SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

January 2017, Vol. 60 012103:1–012103:16
doi: 10.1007/s11432-015-0764-7

# Evaluating the impacts of hugepage on virtual machines

Xiaolin WANG[1], Taowei LUO[1], Jingyuan HU[1], Zhenlin WANG[2] & Yingwei LUO[1]*

[1]*School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871*, China;*
[2]*Department of Computer Science, Michigan Technological University, Houghton, MI* 49931*, USA*

Received March 25, 2016; accepted May 10, 2016; published online November 29, 2016

**Abstract** Modern applications often require a large amount of memory. Conventional 4KB pages lead to large page tables and thus exert high pressure on TLB address translations. This pressure is more prominent in a virtualized system, which adds an additional layer of address translation. Page walks due to TLB misses can result in a significant performance overhead. One effort in reducing this overhead is to use hugepage. Linux kernel has supported transparent hugepage since 2.6.38, which provides an alternate large page size. Generally, hugepage demonstrates better performance on address translations and page table modifications. This paper first analyzes the impact of hugepage on native system, and then, compares the impact of hugepage on different memory virtualization approaches: hardware-assisted paging (HAP), shadow paging, and para-virtualization. We observe that the current implementation of transparent hugepage is inefficient. It cannot exploit the full performance advantage of hugepages. Worse yet, the conservative strategy of transparent hugepage may conflict with existing OS functions, which can lead to performance degradation. So, we propose a new memory allocation strategy, alignment-based hugepage (ABH) that promotes hugepage allocations. We apply ABH to different paging modes in virtualized systems. The results show that the new allocation strategy can significantly reduce TLB misses and up to 90% page walk cycles due to TLB misses and thus improve the performance in real world applications.

**Keywords** hugepage, memory management, translation lookaside buffer, virtualization, performance

**Citation** Wang X L, Luo T W, Hu J Y, et al. Evaluating the impacts of hugepage on virtual machines. Sci China Inf Sci, 2017, 60(1): 012103, doi: 10.1007/s11432-015-0764-7

## 1 Introduction

Virtualization is a core technology in cloud computing. Many datacenter services run on virtual machines. Modern applications often have large working set sizes and require large amount of memory. The large footprints of these applications lead to large page tables and thus put a lot of pressure on TLB (Translation Lookaside Buffer) address translation. TLB misses and the miss penalties become a key bottleneck for a series of memory-intensive applications. This bottleneck is more severe in virtualized systems which introduce an additional layer of address translation.

The TLB pressure can be mitigated by using large page sizes. Many processors support multiple page sizes when mapping a program's virtual memory to physical memory. Recent Intel processors can establish

* Corresponding author (email: lyw@pku.edu.cn)

4KB, 2MB or 1GB page mappings by setting the Page Size Extension (PSE) flag in the corresponding level of page table entry. A page larger than a 4KB *regular page* is referred to as a *hugepage*. Hugepages can deliver better performance than regular 4KB pages in several ways. Firstly, a hugepage's TLB entry covers more address space, which will typically lead to fewer TLB misses. Secondly, a hugepage access takes less memory accesses in page walk on a TLB miss. Finally, using hugepages can reduce the number of page faults because a single page fault for a hugepage can now cover more address mappings than a fault for a regular page.

To take advantage of hugepages, software support is required. The Linux kernel has introduced a simple implementation of hugepage, *transparent hugepage* (THP), since kernel 2.6.38 1). It can automatically establish a hugepage mapping by tagging the page table's Page Middle Directory (PMD) entry as a hugepage. If a hugepage mapping fails to build, the kernel would allocate a regular page instead. THP can only be applied to anonymous pages up to the recent Linux versions.

In a virtualized system, an extra layer address translation is introduced to map a guest OS's physical memory addresses to machine addresses. The extra layer heavily affects the memory performance. The influence depends on the memory access pattern of an application as well as the memory virtualization approach of the system. Hugepage may also affect the performance—the impact varies in different virtualization environments as well. This study systematically evaluates the impact of hugepage on a virtualized system and proposes a software enhancement to improve hugepage allocation ratio in which more pages can be allocated as hugepages. Specifically, our work makes the following contributions:

- We measure and analyze the memory system performance on native and on several virtualization environments.
- We analyze the limitations of the current implementation of THP on alignment, show its inefficiency, and propose an alignment-based hugepage (ABH) approach to improve hugepage coverage.
- We evaluate the performance of ABH. The result shows that shadow paging on KVM has the best memory performance among all virtualization environments. The result also suggests the needs to support hugepage in shadow paging for Xen.

The rest of the paper is organized as follows. In Section 2, we analyze the performance of THP and show that the current implementation of THP can improve the performance, but the implementation is ineffective. In Section 3, we compare the performance of THP on different virtualization environments and explain why their performance is different. In Section 4, we optimize THP and attempt to exploit the full performance advantage of hugepage. And in Section 5, we evaluate our optimization scheme in those virtualization environments with our improvement. Section 6 makes some discussions about our optimization and the limitation of hugepage. Section 7 discusses related work. Finally, Section 8 concludes.

## 2 Overview of transparent hugepage

In order to measure the impact of THP, we use a computer which has an Intel i7-3770 processor with 32 GB of memory. The CPU has 512 4KB-page TLB entries for regular page and 32 2MB-page TLB entries for hugepage. The operating system is 64-bit CentOS 6, with Linux kernel 3.6.3. We use Ptmalloc as the memory allocator 2). Since we use a 64-bit system, the hugepage size is 2 MB. We select a set of benchmarks from SPEC CPU2006 [1] and Parsec 3.0 [2] where the time spent on TLB page walk costs more than 5% of the total execution time in native mode with regular page. In addition, we choose *gcc* and *dedup*, both of which cause lots of page faults.

### 2.1 The performance of transparent hugepage

Hugepage can improve the performance by reducing TLB and page fault overhead. In native mode, the cost of page fault handling is not as significant as in some virtualization environments discussed later.

1) Andrea Arcangeli. Transparent hugepage support. In: KVM Forum, 2010.
2) Wolfram Gloger. Ptmalloc, 2006.

