Revisiting Memory Management on Virtualized Environments

XIAOLIN WANG and LINGMEI WENG, Peking University
ZHENLIN WANG, Michigan Technological University
YINGWEI LUO, Peking University, lyw@pku.edu.cn

With the evolvement of hardware, 64-bit Central Processing Units (CPUs) and 64-bit Operating Systems (OSs) have dominated the market. This article investigates the performance of virtual memory management of Virtual Machines (VMs) with a large virtual address space in 64-bit OSs, which imposes different pressure on memory virtualization than 32-bit systems. Each of the two conventional memory virtualization approaches, Shadowing Paging (SP) and Hardware-Assisted Paging (HAP), causes different overhead for different applications. Our experiments show that 64-bit applications prefer to run in a VM using SP, while 32-bit applications do not have a uniform preference between SP and HAP. In this article, we trace this inconsistency between 32-bit applications and 64-bit applications to its root cause through a systematic empirical study in Linux systems and discover that the major overhead of SP results from memory management in the 32-bit GNU C library (*glibc*). We propose enhancements to the existing memory management algorithms, which substantially reduce the overhead of SP. Based on the evaluations using SPEC CPU2006, Parsec 2.1, and cloud benchmarks, our results show that SP, with the improved memory allocators, can compete with HAP in almost all cases, in both 64-bit and 32-bit systems. We conclude that without a significant breakthrough in HAP, researchers should pay more attention to SP, which is more flexible and cost effective.

Categories and Subject Descriptors: D.4.2 [Storage Management]: Allocation/Deallocation Strategies

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Virtualization, shadowing paging, hardware-assisted paging, memory allocation

ACM Reference Format:

Wang, X., Weng, L., Wang, Z., and Luo, Y. 2013. Revisiting memory management on virtualized environments. *ACM Trans. Architect. Code Optim.* 10, 4, Article 48 (December 2013), 20 pages.
DOI: <http://dx.doi.org/10.1145/2555289.2555304>

1. INTRODUCTION

System virtualization is widely used in cloud computing and data centers because of the advantages of server consolidation, fault tolerance, performance isolation, security, and maintenance. However, virtualization often results in significant performance overhead where memory virtualization overhead plays a key role. Despite continuous efforts to reduce memory virtualization overhead [Adams et al. 2006; Menon et al. 2006; Santos et al. 2008; Zhao et al. 2009; Wang et al. 2011], it remains significant for certain applications and often the last obstacle among the overall virtualization overhead.

Memory virtualization needs to provide mechanisms to facilitate address translations from guest virtual addresses to machine addresses, which are no longer guest

Author's addresses: XL Wang, LM Weng, and YW Luo, Computer Science Department, Peking University; ZL Wang, Computer Science Department, Michigan Technological University. Corresponding author: YW Luo, lyw@pku.edu.cn, Computer Science Department, Peking University, Beijing, China, 100871.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481 or permissions@acm.org.

© 2013 ACM 1544-3566/2013/12-ART48 \$15.00

DOI: <http://dx.doi.org/10.1145/2555289.2555304>

physical addresses as seen in a native system. The primordial approach of memory virtualization is *Shadow Paging* (SP), a pure software-based approach. The hypervisor, which manages all Virtual Machines (VMs), maintains a *shadow page table* for each process running in a VM, and a physical-to-machine table (*p2m*) used to synchronize the *shadow page table* with the guest page table. A shadow page table is responsible for mapping guest virtual addresses to machine addresses directly. It is linked to the hardware Memory Management Unit (MMU) so that a Translate Lookaside Buffer (TLB) miss could be resolved by walking the single-page table, as in native systems. However, the guest Operating System (OS) still sees and maintains conventional virtual-to-physical page tables. Any updates to the guest page table will cause several context switches (VM exits) from the guest to the hypervisor in order to synchronize with the shadow page table. Thus, any page faults in the guest will result in expensive hypervisor interferences.

To avoid such VM-exit or context-switching overhead, major x86 CPU manufacturers have proposed a hardware-based solution, called *Hardware-Assisted Paging* (HAP) [Bhargava et al. 2008; Gillespie 2009]. Almost all current x86 processors by Intel and AMD introduce two-dimensional page tables (Figure 2) to assist memory virtualization. HAP nests a guest-to-physical page table with a physical-to-machine table. Guest page table updates can be done in the two-dimensional page table without hypervisor intervention. However, resolving a TLB miss in HAP requires more page table accesses, called a *page walk*, compared to SP as discussed in Section 2. In 64-bit operating systems, this TLB-related penalty is more significant compared to 32-bit systems as the page table becomes deeper and the miss frequency grows [Talluri et al. 1992, 1994; Barr et al. 2011].

Our experiments of SPEC CPU2006 [Henning et al. 2009] and other C/C++ benchmarks show that 64-bit applications running in a VM using SP always perform on par with or better than using HAP, while this is not the case for 32-bit applications. Surprisingly, with further experiments oriented to explain this phenomenon, we find that during the execution of the 32-bit applications, the major overhead of SP results from VM exits, many of which stem from dynamic memory management in glibc, a C library that is frequently used by C/C++ applications. Although many researchers have already examined the differences between SP and HAP [Wang et al. 2011; Ahn et al. 2012], they simply ascribe the overhead of SP to hypervisor intervention on page faults. Their studies neither delve further into the root cause of the excessive VM exits nor pay attention to the runtime libraries in guest OSs.

The most significant contribution of our work is to identify a root cause of excessive VM exits in SP for some pathological cases and propose a pure software solution to reduce the VM exits. With the improvement, we show that SP can compete with HAP in both 32-bit and 64-bit systems and often performs better.

