A Simplified TDP with Large Tables

Yu Zhang(✉)

Rechnerarchitektur, Fakultät Für Informatik,
Technische Universität Chemnitz, 09126 Chemnitz, Germany
zhayu@hrz.tu-chemnitz.de

Abstract. Among the performance bottlenecks for the virtual machine, memory comes next to the I/O as the second major source of overhead to be addressed. While the SPT and TDP have proved to be quite effective and mature solutions in memory virtualization, it is not yet guaranteed that they perform equally well for arbitrary kind of workloads, especially considering that the performance of HPC workloads is more sensitive to the virtual than to the native execution environment. We propose that based on the current TDP design, modification could be made to reduce the 2D page table walk with the help of large page table. By doing this, not only the guest and host context switching due to guest page fault could be avoided, but also the second dimension of paging could be potentially simplified, which will lead to better performance.

1 Introduction

In the context of system virtualization, SPT (shadow page table) and TDP (two-dimensional paging)[1] are the two mature solutions for memory virtualization in the current hypervisors. Both of them perform address translation transparently from the guest to the host. In dealing with the translation chain from GVA (guest virtual address) to HPA (host physical address), the SPT combines the three intermediate steps for each GVA→HPA into a single entry, which contains the wanted address and saves further efforts to walk through both of the guest and host page tables as long as the cached entry is not invalidated in any form. However, as SPT is a part of the hypervisor and must be kept as consistent as possible with the guest page table, the processor had to exit from the guest to host mode to update the SPT and make it accessible, during which a considerable number of CPU cycles could have been wasted. TDP comes as a remedy by keeping GVA→GPA translation in the guest, while shifting GPA→HPA translation from the hypervisor to the processor. The expensive `vmexit` and `vmentry` due to guest page fault are avoided by TDP. Unfortunately in case of TLB-miss (translation look-aside block) the multi-level page table must still be walked through to fetch the missing data from the memory. Because of this nature, the TLB is not quite helpful in preventing page table from being walked when running workloads with poor temporal locality or cache access behavior. As a result,

[1] TDP it is known as the AMD NPT and Intel EPT. For technical neutrality reason TDP is used to refer to the paging mechanism with hardware assistance.

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 789–801, 2015.
DOI: 10.1007/978-3-319-27308-2_63

the performance gain could be more or less offset by the overhead. We attempt to combine the merits of the two methods and meanwhile avoid the downsides of them by adapting the `mmu` code in the hypervisor.

For TLB contains a number of the most recently accessed GPA→HPA mappings, the more likely these entries will be needed in future, the more time could be saved from the page table walk in subsequent operations. To the nature of the paging methods themselves, the efficiency of both SPT and TDP rely on to what extent the cached results of the previous page table walks could be reused. Since the SPT cannot be maintained without interrupting the guest execution and exit to the host kernel mode, there will be little chance other than reducing the occurrence of the page fault in the guest to improve the performance of SPT. This, however, largely depends on the memory access behavior of the individual workload and remains beyond the control of the hypervisor. TDP, on the hand, bears the hope for performance improvement. Currently the TDP is adopting the same paging mode as the host does, known as "multi-level page table walk-through". It is an N-ary tree structure [1], where N could be 1024 in 32-bit mode, or 512 in 64-bit or 32-bit PAE modes. In the 32-bit mode only 2 level page table are involved for walking through, which poses minor overhead. However, the overhead grows quickly non-negligible as the paging level increases. In spite of the various paging modes adopted in the guest, only two modes - the 64-bit and the 32-bit PAE modes are available for the TDP. In the worst case if all TLB large missed, up to 24 memory accesses are possible for a single address translation.

Though undesirable, this has presumably been done for two reasons: 1. compatibility between the host and guest paging modes, and more importantly, 2. efficiency in memory utilization. However, as the TDP table is in the hypervisor and invisible to the guest, it is actually up to the hypervisor to adopt the paging mode without having to maintain this kind of compatibility [2]. For the tree structure forms a hierarchy of the 1-to-many mappings, mappings could be built in a "lazy" way only on demand, significant amount of memory space for entries could be saved compared with the 1-to-1 mapping based structure, say, an array. On the other hand, this structure also means more memory access and time cost while looking up an element within it. As performance rather than memory saving comes as the top concern, paging methods more efficient than the current one may exist. One candidate is naturally a 1-to-1 mapping based structure, such as an array, or a hash list. With more memory being invested to save all the possible GPPFN (guest physical page frame number) to HPPFN (host physical page frame number) mappings, fewer memory accesses suffice[2] in the second dimension of the TDP. By doing this, the whole translation from GVA to HPA is expected to be effectively simplified and accelerated due to a reduced paging structure in TDP table. In addition, a simplified TDP method combines the merits of both TDP and SPT - to avoid the `vmexit` as well as to maintain the relatively short mapping chains from GVA to HPA. Until signif-

[2] This is the case only if a single large table is used. In our design, however, due to the limitation of memory chunk size in the kernel, multiple tables could be used.

icant change is made available to the paging mode of the current processor, it could be a better practice merely by modifying the current hypervisor software.

