The para-virtualization technology has been pervasively deployed in many cloud computing platforms (e.g., Amazon EC2). This revolutionary trend also urges us to significantly reshaped 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. ~~In specifically~~(Specifically), we aim to adopt OS kernel to further improve the I/O performance of all peripheral devices without scarifying 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 (in a granularity of page, which is for the sake of Xen’s security). However, these ~~unnecessary~~ flushes directly increase the miss rate of IOTLB, and consequently reduce I/O performance, especially for the high-speed devices. Note that we are the first one to ~~discover this dependence relationship~~ (explicitly claim that page table updates badly affect I/O performance). Based on this observation, ~~we propose a novel page table update algorithm to reduce the miss rate of IOTLB~~, (a novel algorithm is proposed to manage page table updates in guest kernel in order to reduce the miss rate of IOTLB while conform to Xen’s security policy). ~~and thereby improve overall I/O performance.~~

We implement our algorithm ~~on~~ (at the minimal customization of) modern Linux 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 improve the I/O performance by effectively reducing 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. (Also, the algorithm applies to other Linux versions above 3.2.0.)

Bad effect on IOTLB miss

To increase DMA throughput, the number of IOMMU IOTLB misses should be decreased, as each 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, thereby increasing latency

invalidation requests and invalidation interface

According to the documentation of Intel Virtualization Technology for Directed I/O[xxx], we get to know that IOMMU (an abbreviation for Memory Management Unit) 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 the specified DMA addresses need to be invalidated, a page-selective invalidation is the best choice for the sake of performance. Besides that, IOMMU also provides two types of invalidation interfaces: register based invalidation and queued invalidation interface, between which queued invalidation performs better. And that’s why Xen hypervisor in its default setting employs page-based queued invalidation interface when handling updates of the guest page tables

x Evaluation

We have implemented the proposed novel algorithm which is described in previous sections. The algorithm requires no more than xxx SLoC added to Linux while only around xxx SLoc added to Xen hypervisor. As mentioned before, the new code added to Linux kernel is to build a cache pool for guest page tables and a slight modification to Xen hypervisor is to avoid unnecessary IOTLB flushes.

This section evaluates the I/O performance as well as CPU usage of our implementation by running both macro- and micro-benchmark kits.

x.1 Experimental Setup

Our experimental platform is a HP Compaq 8100 Elite CMT PC with an Intel Core i5-3470 running at 3.20 GHz with VT-d feature enabled in the BIOS menu and we use Xen version 4.2.1 as the hypervisor while domain0 uses the Ubuntu version xxx and kernel version 3.2.0-rc1. In addition, several SLoC are added to grub.conf of the domain 0 to turn on I/O address translation and obtain log information from serial port in debug mode.

x.2 Micro-Benchmarks

To evaluate our algorithm, a possible micro-way is to measure the frequencies of IOTLB-flush with/without the cache pool. Thus, we develop a tiny stress test tool to give rise to frequent updates of page tables by launching a default browser(e.g., Firefox version), opening new tab one by one and closing the browser gracefully. As for the interval time between browser launch and closure, we set it to five minutes according to the frequency of IOTLB-flush in the control group will become relatively stable in five minutes every time the tool is launched. N.B., the control group will be talked about later.

Specifically, by making use of the tool, the measurements are performed under three particular settings. The first setting is a normal control group where system is only running the tool with cache pool disabled. Then the pool is enabled and enjoys different pressure imposed by the tool under the rest two settings. In other words, under the second setting called low-pressure group, the system activates the cache mechanism through a kernel loadable module and runs the tool concurrently. Consequently, the last one is called high-pressure group where the system activates the cache mechanism five minutes (to match the interval time) after the tool begins running.

In addition, please note that under all settings the time that system starts to run the tool is ten minutes after system boots up, since when the log info from serial port indicates that the system do not cause IOTLB-flush any more.

Now, we can take a look at how the IOTLB-flush behaves in three different groups.

Since we know that whenever the function iotlb\_flush\_qi() is invoked, several IOTLB entries will be invalidated in a page granularity. As a result, y-axis in figure 1 represents the invocation times of that function per minute, which indicates the frequency of IOTLB-flush. From this figure, the frequency in control group remains quite stable and high after five minutes. By contrast, frequency in low-pressure group drops to zero in a very short time and remains stable from then on, while frequency in high-pressure group has very similar behavior with that of control group in the first five minutes, but is reduced to zero quickly due to the cache pool.

It can be concluded that once our proposed algorithm is enabled, the IOTLB frequency can be reduced at a maximum level.

Having discussed about the I/O performance in a micro way, now let’s move to the CPU time that each group will take when the tool is put to use. Specifically, the time to allocate and free a particular level of page table (e.g., page global directory) from the cache pool or from the buddy system is calculated to evaluate the effects that the algorithm has on CPU usage. From figure 2, compared with control group which interacts with the buddy system, low-pressure group takes much less CPU time to allocate and free pages from the pool while high-pressure group also decreases to a similar level five minutes after the cache is turned on. And so do the rest two levels of page tables. As a result, the algorithm has helped to shorten the time that creating/terminating a process takes and thus increase efficiency.

Besides the I/O efficiency and CPU usage, the algorithm should also be evaluated in the aspect of memory usage since three levels of cache pools have been built to support a fast process creation/termination. We try to reach a satisfying balance between time and space. From figure 5 to 7, low-pressure group takes up to only 250 pages(i.e., 1000K < 1M) in the long time run while high-pressure group caches only 210 pages at most. It is reasonable that high-pressure group has less pages in the pool since a certain amount of page tables is freed to the buddy system right after the cache pool is put to use. And the proportion among the total pages of each level of page table including pages both in pool and in use also matches kernel paging mechanism. On top of that, when to free the pages from the pool to the buddy system is based on the training study. Specifically, using PGD pool as an example, pages in pool will not be freed unless two conditions are true. 1) proportion between pages in pool and that in use is greater than 1:1; 2) total number of pages in pool and in use exceeds 10 pages. As for the page number to free, the difference number between them is set in order to adjust the proportion to 1:1. An equation below can clearly state this point: Δnum\_to\_free = num\_in\_pool – num\_in\_use.

Note that the method to determine the proportion and total page number is based on an experiment when no pages are freed, which is the so-called training study. Further discussions about when to free will be talked about in the future work.

x.3 Macro-Benchmarks

Three different micro-benchmark tests have reached a good result for the algorithm. And then we will make use of macro-benchmarks to evaluate it also from three different aspects. Within each benchmark test, two experimental settings are designed for fair comparison. One is benchmark with the stress test tool, and the other has nothing but the benchmark. Based on each setting, performance of the algorithm is compared with that of general guest system. As a result, each benchmark will have four groups of experimental data.

SPECint\_2006v1.2 is chosen to test what effects that the algorithm will have on the whole system performance.

netperf

As for I/O performance, we use netperf to measure networking performance. Is it necessary?