Small Changes Big Achievements: Improving I/O Performance 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 performance of all peripheral devices will be affected by the seemingly unrelated guest page table updates. Unfortunately, no existing work uncovers this dependency 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 accelerates the creations and exits of user processes. We implement a 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 the paravirtualization [**?**, **?**], the kernel of each guest Virtual Machine (VM) and the hypervisor share the same virtual space including many security-critical data structures, like page tables and Global Descriptor Table (GDT). All these data structures are managed by the guest VM, but their updates are intercepted and validated by the hypervisor [**?**] with the purpose 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 peripheral devices.

Problem. Although the combination of the paravirtualization and I/O virtualization brings strong security protection, but we surprisingly find out that it slows down the speed of DMA address translation due to the additional I/O translation look-aside buffer (IOTLB) flushes. Specifically, the guest page tables should be protected from any DMA access. Thus, when a writable page[1] turns into a page-table page, the hypervisor has to update the IOMMU page table and flush an IOTLB entry to prevent DMA from accessing it. The IOTLB flush is necessary for the sake of the security of the hypervisor, but it inevitably increases the miss rate of IOTLB, and consequently reduces I/O performance, especially for the high-speed devices. Similarly, when a page-table page is turned to a writable page, the hyperisor needs to return the ownership to the guest VM. As a result, the hypervisor has to update the IOMMU page table and flush IOTLBs for allowing the guest VM to freely access the returned writable page. In addition, these page-type changes between page-table page and writable page are *often* triggered during the whole life cycle of a run-

[1] A *writable page* is the page in the guest VM, allowing the guest VM to freely read and write according to its own purpose.

ning system. As a consequence, the IOTLB flushing events are *frequently* triggered, which inevitably lower the speed of the DMA address translation and the I/O performance of all peripheral devices.

This new dependency issue between guest page tables 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. In response to this problem, we aim to improve the guest kernel and the hypervisor to reduce the flush rate of IOTLB, and therefore ameliorate its impacts on the I/O performance of all peripheral devices.

Our approach rests on the observation that the high IOTLB flush rates experienced by a guest VM are due to the page-type changes generated by the numerous creations and destructions of page tables. We cannot reduce the number of the creation/destruction of page table, but reducing the number of page-type change is possible. Inspired by this observation, we propose the page-table cache, which queues the released/freed page-table page in the hope it will be reused (poped out the cache) in the creation of page table in the near future. During this process, the destruction and the creation of the page table will not introduce page type change, and therefore there will be no IOTLB flush. Besides minimizing the IOTLB flush rate, the page-table cache also accelerates the creation and destruction of page table. The reason is that the kernel does not need to ask for pages from enormous and costly memory management subsystem, instead it could quickly get pages from the page-table cache with very small cost. In order to allow the kernel memory management subsystem to use the pages cached in the page-table cache, we add an interface for the page-table cache to free pages.

We implement the page-table cache with small modifications of hypervisor Xen (xxx SLoC) and Linux kernel version 3.2.0 (xxx SLoC), and evaluate the I/O performance in micro and macro ways. The micro experiment results indicate that the new page-table cache is able to effectively reduce the miss 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.

In particular, we make the following contributions:

1. We are the first, to the best of our knowledge, to identify the dependency issue between guest page table and DMA transferring.

2. We proposed a novel approach - page table cache, to improve I/O performance of all peripheral devices by minimizing the flush rate of IOTLB, without sacrificing the system security.

3. We implemented a prototype of the page table cache and evaluated the performance in both micro and macro ways.

The rest of the paper is structured as follows: In Section **??** and Section 3, we briefly describe the background knowledge, and highlight our goal and the thread model. In Section 4.1 we discuss the design rationale. Then we describe the system overview and implementation in Section 4 and Section **??**. In Section **??**, 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

The dependency between the guest page table and the IOTLB is very subtle. To fully understand the dependency, we need to know the details about how the paravirtualized hypervisor protects itself through guest page table and IOMMU. Specifically, we explicitly describe 1) paravirtualized MMU mode, and 2) IOMMU and DMA address translation. As Xen [**?**] is a typical and popular paravirtualized hypervisor, in this section we use Xen in a x86 MMU model [**?**] to illustrate the details. We believe you can easily find the corresponding mechanisms on other paravirtualized hypervisors. If you are already familar with these, you can skip this section to the motivation section (Section **??**).