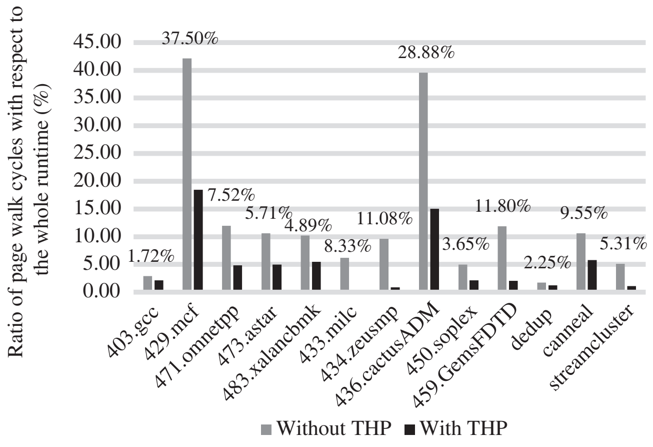

**Figure 1** The TLB impact of transparent hugepage (THP) in native mode.

**Table 1** Page fault counts in native mode

| Benchmark | # Regular page faults | # Hugepage faults | Total | Memory allocated (MB) | *Alloc.Ratio* (%) |
|---|---|---|---|---|---|
| 403.gcc | 679035 | 3862 | 682897 | 10376.48 | 74.44 |
| 429.mcf | 1574 | 835 | 2409 | 1676.15 | 99.63 |
| 471.omnetpp | 46702 | 0 | 46702 | 182.43 | 0.00 |
| 473.astar | 153892 | 105 | 153997 | 811.14 | 25.89 |
| 483.xalancbmk | 124534 | 0 | 124534 | 486.46 | 0.00 |
| 433.milc | 876160 | 17560 | 893720 | 38542.50 | 91.12 |
| 434.zeusmp | 471 | 254 | 725 | 509.84 | 99.64 |
| 436.cactusADM | 476194 | 323 | 476517 | 2506.13 | 25.78 |
| 450.soplex | 59761 | 173 | 59934 | 579.44 | 59.71 |
| 459.GemsFDTD | 38761 | 358 | 39119 | 867.41 | 82.54 |
| dedup | 3404457 | 675 | 3405132 | 14648.66 | 9.22 |
| canneal | 195282 | 87 | 195369 | 936.82 | 18.57 |
| streamcluster | 2832 | 49 | 2881 | 109.06 | 89.86 |

The performance improvement by THP mainly comes from reducing TLB miss page walking. We count the CPU cycles pending on page walk as well as the total execution time in term of cycles. Figure 1 shows that in native mode, the page walk cycles can be significantly reduced by THP. The percentages above the bars show the performance improvements by THP.

Although THP shows significant performance improvement, we observe that the current implementation does not fully exploit the advantage of hugepage. We introduce a metric, *Alloc.Ratio*, to measure the portion of pages that are allocated as hugepages. This metric can be interpreted as the coverage of hugepage. *Alloc.Ratio* is calculated by dividing the space of hugepages by the total space of both hugepages and regular pages allocated during the page fault handing process. Note that 512 regular (4KB) pages are equal to 1 hugepage (2 MB). So *Alloc.Ratio* is equal to $\frac{512\times hpgf}{512\times hpgf+rpgf}$, where *hpgf* and *rpgf* are the number of hugepages and regular pages allocated during page fault process, respectively. Note that an application can also release memory during its execution. Therefore, *Alloc.Ratio* is not the real distribution of regular pages and hugepages in page table, but an approximate metric.

Table 1 shows how many regular page faults and hugepage faults occur in each benchmark. Based on the page fault counts, we calculate how much memory is allocated in page fault handling process and *Alloc.Ratio*. As shown in Table 1, *Alloc.Ratio* is far away from 100%. Even more, there are six benchmarks whose *Alloc.Ratio* is below 50%. *Alloc.Ratio* for *dedup* is only 9%, a great amount of memory is allocated in regular 4KB pages which causes large overhead on page fault handling process. For other benchmarks,

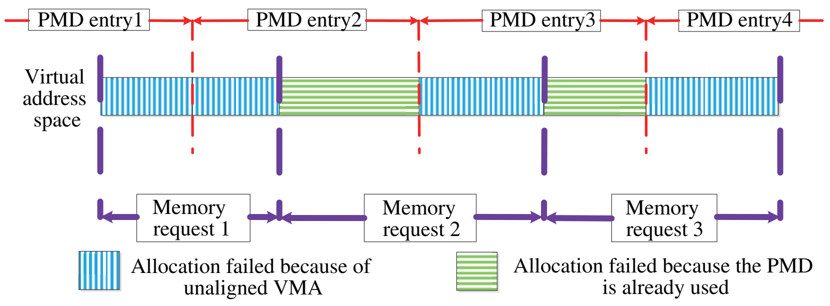

**Figure 2** (Color online) A demonstration on PMD pollution.

their low *Alloc.Ratio* attributes to low number of hugepage allocations. As we know, hugepage usually leads to better performance on page walk than regular page, so next, we will investigate why these benchmarks fail to allocate hugepages and thus still suffer significant TLB overhead.

## 2.2 The usage of transparent hugepage

In this section, we explain why the OS kernel fails to allocate hugepages sometimes. Any application must reserves memory addresses from the OS before accessing them, usually by invoking system calls such as *mmap* and *sbrk*. The OS kernel allocates virtual memory in the application's address space and creates a VMA (virtual memory area) to track the virtual memory space. Physical memory will not be allocated until the virtual memory in a VMA is accessed to trigger a page fault. In the page fault handling process, the kernel records the virtual address *addr* which causes the fault and performs the following checks:

(1) Does the *addr* belong to an existing VMA?

(2) Is the memory in the VMA anonymous?

(3) Is the PMD (Page Middle Directory) entry of *addr* entirely covered by the VMA?

(4) Is the PMD entry of *addr* empty?

If the first check fails, the application might access an illegal address and the kernel will terminate the process. If the second check fails, the accessed memory may be mapped to a file or already be swapped out. THP does not consider this type of memory. Then, if both the third and fourth checks pass, the system will allocate a hugepage for address *addr*. Otherwise, a regular page will be allocated.

Figure 2 shows how PMDs can be polluted in such process, which causes failures on hugepage allocation. In our benchmarks, most failures on hugepage allocation are due to rules (3) and (4). An application might repeatedly request a small piece of memory and then access it. On the first memory access, the kernel will not allocate hugepage because the requested memory is not large enough for a hugepage. In the next iterations, the kernel will allocate virtual memory areas near the last request, and might combine the VMAs into a larger one. The combined VMA may cover several PMDs eventually, but the kernel is still not able to allocate hugepages in the page fault handling process, because the PMD is not empty and has been used for mapping regular pages in previous page faults.

Hugepage promotion can influence the distribution of regular pages and hugepages in page table. There is a daemon process *khugepaged* which scans the application's memory and try to promote regular pages to hugepage if possible. But the promotion cannot fully eliminate the TLB overhead as it is hard to find a suitable frequency to scan and promote. If it works on a high frequency, it would cause overhead on scanning. On the other hand, if the scanning frequency is too low, the application would suffer the overhead of regular page in a longer period. The default frequency for scanning and promoting is 8 hugepages per 10 seconds. It is quite slow. Lots of extra TLB misses may happen before a page is promoted. What's more, it does nothing to alleviate the page fault overhead.

The memory release can also influence the ratio of regular page and hugepage. If lots of regular pages are released and thus hugepages dominate, the TLB overhead would be small. But this situation seldom happens. The upper layer has no idea about which part of memory is mapped with regular pages or hugepages, so it cannot accurately release in concert with the OS kernel to maintain a high hugepage coverage. Indeed, memory release may break a hugepage if the address is unaligned to hugepage size. This hugepage demotion can create lots of regular pages which will damage TLB performance and introduce additional overhead on page table modification. *dedup* is such a case that suffers from memory release, which will be discussed in Subsection 3.3.

In summary, THP can improve the overall performance by using hugepage to reduce the CPU pending time on page walks. But its implementation is inefficient because the current memory management library typically does not generate hugepage-aligned memory requests to the kernel and thus breaks the alignment requirement for hugepage allocation.