2 Related Work

Major work focusing on improving the memory virtualization could be summarized as the following. To work around the unfavorable sides and combine the best qualities of SPT and TDP, one attempt is to enable the hypervisor to reconfigure its paging method at run-time as a response to the ever changing behavior of the workloads in memory accessing, which were implemented in the past in a few hypervisors such as Xen and Palacios according to [3,4]. Although not all workloads could be benefited from this, overall performance gain have been observed for the selected benchmarks. The downside, however, is that it adds further complexity to the hypervisor with the methods of performance metric sampling, paging method decision making, as well as the dynamic switching logic. Furthermore, such activities in the kernel could also do harm to the performance.

To reduce the overhead for walking through the multi-level page tables in TDP, a hashed list is applied to provide direct address mapping for GPA [2]. In contrast with the $O(n^2)$ complexity of the conventional multi-level forward page tables for both GVA→GPA and GPA→HPA translations, the hashed page table has only one paging level and achieves a complexity of $O(n)$ in theory. The performance is at least not worse due to the reduced page table walk and cache pressure, showed by the benchmark. Since the hash table is a data structure more capable in searching, inserting and deleting etc., and relatively easier to be implemented within the existing framework of the hardware and software, current TDP design could be simplified by applying it for better performance. As more reflections were cast on the current multi-level paging modes, a variety of changes have been prompted for a simplified paging work. Theoretically, a "flat nested page table" could be formed by combining the intermediate page levels from 4 to 1, which results in an 8 memory access for the translation from GVA to HPA, and a reduced overhead for 2D TDP walk [5]. By extending the processor and hypervisor with the "direct segment" function, the memory access for the GVA to HPA translation could even be further reduced to 4 or 0 [6].

For the TLB plays a critical role in reducing the address translation overhead [7] and justifies the use of TDP, it becomes another concern besides the paging level. Specific to the AMD processor, a way is suggested in [8] to accelerate the TDP walk for guest by extending the existing page walk cache also to include the nested dimension of the 2D page walk, caching the nested page table translations, as well as skipping multiple page entry references. This technique has already gained its application in some AMD processors. Not limited to virtual cases, attention is paid in [9] to compare the effectiveness of five MMU cache organizations, which shows that two of the newly introduced structures - the variants of the translation outperform the existing structures in many situations.

3 Structures and Operation of the TDP

As a potential technical breakthrough, TDP is different from SPT in many aspects. However, for compatibility reason, the main structure of SPT is still reused by TDP. This, though at first may seem quite misleading, enables the TDP to fit seamlessly into the current framework previously created for SPT. As far as TDP feature is available in the hardware, it is preferred to SPT for general better performance. While in the absence of TDP hardware feature, SPT may serve as a fall-back way and the only choice for the hypervisor to perform the guest-to-host address translation.

