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 400 SLoC added to Linux while only around 200 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 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 should configure its grub.conf to turn on I/O address translation for itself and also print log information to a 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, by making use of a tiny stress tiny tool that we develop, 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. Specifically, the tool aims to give rise to frequent updates of page tables by launching a default browser(e.g., Mozilla Firefox 31.0 in the experiment), opening new tab one by one and closing the browser gracefully. As for the interval time between browser invocation and closure, we set it to five minutes since the frequency of IOTLB-flush in the control group will become relatively stable in five minutes every time the tool is launched.

Then the cache pool is enabled and enjoys different pressure imposed by the tool under the rest two settings. In one setting called low-pressure group, the system activates the cache pool mechanism through a kernel loadable module and runs the tool concurrently.

While in the other setting called high-pressure group, the system activates the cache pool mechanism five minutes (to match the interval time) after the tool begins running. Consequently, the high pressure lies in creating/terminating a large number of Firefox processes in five minutes rather than using too much memory or CPU cycles, while the system is idle in the low-pressure group when activating the mechanism without any update of page tables occurred.

In addition, please note that under all settings the time that system starts to run the tool is ten minutes after system boots up when the log information from serial port indicates that the system do not cause IOTLB-flush any more. Specifically, We have been aware that a type change of a page table will invoke the function iotlb\_flush\_qi(), and then an corresponding IOTLB entry will be invalidated in a page granularity. Thus, a global counter is set to record invocation times of that function and an average number per minute is calculated, representing a frequency of IOTLB-flush per minute. That’s the reason why the average number logged drops to zero, meaning that IOTLB does not be flushed any more.

Now, we can take a look at figures in the three different groups. Y-axis in figure 1 represents the frequency of IOTLB-flush, corresponding to the time interval (i.e., one minute) within thirty minutes in x-axis. As mentioned before, the time to start in x-axis is when the tool is launched, causing frequent invocations of iotlb\_flush\_qi().

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 to zero, which we can regard it as a maximum reduction.

Having discussed about the I/O performance in a micro way, now let’s move to the CPU time that each group will take up 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 up 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 costs and thus increase efficiency.

Lmbench is a suite of micro-benchmarks, and we use it to measure time that process-related system commands cost (i.e., fork+exit, fork+execve, fork+/bin/sh -c), shown in figures xxx to xxx. 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 3.2GHz rather than the default one, memory range uses 1024MB to save time that Lmbench-run takes and we need no results mailed.

From the first two figures, command of fork+exit in the group of tool-off+algorithm-on only costs xxx in microseconds, which is the shortest among the rest three groups, since only lmbench is running without the tool taking up CPU cycles while the algorithm is taking effect. And time costs of the other two commands in the rest figures also support that group of tool-off+algorithm-on costs a least amount of time. As a result, we can reach a conclusion that lmbench results further illustrate the algorithm’s advantage in CPU usage ~~from a macro standpoint.~~

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, we use PGD pool as an example. Pages in pool will not be freed unless two conditions are true. 1) a 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 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 without optimization of the algorithm. As a result, each benchmark will have four groups of experimental data, i.e., (tool-on+algorithm-on, tool-on+algorithm-off), (tool-off+algorithm-on, tool-on+algorithm-on).

SPECint\_2006v1.2 is chosen to test what effects that the algorithm will have on the whole system performance. And 12 benchmarks of SPECint are all invoked with EXAMPLE-linux64-ia32-gcc43+.cfg for integer computation, results of which produce figures x.3.1 to x.3.4.

From figures x.3.1 to x.3.2, system is running SPECint as well as the test tool with/without the algorithm. Let’s use 400.perlbench as an example to explain something. The amount of time in seconds that the benchmark takes to run with the algorithm is from xxx to xxx, and the ration range is from xxx to xxx, both of which are within ranges of perlbench without the algorithm. And this explanation also works for the rest 11 benchmarks. And so do the rest two figures. Thus, we think that experimental results of each benchmark are almost the same with/without the algorithm, which indicate that the algorithm does not have any bad effect on system performance.

Netperf

As for I/O performance, we use netperf to measure networking performance.

Network cards are the

most common use-case of device assignment, due to: (1) their higher

throughput relative to other devices (which makes device assignment

particularly appealing over the slower alternatives of emulation and

paravirtualization); and because (2) SR-IOV network cards make it

easy to assign one physical network card to multiple guests.