## 3 Overview of memory virtualization

Memory virtualization involves page tables that translate among virtual addresses, physical addresses, and machine addresses. We name the page table that maps a guest OS's virtual addresses to physical addresses as the *v2p* table, and the page table that maps a guest OS's physical addresses to machine addresses as the *p2m* table. The *p2m* table is managed by a virtual machine monitor (VMM). A virtual machine can be full-virtualized or para-virtualized. In a full-virtualized virtual machine, the guest OS is unaware that it is running above a virtual machine and the source code is intact for guest OS virtualization. In contrast, in a para-virtualized virtual machine, the guest OS knows it is above a virtual machine. The kernel of the guest OS can be modified for some privilege operations, for example, page table updates.

In a full-virtualized system, memory virtualization can be implemented through hardware-assisted paging (HAP) or shadow paging. In HAP mode, the Memory Management Unit (MMU) can traverse the *v2p* table and *p2m* table for address translation. During page walk, the MMU first walks through the *v2p* table and then the *p2m* table for each level of address translation. The two-dimensional page walk needs more memory accesses to complete an address mapping, which yields much higher penalty than page walk in a native system [3].

Compared to HAP, shadow paging is a pure software approach. In shadow mode, the VMM maintains a *v2m* table that directly maps guest virtual addresses to machine addresses. This one-dimensional *v2m* page table is directly used by the MMU for page walk. The page walk due to a TLB miss is as fast as it is in native mode. However, this page table is transparent to the guest OS. When the guest OS tries to modify its own *v2p* page table, it traps to the VMM. The VMM takes the responsibility for the *v2m* page table updates. The trap would cause a process level context switch, which sometimes needs to flush the whole TLB depending on the architecture and virtual machine implementation. If the application has a lot of page table updates, there would be a noticeable overhead on context switches.

In para-virtualization, the situation is similar to shadow paging. The guest directly maintains a *v2m* table. The MMU uses this one-dimensional page table for page walk. Because the guest OS is aware of the existence of the VMM and the extra *p2m* mapping, when the guest OS tries to modify the page table, it consults with the VMM and gets the real machine address. The guest OS fills the machine address to the page table. VMM calls are injected into the guest OS's kernel so that context switch is avoided. Page fault handling in para-virtualization is faster than shadow paging. But the system needs to maintain the extra *p2m* page mapping, and can result in lots of memory accesses when the table is queried. Therefore, its performance is worse than HAP and native. What's more, it has not yet supported hugepage up to date. The TLB overhead is significant.

When comparing several memory virtualization implementations, para-virtualization does not support hugepage, so its performance is the worst. As for HAP and shadow paging, shadow paging has better performance on page walking, because HAP uses two-dimensional page table which can significantly increase memory accesses for one page walk. But HAP has better performance on modifying page table, because modifying page table in shadow paging would cause context switches and TLB flushing.

We evaluate hugepage on the native system and two popular VMMs, Xen and KVM. The combination of VMM and memory virtualization paging mode yields five types of virtualization environments, HAP on Xen and KVM, shadow paging on Xen and KVM, and para-virtualization on Xen. We name them *XenHAP*, *XenShadow*, *KVMHAP*, *KVMShadow* and *para-virt*, respectively. Their support for hugepage

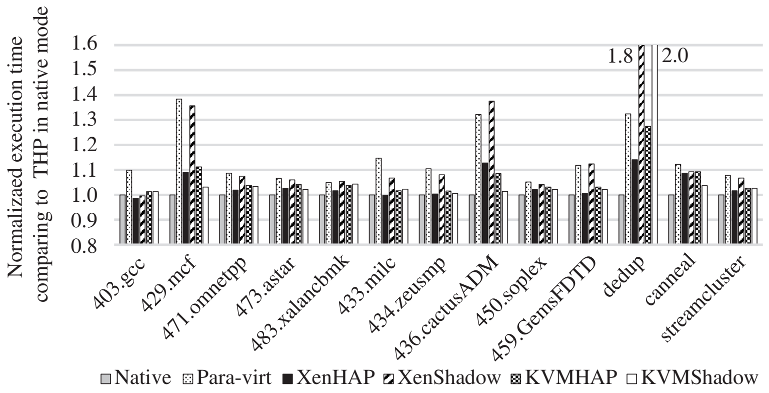

**Figure 3** Execution time comparison of different paging modes.

is different. Xen does not support hugepage in the *p2m* layer of shadow paging and para-virtualization. But hugepage is supported in both *v2p* and *p2m* layers in XenHAP, KVMHAP and KVMShadow. Next we measure their performance with our selected benchmarks and show how their implementation would cause different performance.

## 3.1 Workload performance on virtualization environments

We use the best configuration the current system can support. In KVM, THP (transparent huge page) is enabled in both *v2p* and *p2m* layers. In Xen, we enable THP in the *v2p* layer. The *p2m* layer in Xen is mapped with hugepages in HAP, but with regular pages in shadow paging. In para-virtualization, hugepage is not supported in both layers by THP.

Figure 3 shows the execution times normalized to the native mode with THP enabled. In most cases, the performance of *para-virt* and *XenShadow* in Xen is worse than others as Xen does not support hugepage in these two paging modes[3]. The overhead on TLB misses is much higher in these two settings. In HAP, the performance is almost the same between KVM and Xen as they both benefit from hugepage and have the same paging mode. However, *KVMShadow* shows best performance in many cases. It outperforms *KVMHAP* or *XenHAP* because page walk latency in one-dimensional page table is lower than two-dimensional. A special case is *dedup* where *KVMShadow* shows the worst performance. *dedup* triggers lots of memory release operations which cause a great number of hugepage demotions and page table modifications. In shadow paging mode, page table modification is 2–3 times slower than native mode and HAP mode.

## 3.2 Characteristics of benchmarks

The characteristics of a benchmark can affect its performance on different virtualization environments. Generally, the number of page faults an application raises (with or without hugepage) and the time an application spends on page walking due to TLB misses are two key characteristics that affect paging and memory virtualization performance.

Three benchmarks, *403.gcc*, *433.milc* and *dedup* are sensitive to the speed of page fault handling. The total memory allocated by page fault handling process in these three benchmarks exceeds 10 GB, while others never exceed 3 GB and the page fault handling time for them is negligible. Hugepage can reduce the number of page faults and alleviate the overhead on *403.gcc* and *433.milc*, while *dedup* still shows a significant overhead on page fault even if THP is in use. The behavior of *dedup* can be explained by its excessive number of page faults, which will be discussed further in Subsection 3.3.

As for TLB overhead, Figure 4 shows the ratio of the cycles for page walk time due to TLB misses compared to the total execution time. Generally, *para-virt* and *XenShadow* have much more overhead

3) Actually, even though THP is enabled in *XenShadow*'s guest OS, Xen does not support hugepage in its *p2m* layer, the final *v2m* page table is still mapped with regular pages.

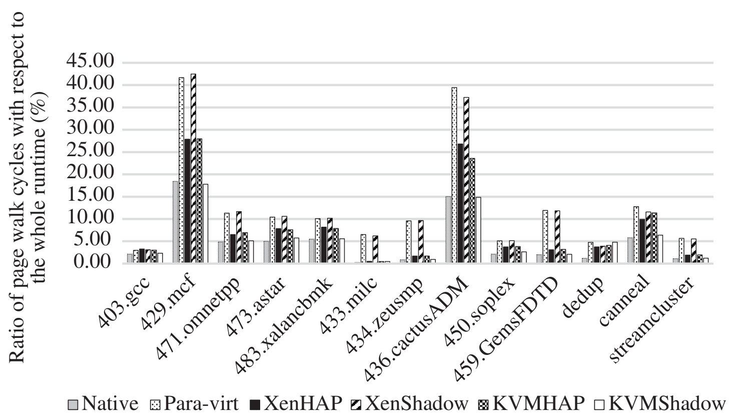

**Figure 4** Ratio of page walk cycles with respect to the whole runtime.

on TLB because they do not use hugepage. *XenHAP* and *KVMHAP* demonstrates better performance on TLB than *para-virt* and *XenShadow* but they perform worse than *KVMShadow* and native.