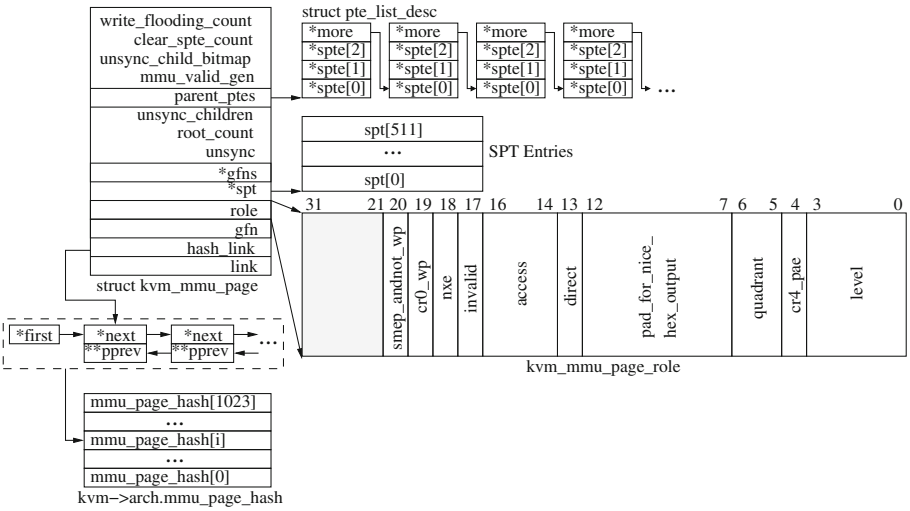

Fig. 1. Data Structure for SPT reused by TDP in KVM

In KVM, SPT and TDP share the same data structure of the virtual MMU and page tables (surprisingly, both are named as shadow page table). The shadow page table is organized as shown by Fig. 1, of which `kvm_mmu_page` is the basic unit gluing all information about the shadow pages together. For each level of the shadow page table, a pageful of 64-bit `sptes` containing the translations for this page are pointed to by `*spt`, whose role regarding the paging mode, dirty and access bits, level etc. are defined by the corresponding bits in `role`. The page pointed to by `spt` will have its `page->private` pointing back at the shadow page structure. The `sptes` in `spt` point either at guest pages, or at lower-level shadow pages [10]. As the `sptes` contained in a shadow page may be either one level of the PML4, PDP, PD and PT, the `pte_parents` provides the reverse mapping for the pte/ptes pointing at the current page's `spt`. The bit 0 of `parent_ptes` is used to differentiate this number from one to many. If bit 0 is zero, only one `spte` points at this pages and `parent_ptes` points at this single `spte`, otherwise,

```
static void set_tdp_cr3(struct kvm_vcpu *vcpu, unsigned long root)
{
        struct vcpu_svm *svm = to_svm(vcpu);
        svm->vmcb->control.nested_cr3 = root;
        mark_dirty(svm->vmcb, VMCB_TDP);
          ...
        svm_flush_tlb(vcpu);
}

static void init_kvm_tdp_mmu(struct kvm_vcpu *vcpu)
{
        struct kvm_mmu *context = vcpu->arch.walk_mmu;
          ...
        context->page_fault = tdp_page_fault;
          ...
        context->root_hpa = INVALID_PAGE;
        context->direct_map = true;
        context->set_cr3 = kvm_x86_ops->set_tdp_cr3;
          ...
        context->inject_page_fault = kvm_inject_page_fault;
          ...
}

static void init_vmcb(struct vcpu_svm *svm)
{
        struct vmcb_control_area *control = &svm->vmcb->control;
        struct vmcb_save_area *save = &svm->vmcb->save;
          ...
        if (npt_enabled) {
                /* Setup VMCB for Nested Paging */
                control->nested_ctl = 1;
                clr_intercept(svm, INTERCEPT_INVLPG);
                clr_exception_intercept(svm, PF_VECTOR);
                clr_cr_intercept(svm, INTERCEPT_CR3_READ);
                clr_cr_intercept(svm, INTERCEPT_CR3_WRITE);
                save->g_pat = 0x0007040600070406ULL;
                save->cr3 = 0;
                save->cr4 = 0;
        }
          ...
}
```

Fig. 2. VMCB configuration for TDP

multiple `sptes` are pointing at this page and the `parent_ptes` & `0x1` points at a data structure with a list of `parent_ptes`. `spt` array forms a directed acyclic graph structure, with the shadow page as a node, and guest pages as leaves [10].

KVM MMU also maintains the minimal pieces of information to mark the current state and keep the `sptes` up to date. `unsync` indicates if the translations in the current page are still consistent with the guest's translation. Inconsistence arises when the translation has been modified before the TLB is flushed, which has been read by the guest. `unsync_children` counts the sptes in the page pointing at pages that are unsync or have unsynchronized children. `unsync_child_bitmap` is a bitmap indicating which `sptes` in `spt` point (directly or indirectly) at pages that may be unsynchronized. For more detailed description, the related Linux kernel documentation [10] is available for reference.