2.1 Paravirtualized MMU Model

In this subsection, we firstly introduce the address translation mechanism on x86, and then we highlight how Xen protects itself in the paravirtualized MMU model.

2.1.1 Address Translation

There are two steps to transform logical addresses (i.e., addresses as viewed by programmers) into physical addresses (i.e., actual addresses in physical memory): 1) segment translation, in which a logical address (consisting of a segment selector and segment offset) is converted to a linear address, and 2) page translation, in which a linear address is converted to a physical address[2]. Figure 1 illustrates the whole translation process.

Segementation Descriptor Table Each logical address consists of a *segment* selector and an *offset*, which are used to locate a segment descriptor. A segment descriptor describes a segment, including the segment

[2]Physical address is also called machine address in Xen setting.

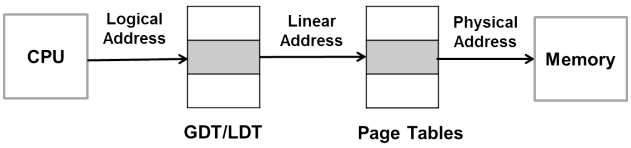


Figure 1: Address Translation

base address, limiation of the segment lenght and other meta-data. All segment descriptors are stored in the Global Descriptor Table (GDT) or Local Descriptor Tables (LDT). In order to translate to the exptected linear address, the integrity of the GDT and LDT should be guaranteed.

Page Table Page tables are used by a hardware, i.e., Memory Management Unit (MMU), to translate the linear addresses into physical addresses used by the hardware to execute instructions. In the PAE-enabled paging mode, a page table has three levels: L1 level (bottom level), L2 level (middle level) and L3 level (top level). The slots in L1, L2 and L3 levels are known as Page Table Entry (PTE), Page Middle Directory (PMD) and Page Global Directory (PGD), respectively. A PTE slot could determine the access permissions of a page, e.g., the kernel could set a page as read-only by clearing the bit within a PTE slot that represents the writable permission.

There are many page tables in a guest VM, as each user process has its own page tables. The creation and exit of a user process will be accompanied by the creation and destruction of a page table respectively. It means that if there are numerous temporary processes generated within a period, there will be a large number of page tables created.

2.1.2 Page Types and Security Validations

Page Type	Descriptions
None	No special uses
L*N* Page-Table Page	Pages used as a page table at level *N*. There are separate types for each of the 3 levels on 32bit PAE guests.
Segment Descriptor Page	Pages used as part of the GDT/LDT
Writable Page	Pages are writable for software and DMA.

Table 1: The page types used in Xen.

In order to ensure that the guest cannot subvert the system, Xen requires that certain update policies are maintained, and thus all updates of the page tables should be vetted by Xen. To this end, the guest OS is deprivileged, from ring-0 to ring-1, leaving ring-0 for the Xen hypervisor. This prevents the guest OS from executing privileged instructions, e.g., the guest OS cannot directly update control registers.

Xen also defines a number of page types, which are listed in Table 1, and maintain a type reference count for each page. Xen enforces the policies that any given page has exactly one type at any given time, and only pages with the writable type have a writable mapping in the page tables. By doing this it can ensure that the guest OS is not able to directly modify any page-table pages and therefore subvert the security of the whole system. If the guest kernel attempts to update the page table, it has to issue a hypercall to ask the hypervisor to complete the update. As Xen is always involved in all updates of the page tables, the policies on the page table updates are non-bypassable.

Whenever a page table is loaded into the hardware page-table base register (cr3), the hypervisor takes an appropriate type reference with the L3 page-table type. If the page is not already of the required type, then in order to take the initial reference it must first have a type count of zero. In addition, it must be validated to ensure that it follows the following policy: for a page with a page-table type to be valid, it is required that any pages referenced by a present page table entry in the page have the type of the next level down. For instance, any page referenced by a page with type L3 Page Table must itself have the type L2 Page Table. This policy is applied recursively down to the L1 page table layer. At L1 the invariant is that any data page mapped by a writable page table entry must have the writable page type. By applying these policies, Xen ensures that all page-table pages as a whole are safe to be loaded into the cr3. Similar requirements are also placed on other special page types, e.g., GTD/LDT pages.

The page type is allowed to be changed. Xen enforces that the type of a page can only be changed when the type count is zero. In addition, Xen also requires that every page type update only occur between writable and non-writable pages, as summarized in Figure 2.