## 3.3 Performance analysis on virtualization environments

Now, we analyze the performance of individual benchmarks across virtualization environments.

Obviously, *403.gcc* performs worse in *para-virt* than in all other environments. The main reason is that in *para-virt* the guest OS does not use hugepage, which leads to lots of page faults.

In *dedup*, *para-virt* does not use hugepage, so it takes lots of time on page fault handling. We observe that shadow paging is much more slower than HAP. *dedup* is a small benchmark with lots of memory operations (memory allocations and memory releases). It is a case that THP can indeed degrade performance. When *dedup* allocates memory, hugepages are used with the help of THP. Then, when *dedup* releases some memory, the kernel would demote the hugepages due to unaligned release operations. In page demotion, the kernel inserts the missed last level page table, and sets each PTE entry. For each PTE entry update, it traps to the VMM and a context switch occurs. The amount of context switches due to these page table updates are much more significant for *dedup*, which causes performance degradation in shadow paging mode. A further discussion about the overhead on page promotion and demotion is made in Subsection 6.3. HAP and *para-virt* do not preform context switches when page table is modified, but it is still slower than native because of the huge number of page faults.

Both *429.mcf* and *436.cactusADM* are TLB-intensive benchmarks. *para-virt* and *XenShadow* fail to use hugepage, so their performance are much worse than other modes. *KVMShadow* can use hugepage on each layer, so it is faster than HAP because it uses one-dimensional page table. *433.milc* is also a benchmark with a large number of page faults so *para-virt* delivers the worst performance. It is also a TLB intensive benchmark. If hugepage is not used, the overhead would be more significant. THP mitigates most TLB overhead on *433.milc*.

For the remaining benchmarks, as they are all TLB intensive, *para-virt* and *Xenshadow* fail to deliver competitive performance. Hugepage can alleviate the overhead on TLB. But for large workloads, due to large page walk penalty, HAP is slower than *KVMShadow* and native mode.

## 4 Alignment-based hugepage (ABH): design and implementation

We have shown that using hugepage can lead to better performance on TLB as well as page fault. However, the implementation of THP is inefficient and the TLB and page fault handling overhead can still be significant in these cases. This section proposes a new Alignment-based Hugepage (ABH) allocation scheme so as to exploit the full performance benefit of hugepage. In our new approach, hugepage will be allocated at the first memory access to the page, and promotion is no longer necessary. For memory release, we want to make the request aware of hugepage to avoid hugepage demotion.

Our preliminary study indicates that by making aligned memory requests, the kernel can allocate many more hugepages in page fault handling process [4]. This section details the design and implementation of ABH. The key trade-off in ABH design is to balance among hugepage allocation ratio, physical memory consumption and cache conflicts.

## 4.1 Memory management in Linux

By tracing how an application's memory is requested and allocated in the Linux kernel, we find out that *sbrk* and *mmap* are the main system calls which an application triggers for memory requests. Most *sbrk* and *mmap* calls are initiated by *malloc*. PTmalloc in Glibc handles users' malloc requests and calls *sbrk* and *mmap* to request memory from the kernel. Unaligned VMA allocation in *sbrk* and *mmap* is the main reason that the kernel cannot allocate hugepages in page fault process. The number of regular pages allocated in page faults is highly relevant to the number of unaligned VMAs generated in *sbrk* and *mmap*. This indicates that we can use more hugepages by aligning those VMAs in *sbrk* and *mmap*.

## 4.2 Making aligned memory area

The system call *sbrk* is used to manage the heap of an application. The kernel sets the start address of the heap once the process is created. The application requests or releases memory in heap by setting the end address of the heap with *sbrk*. We make the heap start address aligned to hugepage when the kernel initializes the memory space of a process. For a *sbrk* call that requests memory, we extend the heap to make it hugepage-aligned. For a *sbrk* call that releases memory, we reserve extra memory to keep the heap aligned.

The memory allocated in *mmap* may not be aligned in some cases. First, if the size of a memory request is not aligned to hugepage, the rest part cannot use hugepage anyway. Next, when allocating virtual address for *mmap*, the kernel prefers to use memory space adjacent to the last *mmap* request. If the previous *mmap* was not aligned, the current *mmap* may be split into several adjacent PMDs. The memory on two sides may not cover a whole PMD and thus cannot be mapped to a hugepage. An *ad hoc* approach is to extend the length of the *mmap* requests to be hugepage-aligned, and allocate an aligned start address in VMA for *mmap* requests.

It would increase memory consumption significantly if we extended every memory request (small request especially) to hugepage. We choose to sacrifice hugepage coverage a bit to save memory. We do not align heap allocation for the first several megabytes. An application which only uses a few KB of memory in heap will not use hugepages in this case. If the total memory in heap exceeds a given threshold, we perform the hugepage extension since now the application consumes more memory and the ratio of memory waste is controlled. In *mmap*, we do not expand the allocation size unless the unaligned fragment on hugepage exceeds a given threshold. This is useful for small memory requests in *mmap*. We also increase *mmap_threshold* in PTmalloc so that small chunks will be left to heap to avoid memory extension and waste. To summarize, our design avoids low hugepage coverage for large memory applications. Meanwhile, it does not waste too much memory for small applications.

## 4.3 Other details in implementation

The implementation will modify code in kernel as well as in PTmalloc. We modify the kernel to allocate hugepage-aligned start address for VMA in *mmap* and perform memory extension in *sbrk* and *mmap*. The *mmap_threshold* is adjusted in PTmalloc.

A VMA may be adjusted by system calls such as *mremap* and *mprotect*. We keep the boundary of the new VMA aligned to hugepage by extending its length. The extension of heap is transparent to user applications as *sbrk* returns the requested heap top and maintains the actual top by itself. The extended memory can be used for future memory requests. User applications assume that new memory allocated through page faults is automatically set to 0 by kernel. Our extension may break this assumption when we reserve extra memory to make heap hugepage-aligned. When the memory is accessed again, the system neither raises a page fault nor sets the memory to 0. We need to check the released memory and