Multiple `kvm_mmu_page` instances are linked by an `hlist_node` structure headed by `hlist_head`, which form the elements in the hash list - `mmu_page_hash` pointed to by `kvm->arch`. Meanwhile it's also linked to either the lists `active_mmu_pages` or `zapped_obsolete_pages` in the `kvm->arch`, depending on the current state of the entries contained by this page. Both SPT and TDP keep their "shadow page table" entries and other related information in the same structure. The major difference lies in the hypervisor configuration of the runtime behaviors upon paging-fault-related events in the guest. While the SPT relies on the mechanism of "guest page write-protecting" and "host kernel mode trapping" upon guest page fault for keeping the SPT synchronized with the guest page table, the TDP achieves the same result by a hardware mechanism. As VMCB (virtual machine control block) by AMD or VMCS (virtual machine control structure) by Intel is the basic hardware facility the TDP makes use of, it's the key thing making difference. Code snippet in Fig. 2 shows the configuration of VMCB for TDP, and that the root address of the TDP page table is kept in the VMCB structure. Meanwhile the guest is configured as exitless for paging-fault exception, which means that the page fault events is handled by the processor. With this configuration, guest is left running undisturbed when the guest page fault occurs.

Besides, as SPT maps GVA to HPA, the `spt` entries are created and maintained in a per-process way, which leads to poor reusability hence higher memory consumption. These are obvious downsides especially when multiple processes are running in parallel. In contrast, the TDP maintains only the mappings from GPA to HPA, which effectively eliminated such problems associated with SPT. Guest page table is also accessed by the physical processor and forms the first dimension of the entire page table walk. In this way the TDP can not only eliminate the cost for frequent switching between the host and guest modes due to SPT synchronization, but also simplify the mappings and maintenance efforts the "shadow page tables" needs.

Two stages are involved in the buildup of the TDP table, namely, the creation of the virtual mmu, and the filling of TDP page tables upon guest page fault during the execution of the guest. As Fig. 3 depicts, in the context of the function `kvm_vm_ioctl`, the virtual mmu is created for the first time along with the guest VCPU. It is also when the VMCB is configured. One thing to be noticed is that, as the root address of the TDP page table, the `root_hpa` of the `kvm_mmu` is left without to be allocated a page table, which is deferred to the second stage.

Figure 4 depicts the context function `vcpu_enter_guest`, in which operations related to the second stage take place. This function serves as an interface for the inner loop[3] in the internal architecture of the QEMU-KVM, dealing with host-guest mode switching. Before the guest mode is entered by the processor, much preparation work needs to be done in this context, including the checking and handling of many events, exceptions, requests as well as mmu reloading or I/O emulation. The only thing needed for mmu reloading is to allocate a page

[3] The outer loop is formed by `ioctl` commands issued by QEMU from the user-space, dealing with user-space and host kernel-space switching.

