly modifying default values of the proportion and the total number. Since these two factors heavily relies on a specific scenario, they may not work for all. How to decide the factors is further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



(a) PGD

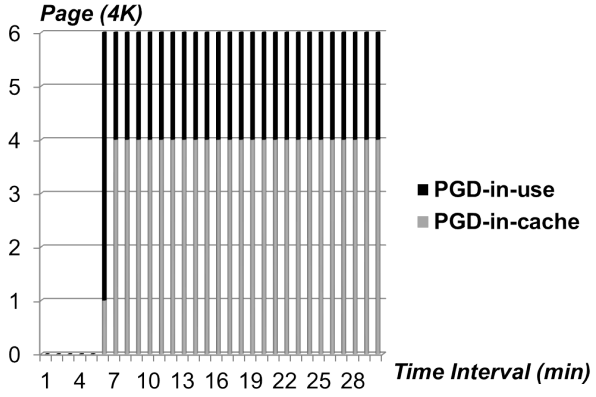


(b) PMD

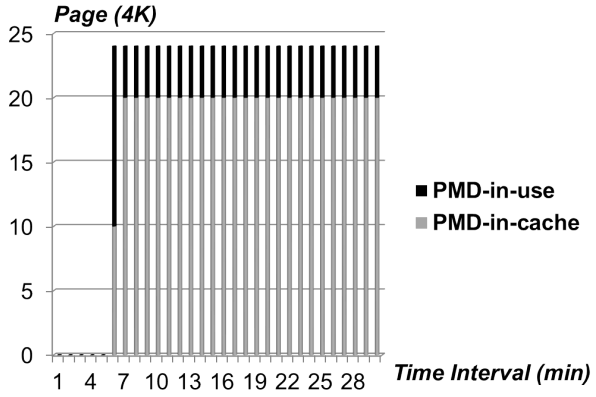


(c) PTE

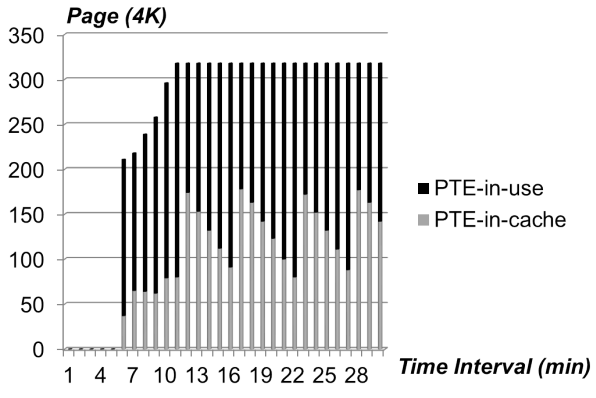
Figure 3: Cache Pools Size for Cache-Pre-Enabled Group



(a) PGD



(b) PMD



(c) PTE

Figure 4: Cache Pools Size for Cache-Dynamic-Enabled Group

3.3 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.

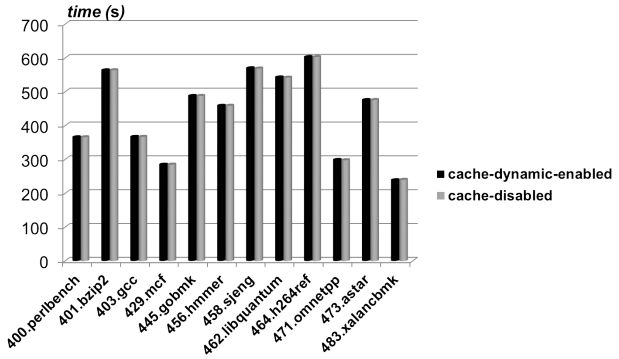


Figure 5: SPECint

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 5. 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.

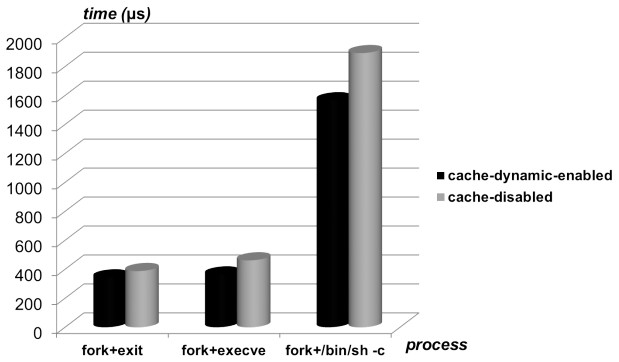


Figure 6: Lmbench

Lmbench is used to measure CPU time that processes cost (i.e., fork+exit, fork+execve, fork+/bin/sh -c), shown in Figure 6. 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 7: 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 7, 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.