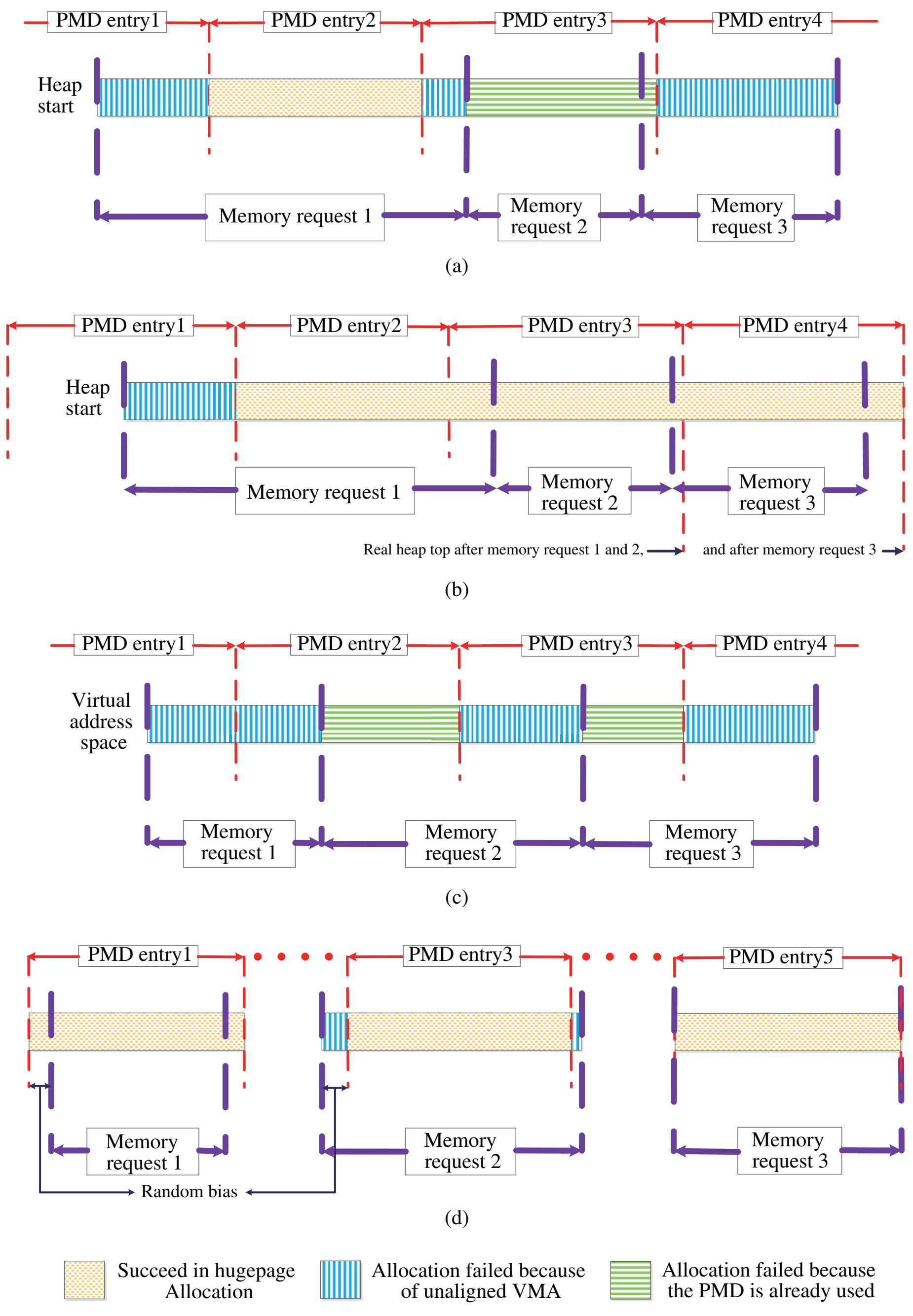

**Figure 5** (Color online) Comparison between THP and ABH. (a) Heap layout of THP; (b) heap layout of ABH; (c) *mmap* layout of THP; (d) *mmap* layout of ABH.

set it to 0. We also need to check whether the memory is mapped before resetting. Otherwise, the kernel will raise a page fault when accessing those memory, which leads to kernel panic.

Alignment may cause conflicts in cache and memory. In some applications, many VMAs have a similar memory accessing pattern. They might use the same cache sets and memory channel that lead to conflicts. We take several approaches to alleviate the conflicts. First, we make a VMA not aligned to its boundary address by randomly leaving some unaligned memory space on both sides. Next we do not return the start address of a VMA if *mmap* extension is performed, by adding a random offset. The offset is acceptable if it is smaller than the amount of memory extended. At last, we generate a random heap start address so that the heap address is not fixed. Figure 5 illustrates our design and compares it to default memory allocation strategy.

**Table 2** Memory waste on hugepage and alignment

| Benchmark | Memory of REG (MB) | Extra | Memory of THP (MB) | Extra | Memory of ABH (MB) |
|---|---|---|---|---|---|
| *456.hmmer* | 24.14 | 0.00 | 24.14 | 0.24 | 29.94 |
| *416.gamess* | 6.59 | 1.47 | 16.27 | 0.00 | 16.27 |
| *435.gromacs* | 13.60 | 0.07 | 14.57 | 0.18 | 17.22 |
| *436.cactusADM* | 622.66 | 0.16 | 725.04 | 0.04 | 753.60 |
| *453.povray* | 2.75 | 0.00 | 2.75 | 0.50 | 4.14 |
| *bodytrack* | 29.32 | 0.00 | 29.38 | 0.35 | 39.80 |
| *ferret* | 96.35 | 0.07 | 103.05 | 0.24 | 127.57 |
| *swaptions* | 2.82 | 0.00 | 2.82 | 1.18 | 6.16 |
| *vips* | 19.01 | 0.12 | 21.27 | 0.35 | 28.73 |

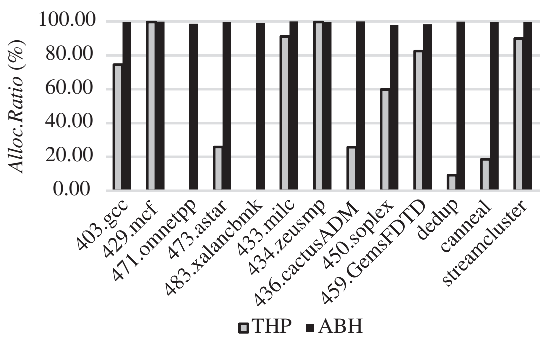

**Figure 6** Improvement in *Alloc.Ratio*.

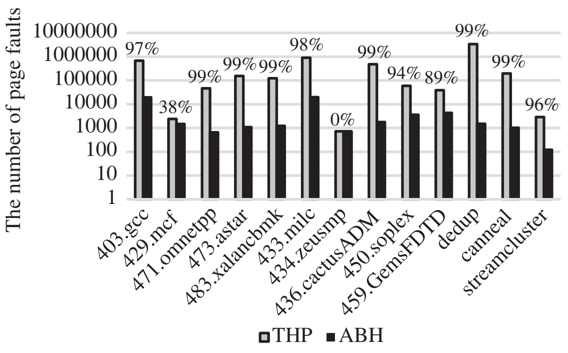

**Figure 7** Reduction in page fault.

## 5 Evaluation of ABH

In our evaluation, we let the first 1 MB of memory in heap grow in normal way and do not align it to hugepage. We change *mmap_threshold* that PTmalloc uses to allocate memory with *mmap* to 4 MB (the default is 128 KB). The extension threshold for *mmap* is set to 1 MB, which suggests that, if the hugepage unaligned fragment exceed 1 MB, we extend it to a hugepage.

In this section, we first evaluate the ABH's influence on page fault reduction, hugepage allocation ratio, and physical memory consumption. Then, we measure its TLB performance, which includes the TLB miss count and page walk time. Next, we show its performance improvement over TPH and make comparison on virtualization environments. The system configuration is described in Section 2.