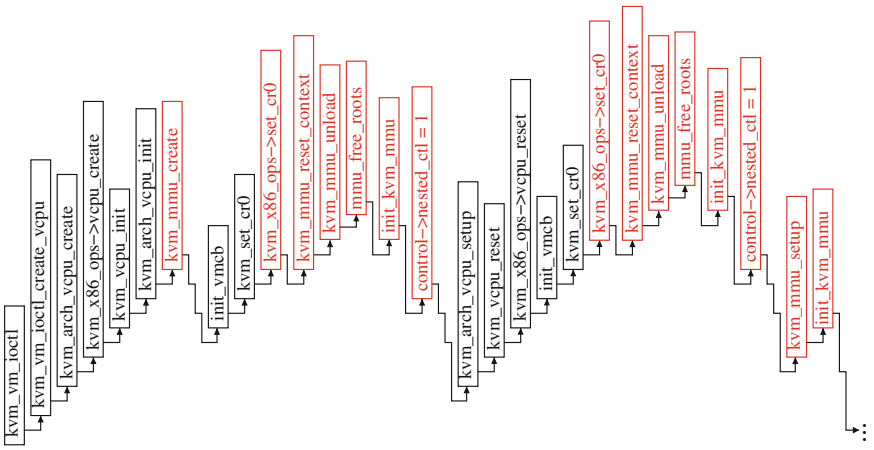

Fig. 3. Framework for virtual MMU creation in KVM

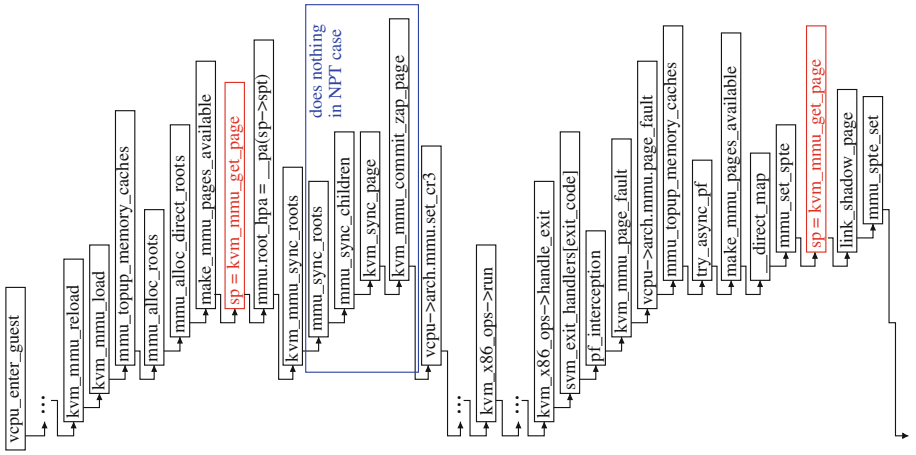

Fig. 4. Framework for page fault handling in KVM

for the TDP table and make the starting address of it known to the `root_hpa` of the `kvm_mmu` and the `CR3` of the VCPU, which is performed by `kvm_mmu_load`.

Guest begins to execute until it can't proceed any further due to some faulty conditions. More often than not, control flow had to be returned to the hypervisor or the host OS kernel to handle the events the guest encountered. Obviously too much vmexit are an interference and grave source of overhead for the guest. With TDP, however, guest is free from `vmexit` upon guest paging faults. As the guest enters for the first time into execution, the paging mode is enabled and the guest page tables are initialized, however, the TDP tables are still empty. Any fresh access to a page by the guest will first trigger a guest page fault. After

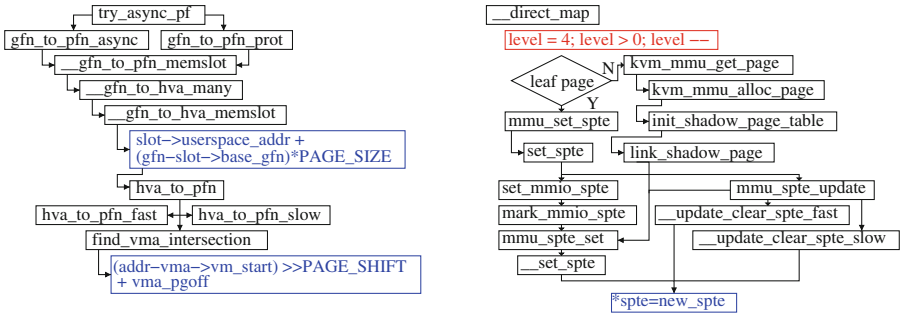

Fig. 5. `pfn` calculation for faulting address and the mapping into TDP tables

the fault is fixed by the guest, another page fault in the second dimension of the TDP is triggered due to the missing entry in TDP table.

`tdp_page_fault` is the page fault handler in this case. As illustrated by Fig. 5, first the host page frame number - `pfn` is calculated for the faulting address through a chain of functions in `try_async_pf`. The `pfn` is then mapped one level after another into the corresponding positions of the TDP tables by the function `__direct_map`. In a predefined format, the entry for a faulting address is split into pieces of `PML4E`, `PDPE`, `PDE`, `PTE` as well as offset in a page. During the loop, `iterator` - an instance of the structure `kvm_shadow_walk_iterator` is used to retrieve the latest physical, virtual addresses and position in the TDP tables for a given address, of which `iterator.level` determines the number of times for the mapping process.

4 Design and Implementation of the Simplified TDP

Although the conventional TDP shown in Fig. 6 is mature and the default configuration for better performance, for a certain kind of workloads the limitation is still obvious. They may suffer large overhead due to walking into the second dimension of multi-level page table upon heavy TLB-miss. It is ideal to have a "flat" TDP table by which the wanted `pfn` can be obtained with a single lookup. Unfortunately, there has long been a problem to allocate large chunk of physically continuous memory in the kernel space. Three functions, namely `vmalloc()`, `kmalloc()` and `__get_free_pages` are used to allocate memory in the current Linux Kernel. The first allocates memory continuous only in virtual address, which is easier to perform but not desired dealing with performance. The second and the third allocate memory chunk continuous in both virtual and physical addresses, however, the maximum memory size allocated is quite limited, thus tends to fall short of the expectation for this purpose. In addition, `kmalloc()` is very likely to fail allocating large amount of memory, especially in low-memory situations [11]. The amount of memory `__get_free_pages` can allocate is also limited within $2^{MAX\_ORDER-1}$, where `MAX_ORDER` in the current Linux Kernel for x86 is 11, which means that each time at most 4MB memory