Yueqiang says: This figure2 should be revised. 1) font size 2) figure size and 3) rectangle is not needed to be shaded. ‘

2.2 IOMMU and DMA address translation

The input/output memory management unit (IOMMU) [**?**] is a memory management unit (MMU) that connects a DMA-capable I/O bus to the main mem-

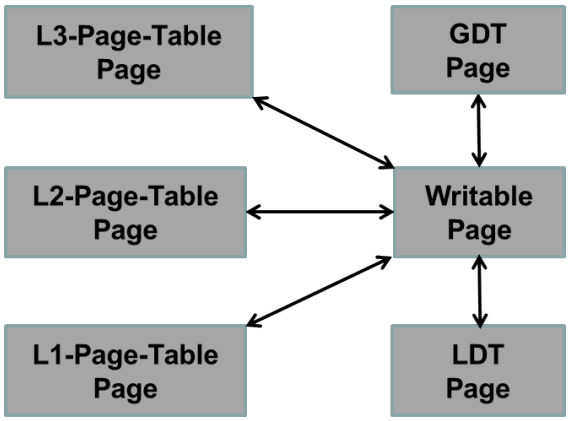
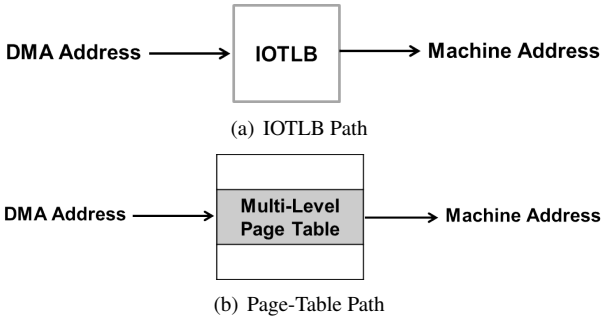


Figure 2: Page Type Updates



(a) IOTLB Path

(b) Page-Table Path

Figure 3: DMA Address Translation

ory. Like a traditional MMU, the IOMMU maps device addresses (also called as I/O addresses) to physical addresses through a dedicated page table. This technique is also known as DMA remapping. The IOMMU page table that is created and maintained by the hypervisor in its own space, is able to restrict the access on a particular page by configuring the permission bits. The hypervisor grants different access permissions for different page types, such as the writable pages are always allowed full access permissions, while the page-table pages are always inaccessible to any devices. This is why when the page-type changes between writable page and page-table page, updating IOMMU page table is always necessary.

However, the DMA remapping always needs the page table walking, which is slow and inefficient. To accelerate the translation speed, the I/O translation look-aside buffer (IOTLB) is introduced. The IOTLB is used to cache frequently accessed page table entries. By doing so, the IOTLB is very likely to be accessed, indicating that the physical address of a queried DMA address will be immediately fetched through the IOTLB path (Figure 7(a)). If unlikely the IOTLB miss occurs, the DMA remapping still can go the slow page-table path to get the physical address (Figure 7(b)). To achieve a better I/O performance, the DMA remapping should avoid taking the page-table path as far as possible.

According to IOMMU specification [**?**], a typical IOMMU is able to provide three types of IOTLB invalidation schemes, i.e., global invalidation, domain-selective invalidation, page-selective invalidation, which differ in granularity. Specifically, the global invalidation will always invalidate all IOTLB entries as a whole. The domain-selective invalidation only invalidates the selected VM domain's IOTLBs, whose performance is a little better than the previous global invalidation. The page-selective invalidation that only invalidate the corresponding IOTLB could achieve the best performance, comparing with the previous two schemes. Besides the invalidation scheme, IOMMU also supports two kinds of invalidation interfaces: register based invalidation and queued invalidation interface, between which queued invalidation performs better. No matter the IOTLB flushes frequently in whatever granularity, it will inevitably increases the probability of IOTLB misses, thereby introduce negative effects on the I/O performance.

2.3 Problem Insights

Based on the observations and deep analysis, we know that there are seven page types, and the updates among them always trigger additional IOTLB flushes. In particular, segment descriptor pages are rarely updated, as they are treated as almost const structures. However, the page-table pages are frequently updated from/to writable pages. These updates that are driven by the process creations and exits are frequently triggered in the whole life cycle of a running system. Thus, they are becoming the main source for contributing the additional IOTLB flushing.