### 5.1 Memory evaluation

Figure 6 shows the *Alloc.Ratio* improvement by ABH compared to the default transparent hugepage implementation, i.e., THP. ABH significantly improves the hugepage allocation ratio in page fault handling process. Almost all of memory is allocated in hugepage with ABH. Figure 7 shows the changes on page fault count. The percentage on each bar shows the page fault reduction by ABH. The results show that page faults are substantially reduced. For those page fault intensive benchmarks such as *403.gcc*, *433.milc* and *dedup*, ABH reduces page fault numbers by more than 98%.

Our optimization will slightly increase the demand of physical memory. Firstly, we request more memory to make VMA aligned. Secondly, compared to regular pages, hugepages will naturally consume more memory if only a few addresses are accessed in the hugepage. Table 2 shows the memory consumption THP and ABH compared to the system using regular pages only. The second, fourth and sixth columns are the total memory consumption of that without THP (REG), with THP, and with ABH, respectively, in megabytes. The third and fifth columns are the ratio of extra memory introduced by THP and ABH, respectively. Those benchmarks which have no obvious changes in memory cost are not listed in the table.

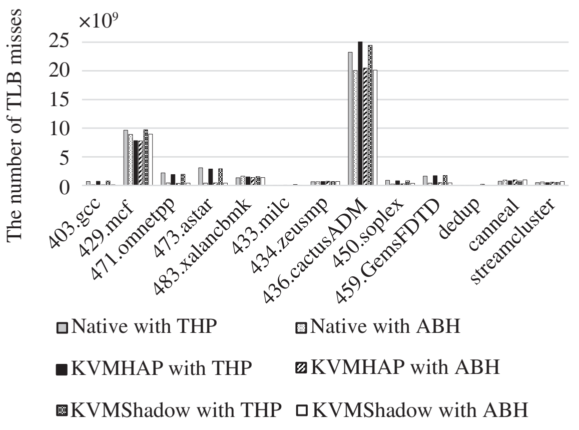

**Figure 8** TLB misses on Intel i7-3770.

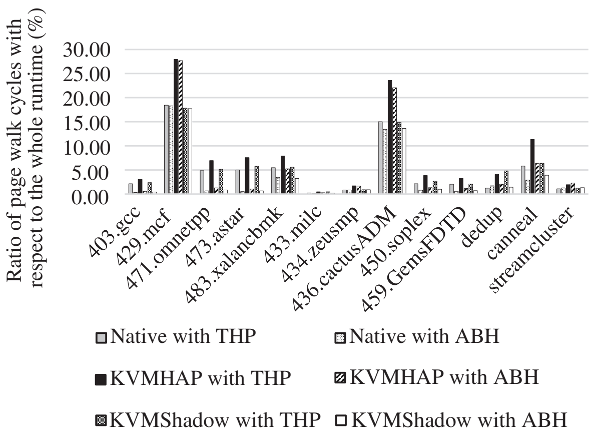

**Figure 9** Ratio of page walk cycles with respect to the whole runtime on Intel i7-3770.

The result shows that large memory waste ratios come from small workloads. Their working set sizes do not exceed 50 MB. Our optimization will increase the demand of physical memory, so does THP. The largest memory overhead of ABH comes from *436.cactusADM* and *ferret*, but it is still tolerable when compared to the extra memory cost by THP.

## 5.2 Evaluation on TLB misses and page walk time

Figure 8 shows the number of TLB misses with THP and ABH. We measure the TLB performance in three environments: *native*, *KVMHAP* and *KVMShadow*. Figure 9 shows the ratio of page walk cycles with TPH and ABH with respect to the total execution time. Comparing these two figures, we have made the following observations.

• ABH can reduce the time for page walk by increasing *Alloc.Ratio*. The benchmarks with significant *Alloc.Ratio* improvement in Figure 6 also show a noticeable reduction on page walk time. For those benchmarks that page walk time is not reduced by ABH, such as *429.mcf*, *433.mile*, *434.zeusmp* and *streamcluster*, their *Alloc.Ratio* with THP is already high. There is not much space for ABH to improve.

• Some benchmarks show reduction on page walk time even though their TLB misses are not reduced, such as *483.xalancbmk*, *dedup* and *canneal*. This is because the number of TLB misses is not the only factor that influences page walk time. Page walk time may be affected by the depth of a page table and whether the page table is cached.

• When comparing across virtualization environments, the difference of TLB miss count is small, but the page walk time in HAP is longer than native and shadow.

• Some benchmarks still have a significant overhead on TLB, such as *429.mcf*, *436.cactusADM*, *483.xalancbmk*, and *canneal*, although hugepages almost fully occupy the page tables in these benchmarks. These workloads are too big to be cached by TLB in the test machine. These benchmarks can be improved by introducing more TLB entries and sharing TLB entries between regular page and hugepage.

## 5.3 Evaluation on performance

Figure 10 shows the overall performance improvement by ABH. In most cases, the performance improvement is highly correlated to the page walk time reduction. It proves that by reducing page walk time, we can receive better performance. A special case is *dedup*, the improvement by ABH is 24% in native, 28% in *KVMHAP* and 54% in *KVMShadow*, which is much larger than the improvement on page walk. *dedup* has an excessive number of page faults. ABH significantly reduces its page fault count, so the performance is improved. In shadow mode, an additional benefit comes from the avoidance of page demotion or promotion. Some benchmarks show a slight performance degradation, such as *streamcluster* and *429.mcf*. Since ABH increases the allocation of hugepage, it may increase the pressure on hugepage TLB. A further discussion about TLB overhead is made in Subsection 6.1.

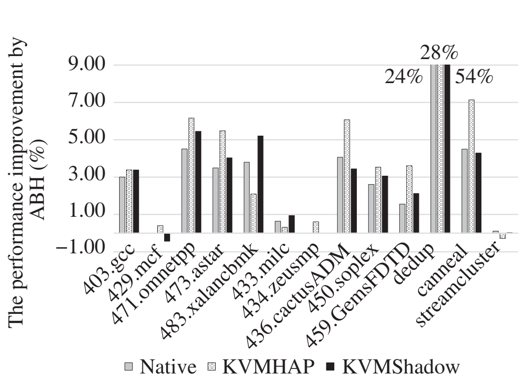

**Figure 10** Performance improvement by ABH.

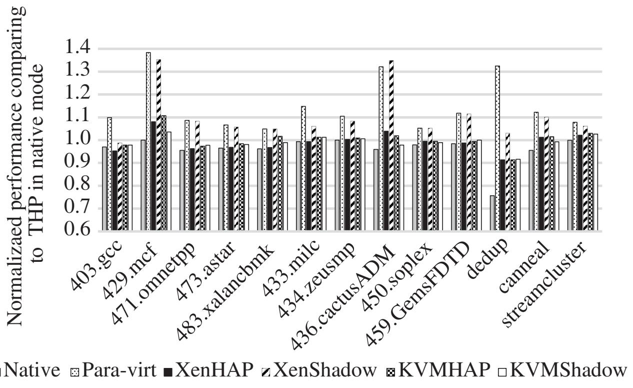

**Figure 11** Performance on different virtualization environments after optimization.

We also test other benchmarks in SPEC CPU2006 and Parsec. They are neither TLB intensive nor page fault intensive. Our optimization dose not show much improvement for them. Since we have carefully tuned the memory allocation scheme for hugepage to minimize the potential overhead, our approach does not cause any noticeable performance degradation either.