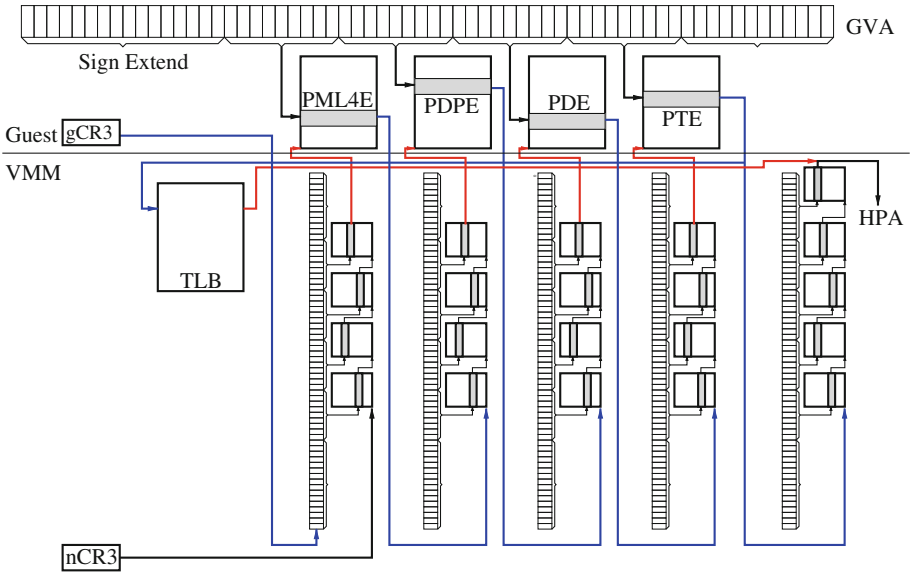

Fig. 6. The conventional TDP with 4 paging levels

can be obtained in the hypervisor. In this condition what we could do is to make the TDP table as "flat" as possible, and to reduce the number of paging with it. Here "flat" means large and physically continuous memory chunk for TDP table. Instead of having thousands of TDP tables managed by their own `kvm_mmu_page` instances, we want to merge as many TDP tables as possible into a larger table managed by fewer `kvm_mmu_page` instances.

There could be various ways to implement this, depending on how the indices of a page table entry are split. Two things are to be noticed for this: 1. to leave the indices for paging as long as possible, and 2. to reuse the current source code for KVM as much as we can.

Consequently, we come up with a quite straightforward design by merging the bits for currently used indices within a guest page table entry. As Figs. 7 and 8 depict, the former PML4, PDP (higher 18 bits) could be combined as a single index to the root of a TDP table segment, and similarly PD and PT (lower 18 bits) as the index for a physical page. By filling the TDP table entries in a linear ascend order for the GPPFN, the HPPFN could be obtained conveniently by a single lookup into the table. As a result, for the currently used maximal address space of a the 64-bit(48 bits effectively in use) guest, we may have $2^{18} = 256K$ segments for the TDP tables, with the index of each segment ranging from 0 to $2^{18} - 1$ to the host physical pages. The TDP table size is enlarged by 2^9 times, while the number of the table segments could be reduced to $\frac{1}{2^9}$ of the former.

This is actually a fundamental change to the current `mmu` implementation. Several data structures and functions oriented to the operations upon $4KB{*}2^9 =$

$2MB$ TDP page table must be adopted to the type upon $4KB * 2^{18} = 1GB$. For example, as depicted by Fig. 9, the data structure of `kvm_mmu_page` could be modified as following to reflect the change: 1. since in a "flat" table, there is only two levels and a single root table as parents, the parents-children relation is quite obvious. Besides, all the first level pages have a common parent but no children at all. Members such as `unsync_children`, `parent_ptes` and `unsync_child_bitmap` are not necessary; 2. members as `gfn`, `role`, `unsync` etc. are multiplied by 512 to hold the informations previously owned by an individual $4KB * 2^9 = 2MB$ page table; 3. `spt` points to a table segment covering an area of $4KB * 2^{18} = 1GB$; 4. `link` is moved to a newly introduced structure - `page_entity` to identify the $4KB * 2^9 = 2MB$ pages that are either in the active or zapped_obselete list. By modifying it this way, the depth of the TDP table hierarchy could be reduced from 4 to 2, while the width expanded from 2^9 to 2^{18}.