The additional IOTLB flushes are likely to let the DMA address translations take the slow and inefficient page-table path, instead of taking the fast and efficient IOTLB path (Figure 3), which inevitably lowers the speed of the whole DMA transfering, especially for the high performance devices.

In brief, we summarize all these into three key points, which are listed as follows:

1. (O1) Each page-type change triggers the invalidation of at least one IOTLB entry.

2. (O2) The main source of causing IOTLB flush is the page-type changes between writable pages and page-table pages.

3. (O3) The additional IOTLB flushes inevitably lower the I/O performance of the peripheral devices.

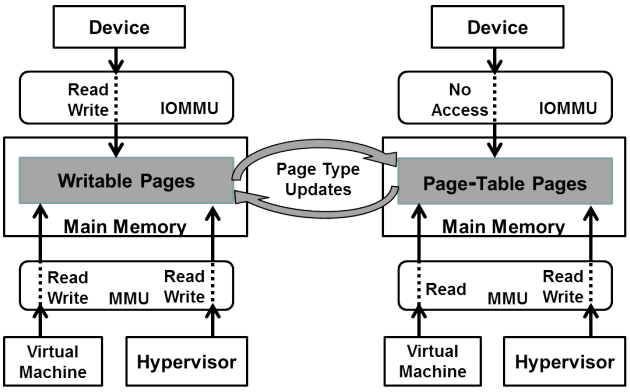


Figure 4: The page type updates between writable and page-table pages.

3 Problem Definition

3.1 Page Table Update in Bare-Metal Environments

4 IOSUP Overview

4.1 Design Rationale

It is challenging to reduce the number of the IOTLB flushes without sacrificing the system security.

In order to reduce the number of the IOTLB flushes, we need either

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.

4.2 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.

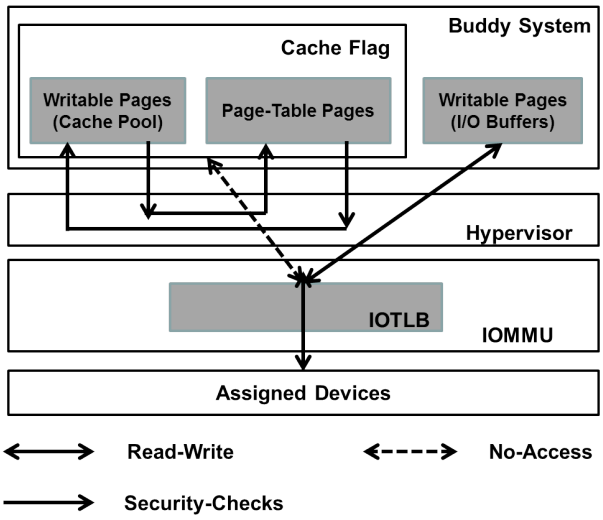


Figure 5: Overview

4.2.1 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.

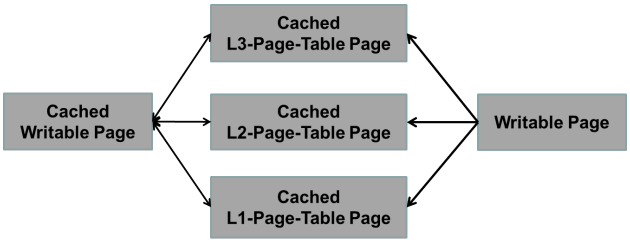


Figure 6: Page Type Updates with Cache Flag

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.

4.3 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 **??**), and the page number to free is stated in the equation below: Δnum_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 5 will demonstrate its small modifications in details.

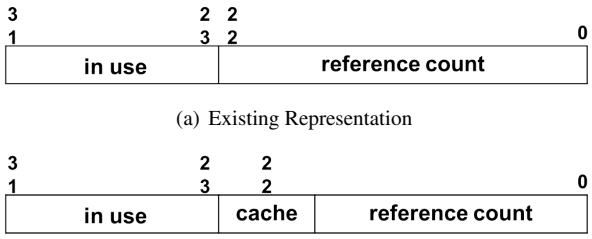
5 Implementation

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

5.1 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 7(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 7(b)). Actually, Xen



(a) Existing Representation

(b) Cache Flag Representation

Figure 7: Field of Page Type Info

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.

5.2 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.