### 5.4 Impact of different virtualization environments

We make comparison among the native and several virtualization environments. The base line is native mode with THP. Native mode with ABH, Xen's para-virtualization, HAP and shadow paging with ABH in Xen and KVM are included for our comparison[4]. Figure 11 shows the result.

When comparing HAP and shadow paging within KVM, shadow paging has better performance on big workloads, such as *429.mcf*, *436.cactusADM*, *483.xalancbmk*, and *canneal*. Those benchmarks still have notable overhead on page walk. In all cases, *KVMShadow* is not slower than *KVMHAP*. It suggests that shadow wins the page mode comparison against HAP. But when comparing HAP between Xen and KVM, Xen is better. It seems Xen has little overhead on system virtualization with HAP. Unfortunately, Xen does not support shadow paging well. Currently, KVM's shadow paging mode still shows better performance on big workloads when compared to Xen. *para-virt* and *XenShadow* are still the slowest virtualization environments, as they do not use hugepage as the gap between regular page and hugepage is significant. The comparison shows the need of supporting hugepage in shadow paging for Xen.

## 6 Discussion

The goal of ABH is to provide a general purpose solution to reduce TLB pressure using hugepage. We not only improve the hugepage ratio and TLB performance, but also carefully tailor the memory allocator to avoid excessive memory consumption and take care of the potential overhead that THP or ABH may have. In this section, we discuss some complexities that THP or ABH brings in.

### 6.1 The overhead on separated TLB and hugepage

The Intel i7-3770 processor has 512 4KB-page TLB entries for regular pages and 32 2MB-page TLB entries for hugepages. The TLB entries are dedicated for one specific page size. If we put all memory mappings to hugepage, it has potential to cause many more TLB misses on hugepage TLB and thus degrade the performance. We manually create a memory access pattern that can be cached by regular page TLB but overflows hugepage TLB. Hugepage can cause 46% performance degradation in our stress test.

4) Although ABH have no improvement on *XenShadow*, it does not have noticeable degradation either. We just implement ABH in the guest OS despite that hugepage in the hypervisor will lead to performance improvement evidenced by the results in KVM.

In real cases, we find the TLB misses on some benchmarks increase slightly after our optimization. *483.xalancbmk* is an example. In Figure 8, the TLB misses increases after ABH is applied in native. Moreover, we find that the cycle count for page walk is reduced as shown in Figure 9. It is because that page walk on hugepage is much faster than on regular page. In addition, using hugepage means reducing the amount of last level page table, which mitigates the pressure on page walk cache. For *483.xalancbmk*, we find that the number of L3 references is reduced by 12% in ABH. The average cycles to resolve an L1 cache miss is reduced by 8% in ABH. When running *streamcluster* in *KVMHAP*, we find that the number of TLB misses and the cycles of page walk increase after ABH so the performance degrades. The benefit of hugepage may not fully coincide with the overhead on TLB misses. But this case rarely happens (mostly in Parsec). The overhead of ABH has never exceeded 2% for all benchmarks in SPEC and Parsec with Intel i7-3770 processor.

In a new architecture we tested where TLB entries can be shared between 4KB pages and 2MB pages, the huge TLB overhead never increases with ABH.

### 6.2 Overhead on alignment in ABH

Memory alignment can cause extra address conflicts on cache and memory, which can degrade performance. Some applications might require several memory chunks and access them with the same pattern simultaneously. If we align those chunks to hugepage size (which is 2 MB), the memory accesses on different chunks will be allocated into the same cache sets in all levels. This would cause cache conflicts. In the default OS regular page allocation, chunks are aligned to regular 4KB page. They conflict only in the L1 cache. The memory accesses will be scattered in L2 and L3 cache. If we align all memory chunks to hugepage, this would cause conflicts on multiple levels of cache and hurt the performance. In order to avoid conflicts, we add a random offset for memory chunks. Some benchmarks are affected by this conflict. The worst case is *459.GemsFDTD*, which shows a 60% performance degradation without adding offset.

### 6.3 Overhead on page promotion and demotion

Another overhead of hugepage comes from its promotion and demotion process. In shadow paging, page table modification is expensive. Each promotion or demotion contains lots of page table modifications. In shadow mode (no matter in Xen or KVM), *dedup* shows a 40% degradation with THP. The number of TLB_FLUSH.DTLB_THREAD event is increased by 100% after THP is applied in *KVMShadow*. With ABH, the TLB flush events are reduced by 90% when compared with regular page allocation, and 95% compared with THP. Our optimization can avoid page promotion and demotion and deliver an impressive performance improvement for *dedup*.

## 7 Related work

### 7.1 Support for hugepages

Plenty of work is done to support hugepage in operating systems$^{1)}$ [5, 6]. Lu et al. [7] propose a method to use hugepage in text region with *hugetlbfs*. Romer et al. [8] dynamically promote regular pages to hugepage by trading off the penalties on page walk and hugepage promotion. Some work breaks up the limitation of contiguous memory request for hugepage promotion, which allows us to use hugepage in more situations [9, 10].

### 7.2 Optimization on TLB

Rather than utilizing hugepage, some other studies focus on improving the TLB performance. Talluri et al. [11] propose a new TLB design, which can merge continuous TLB mappings so that one TLB entry will cover as many addresses as a hugepage can. Similarly, entries in the page cache table can be merged to get more coverage for each record [12]. For chip multiprocessor, shared last level TLB can increase the capacity of entries and achieve high TLB performance on both single-threaded and multi-threaded programs [13–15]. Barr et al. evaluate the performance of page cache with different designs [16] and propose speculative translation to accelerate page walk [17]. Papadopoulou et al. [18] predict the page size of a TLB entry, in order to improve TLB performance and reduce energy cost. Basu et al. [19] suggest that we can use segment mapping to avoid page walk latency for big memory servers. Karakostas et al. [20] use redundant memory mapping to reduce TLB overhead for big workloads. Fang et al. [21] compare the copy-based and remapping-based page promotion in single-issue and super-scalar architectures. Other studies focus on using hardware prefetching to reduce the TLB misses [22, 23]. In multiprocessors, TLB accesses in one core can help prefetch for other cores [24, 25].

### 7.3 TLB on virtualization

Adams and Agesen make a comprehensive comparison between software VMM and hardware VMM [26]. In memory virtualization, Bhatia and Nikhil [27] evaluate the performance of Intel's hardware-assisted paging design, Extended Page Tables, and suggest to use hugepage to accelerate page walk.

For hardware-assisted memory virtualization, Buell et al. [28] have shown that the increased overhead on TLB miss handling is the largest contributor to the performance gap between native and virtual servers. Bhargava et al. find that nested page walking in guest OS and virtual machine monitor (VMM) is the main reason that causes the performance gap. They also suggest to use page cache to accelerate page walking [3]. Ahn et al. [29] propose to use a flat page table to perform address translation in VMM. Gandhi et al. [30] suggest to use segment mapping to accelerate nested page walking. Gadre et al. [31] implement the hugepage support in Xen to reduce the page walking in VMM. Speculation is also used in HAP's TLB to accelerate address translation in the *p2m* layer [32].

For software virtualization, the main performance gap comes from VMM traps (VM exits) when page table changes. Wang et al. [33] propose a dynamic switching scheme to choose a better virtualization paging mode dynamically based on applications' behavior. Their another attempt is to reduce VM exits by optimizing memory allocation [34]. In software managed TLB, Chang et al. [35] move the TLB reading and writing operations into the guest OS to reduce the overhead on VM trapping.