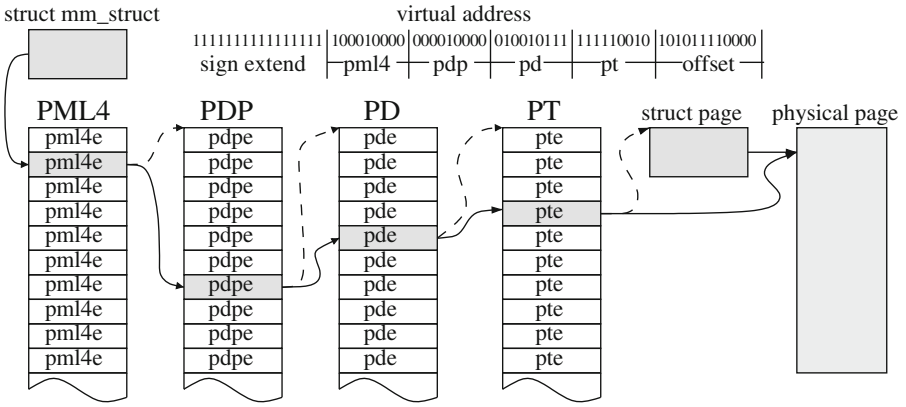

Fig. 7. Paging mode in the second dimension of conventional TDP [12].

Since each `kvm_mmu_page` instance contains 2^{18} table entries now, there will be less `kvm_mmu_page` instances in use, which means that 2^{18} rather than 2^9 `sptes` need to be mapped to a single `kvm_mmu_page` instance. This could be achieved by masking out the lower 30 bits of an address and setting the obtained page descriptor's `private` field to this `kvm_mmu_page` instance, as shown in Fig. 10. Other major affected functions include 1. `shadow_walk_init`, 2. `kvm_mmu_get_page`, 3. `__direct_map`, 4. `kvm_mmu_prepare_zap_page`, 5. `kvm_mmu_commit_zap_page` and 6. `mmu_alloc_direct_roots`.

Taken a guest commonly with 4GB memory as an example. A page contains $4KB/8B = 512$ entries, and for the 4GB, $4GB/4KB = 2^{20}$ entries are needed, so $2^{20}/512 = 2048$ pages of 4KB size should be used to save all the table entries. All together it makes a space of about $4KB * 2048 = 8MB$ size. Although this may be far more than in the conventional TDP case, it is a modest demand and acceptable compared with a host machine configured with dozens of GB RAM.

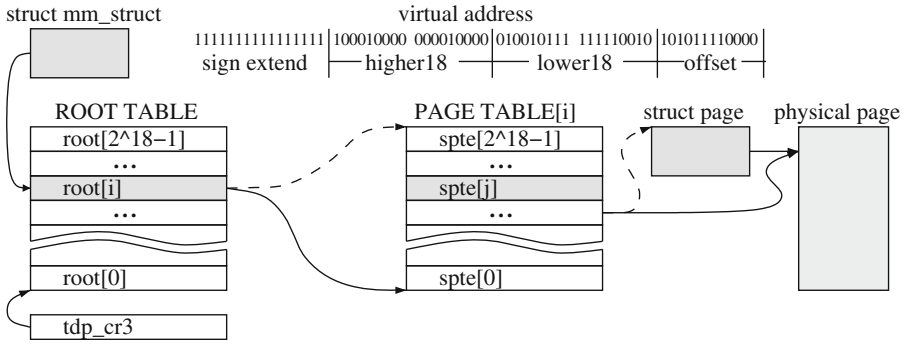

Fig. 8. Paging mode in the second dimension of simplified TDP.

```
struct kvm_mmu_page {
//struct list_head link;
  struct hlist_node hash_link;
  gfn_t gfn[512];
  union kvm_mmu_page_role role[512];
  u64 *spt;
  gfn_t *gfns;
  bool unsync[512];
  int root_count[512];
//unsigned int unsync_children;
//unsigned long parent_ptes;
  unsigned long mmu_valid_gen[512];
//DECLARE_BITMAP(unsync_child_bitmap, 512);
#ifdef CONFIG_X86_32
  int clear_spte_count[512];
#endif
  int write_flooding_count[512];
};
```

```
struct page_entity {
  struct list_head link;
  gfn_t gfn;
  union kvm_mmu_page_role role;
  bool unsync;
  u64 spte;
  int root_count;
};
```

Fig. 9. (a) modified `kvm_mmu_page`. (b) newly defined `page_entity` as the new entity for hash lists, as a replacement of the former `kvm_mmu_page`.

On the other hand, with the 2MB TDP large pages, only 4 `kvm_mmu_page` instances are sufficient to cover the entire 4GB address space. Only 4 entries are filled in the root table, which poses no pressure at all to the TLB. For an arbitrary guest virtual address, at most $2 * 5 + 4 = 14$ (10 in hypervisor, 4 in guest) memory accesses are enough to get the host physical address - far less than that of the current translation scheme (20 in hypervisor, 4 in guest). With a flatter TDP page table and reduced number of memory access, the KVM guest is expected to be less sensitive to workloads and yield higher performance.

5 Conclusion and Further Work

We studied the current implementation of the SPT and TDP for the KVM, and attempted to simplify the second dimension paging of the TDP based on a change of the table structure and the related functions in the hypervisor. With

```
#define BASE_INDEX_MASK ~(u64)((1ULL << 30) - 1)
#define BASE_INDEX(addr) ((u64)(addr) & BASE_INDEX_MASK)
static inline struct kvm_mmu_page *page_header(hpa_t shadow_page)
{
        struct page *page = pfn_to_page(shadow_page >> PAGE_SHIFT);
        return (struct kvm_mmu_page *)page_private(page);
}

static struct kvm_mmu_page *kvm_mmu_alloc_page(struct kvm_vcpu *vcpu, int direct)
{
  struct kvm_mmu_page *sp;
  sp = mmu_memory_cache_alloc(&vcpu->arch.mmu_page_header_cache);
  sp->spt = mmu_memory_cache_alloc(&vcpu->arch.mmu_page_cache);
  if (!direct)
    sp->gfns = mmu_memory_cache_alloc(&vcpu->arch.mmu_page_cache);
  set_page_private(virt_to_page(BASE_INDEX(sp->spt)), (unsigned long)sp);
  return sp;
}
```

Fig. 10. Way to map 2^{30} addresses to a single `kvm_mmu_page` instance

this change on software, the current TDP paging level could be reduced and the overall guest performance will be improved. We have implemented a part of this design and found that, the large TDP page table could be allocated without problem as long as the amount is less than 4MB. However, as it is a relative radical change to the traditional mainstream KVM source code, many functions within the `mmu` code are affected, an executable implementation as well as a benchmark result are unfortunately not yet available. In future we will keep on engaging with this task and work out a concrete solution based on this design.

References

1. Preiss, B.R., Eng, P.: Data Structures and Algorithms with Object-Oriented Design Patterns in Java. Wiley, Chichester (1999)
2. Hoang, G., Bae, C., Lange, J., Zhang, L., Dinda, P., Joseph, R.: A case for alternative nested paging models for virtualized systems. Comput. Archit. Lett. **9**, 17–20, University of Michigan (2010)
3. Wang, X., Zang, J., Wang, Z., Luo, Y., Li, X.: Selective hardware/software memory virtualization, VEE 2011, Department of Computer Science and Technology, Beijing University, March 2011
4. Bae, C.S., Lange, J.R., Dinda, P.A.: Enhancing virtualized application performance through dynamic adaptive paging mode selection, Northwestern University and University of Pittsburgh, ICAC 2011, June 2011
5. Ahn, J., Jin, S., Huh, J.: Revisiting hardware-assisted page walks for virtualized systems. Computer Science Department, KAIST, ISCA 2012, April 2012
6. Gandhi, J., Basu, A., Hill, M.D., Swift, M.M.: Efficient memory virtualization. University of Wisconsin-Madison and AMD Research, October 2014
7. Adavanced Micro Devices Inc, AMD-V Nested Paging White Paper. Adavanced Micro Devices, July 2008

8. Bhargave, R., Serebin, B., Spadini, F., Manne, S.: Accelerating two-dimensional page walks for virtualized systems. Computing Solutions Group and Advanced Architecture & Technology Lab, March 2008
9. Barr, T.W., Cox, A.L., Rixner, S.: Translation Caching: Skip, Don't Walk (the Page Table), Rice University, June 2010
10. Linux kernel Documentation about MMU in KVM. https://www.kernel.org/doc/Documentation/virtual/kvm/mmu.txt
11. Johnson, M.K.: Memory allocation. Linux Journal, issue 16, August 1995. http://www.linuxjournal.com/article/1133
12. Rubini, A., Corbet, J.: Linux Device Drivers, 2nd edn, June 2014. http://www.xml.com/ldd/chapter/book/ch13.html