## 8 Conclusion

In this paper, we analyze the impact of hugepage on native and virtualized systems. We find that the current implementation of Linux transparent hugepage cannot fully exploit the benefit of hugepage due to alignment and other limitations. We optimize the memory allocation process in Linux kernel and the memory management library. The experiments show our optimization can reduce the number of page faults and page walk time due to TLB misses. Our approach allocates almost 100% hugepages in page fault handling process, almost reaching the upper bound on TLB performance with hugepage. We evaluate our new design in various virtualized systems. The results show that shadow paging has better performance than HAP on large workloads when hugepage can be applied in both the guest OS and the hypervisor. We also find Xen's shadow paging is slower than HAP due to the miss of hugepage implementation in *p2m*, it would be valuable to make Xen support hugepage in its shadow paging mode.

**Acknowledgements** This work was supported by National Natural Science Foundation of China (Grant Nos. 61232008, 61272158, 61328201, 61472008, 61170055), National High-tech R&D Program of China (863) (Grant Nos. 2012AA010905, 2015AA015305), Research Fund for the Doctoral Program of Higher Education of China (Grant No. 20110001110101). Zhenlin WANG is also supported by National Science Foundation (Grant No. CSR1422342).

**Conflict of interest** The authors declare that they have no conflict of interest.

## References

1 Henning J L . SPEC CPU2006 benchmark descriptions. ACM SIGARCH Comput Architect News, 2006, 34: 1–17

2 Bienia C, Kumar S, Singh J P, et al. The parsec benchmark suite: characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. New York: ACM, 2008. 72–81

3 Bhargava R, Serebrin B, Spadini F, et al. Accelerating two-dimensional page walks for virtualized systems. ACM SIGOPS Oper Syst Rev, 2008, 42: 26–35

4 Luo T W, Wang X L, Hu J Y, et al. Improving TLB performance by increasing hugepage ratio. In: Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). Washington, DC: IEEE, 2015

5 Ganapathy N, Schimmel C. General purpose operating system support for multiple page sizes. In: Proceedings of USENIX Annual Technical Conference. Berkeley: USENIX Association Berkeley, 1998. 8

6 Navarro J, Iyer S, Druschel P, et al. Practical, transparent operating system support for superpages. ACM SIGOPS Oper Syst Rev, 2002, 36: 89–104

7 Lu H J, Seth R, Doshi K, et al. Using hugetlbfs for mapping application text regions. In: Proceedings of the Linux Symposium, Ottawa, 2006. 2: 75–82

8 Romer T H, Ohlrich W H, Karlin A R, et al. Reducing tlb and memory overhead using online superpage promotion. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture. New York: ACM, 1995. 176–187

9 Du Y, Zhou M, Childers B R, et al. Supporting superpages in non-contiguous physical memory. In: Proceedings of IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), Burlingame, 2015. 223–234

10 Swanson M, Stoller L, Carter J. Increasing TLB reach using super backed by shadow memory. ACM SIGARCH Comput Architect News, 1998, 26: 204–213

11 Talluri M, Hill M D. Surpassing the TLB performance of super with less operating system support. ACM SIGPLAN Notices, 1994, 29: 171–182

12 Bhattacharjee A. Large-reach memory management unit caches. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. New York: ACM, 2013. 383–394

13 Bhattacharjee A, Lustig D, Martonosi M. Shared last-level tlbs for chip multiprocessors. In: Proceedings of IEEE 17th International Symposium on High Performance Computer Architecture (HPCA). Washington, DC: IEEE, 2011. 62–63

14 Lustig D, Bhattacharjee A, Martonosi M. TLB improvements for chip multiprocessors: inter-core cooperative prefetchers and shared last-level TLBs. ACM Trans Architect Code Optim, 2013, 10: 2

15 Srikantaiah S, Kandemir M. Synergistic tlbs for high performance address translation in chip multiprocessors. In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC: IEEE, 2010. 313–324

16 Barr T W, Cox A L, Rixner S. Translation caching: skip, don't walk (the page table). ACM SIGARCH Comput Architect News, 2010, 38: 48–59

17 Barr T W, Cox A L, Rixner S. SpecTLB: a mechanism for speculative address translation. In: Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA). New York: ACM, 2011. 307–317

18 Papadopoulou M-M, Tong X, Seznec A, et al. Prediction-based superpage-friendly TLB designs. In: Proceedings of IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), Burlingame, 2015. 210–222

19 Basu A, Gandhi J, Chang J C, et al. Efficient virtual memory for big memory servers. ACM SIGARCH Comput Architect News, 2013, 41: 237–248

20 Karakostas V, Gandhi J, Ayar F, et al. Redundant memory mappings for fast access to large memories. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture. New York: ACM, 2015. 66–78

21 Fang Z, Zhang L X, Carter J B, et al. Reevaluating online superpage promotion with hardware support. In: Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA). Washington, DC: IEEE, 2001. 63–72

22 Saulsbury A, Dahlgren F, Stenström P. Recency-based TLB preloading. ACM SIGARCH Comput Architect News, 2000, 28: 117–127

23 Kandiraju G B, Sivasubramaniam A. Going the distance for TLB prefetching: an application-driven study. ACM SIGARCH Comput Architect News, 2002, 30: 195–206

24 Bhattacharjee A, Martonosi M. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In: Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09). Washington, DC: IEEE, 2009. 29–40

25 Bhattacharjee A, Martonosi M. Inter-core cooperative TLB for chip multiprocessors. ACM SIGARCH Comput Architect News, 2010, 38: 359–370

26 Adams K, Agesen O. A comparison of software and hardware techniques for x86 virtualization. ACM SIGPLAN Notices, 2006, 41: 2–13

27 Bhatia N. Performance evaluation of Intel EPT hardware assist. VMware, Inc, 2009. http://www.vmware.com/techpapers/2009/performance-evaluation-of-intel-ept-hardware-assis-10006.html

28 Buell J, Hecht D, Heo J, et al. Methodology for performance analysis of VMware vSphere under Tier-1 applications. VMware Technical J, 2013. 19

29 Ahn J, Jin S, Huh J. Revisiting hardware-assisted page walks for virtualized systems. ACM SIGARCH Comput Architect News, 2012, 40: 476–487

30 Gandhi J, Basu A, Hill M D, et al. Efficient memory virtualization. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). Washington, DC: IEEE, 2014. 178–189

31 Gadre A S, Kabra K, Vasani A, et al. X-xen: huge page support in xen. In: Proceedings of the Linux Symposium, Ottawa, 2011. 7

32 Pham B, Vesely J, Loh G H, et al. Using TLB Speculation to Overcome Page Splintering in Virtual Machines. Technical Report DCS-TR-7132015. Rutgers University, 2015

33 Wang X L, Zang J R, Wang Z L, et al. Selective hardware/software memory virtualization. ACM SIGPLAN Notices, 2011, 46: 217–226

34 Wang X L, Weng L M, Wang Z L, et al. Revisiting memory management on virtualized environments. ACM Trans Architect Optim, 2013, 10: 48

35 Chang X T, Franke H, Ge Y, et al. Improving virtualization in the presence of software managed translation lookaside buffers. In: Proceedings of the 40th Annual International Symposium on Computer Architecture. New York: ACM, 2013. 120–129