The rest of the article is organized as follows. Section 2 provides background for this article. Section 3 analyzes the root cause of excessive VM exits. In Section 4, we present our solution in detail. Section 5 evaluates our software approach. We discuss related work in Section 6 and conclude in Section 7.

2. BACKGROUND

2.1. Overhead of Memory Virtualization

The study in this article focuses on full virtualization systems where the guests are not aware that they are running on virtual machines. The guest page tables only record mappings from virtual addresses to physical addresses. Memory virtualization bridges the gap between physical addresses and machine addresses.

Fig. 1. Memory virtualization. This figure shows two conventional approaches of memory virtualization.

Fig. 2. Page walks of HAP. This figure shows the structure of HAP. To translate a virtual address, the MMU first loads CR3, which contains the physical address pointing to the guest table. Translating it into the machine address of PGD requires a walk through the extended page table, which consists of four memory accesses. To get an entry in PGD requires one memory access. To access PUD, the entry needs to go through the extended page table again to get the machine address of PUD, and so on. Thus, the whole page table walk requires $4 + 4 * 5 = 24$ memory accesses altogether.

Figure 1 illustrates the two memory virtualization mechanisms, SP (on the left) and HAP (on the right). SP maintains a shadow page table in the hypervisor for each process running on the guest. The shadow page table entries store machine addresses. In x86 architecture, when a process is running, the starting address of the corresponding shadow page table is loaded into CR3, a register that points to the guest page table in a native system. Now, a TLB miss will result in a page walk by the MMU on the shadow page table to find the target machine address. Note that the shadow page table has the same depth as the guest page table. Thus, the page walk latency using SP is comparable to the native system. However, any guest page table updates need to be reflected in the shadow page table. The hypervisor needs to monitor guest execution and synchronize the tables, which incurs overhead.

In HAP, the guest page table is extended with a layer of physical-to-machine mappings called extended page tables or nested page tables. HAP is introduced to eliminate the VM exits that result from modifications of the guest page table, CR3 load and TLB flush. Figure 2 shows the HAP page table structure for 64-bit Linux operating systems. In 64-bit systems, the virtual address grows to 48 bits. Both the guest table and the extended table are four levels, compared to two levels for 32-bit systems. The increase in levels greatly magnifies the TLB miss penalty in 64-bit systems. As is shown in Figure 2, walking the extended page table is necessary each time we try to get the

Fig. 3. Execution time of 64-bit SPEC INT. This figure compares the execution times of SPEC INT programs running in SP and HAP, normalized by their execution times in the native OS.

Fig. 4. Execution time of 32-bit SPEC INT. This figure shows that neither SP nor HAP is an absolute winner when 32-bit SPEC INT is running on a VM.

Fig. 5. Execution time of 64-bit Parsec. This figure compares the execution times of 64-bit Parsec applications in SP and in HAP, normalized by their execution times in the native OS.

Fig. 6. Execution time of 32-bit Parsec. This figure shows that HAP performs notably better than SP for vips and x264, while the gaps between SP and HAP are indistinguishable for other programs.

corresponding machine address based on the guest physical addresses stored in CR3 and the guest page table entries. As a result, 24 memory accesses are required to fill a TLB entry for a 64-bit Guest OS, compared to four accesses in a native OS or a VM using SP. This gap between HAP and SP is smaller in 32-bit systems because both guest tables and extended tables are two levels.

2.2. SP vs. HAP in 32-Bit and 64-Bit Systems

As discussed in Section 2.1, HAP suffers greater TLB miss latencies, particularly in 64-bit systems, while SP can incur a performance drop when there is a large number of page faults and page table updates. One would expect that HAP and SP would each demonstrate its own advantage in certain benchmarks. Surprisingly, this expectation is true for 32-bit systems but not for 64-bit systems based on the experimental results shown in Figures 3 through 6. We show normalized execution times with 95% confidence intervals.

SP performs the same as or better than HAP for all 64-bit SPEC CPU and Parsec programs [Bienia et al. 2009], as shown in Figures 3 and 5. *429.mcf* runs over 20% slower in HAP than in SP due to a large amount of operations on pointer structures that lead to TLB misses. *471.omnetpp*, *473.astar*, and *483.xalancbmk* also run faster in SP to different extents. Other 64-bit applications spend almost the same amount of time in SP and in HAP. Based on these results, we conclude that VMs in cloud computing that use 64-bit systems should be launched with SP to guarantee overall execution speed of applications. However, 32-bit legacy code still exists.

Figures 4 and 6 show that neither SP nor HAP is an absolute winner for 32-bit applications running on a 64-bit VM. The advantage of SP is clear for 32-bit *429.mcf*,

but *403.gcc* runs 5% faster in HAP than in SP. The difference for *403.gcc* in a 32-bit VM is much greater (we observe up to 50% slowdown in SP) because of additional mappings in the 32-bit guest kernel. For 32-bit *vips*, HAP outperforms SP by 27.5%. The advantage of HAP for *x264* is also notable, around 5.6%.

Since 64-bit applications perform better using SP and reducing the cost of a page walk for HAP in existing solutions requires a hardware enhancement [Ahn et al. 2012], we are motivated to reduce the overhead of SP for 32-bit systems. In addition, SP as a software-based solution can be implemented and deployed more easily than new HAP techniques.

2.3. A Main Reason for VM Exit

Many studies in the past have focused on the significant overhead of HAP in 64-bit guests and proposed optimizations from both software and hardware perspectives [Bhargava et al. 2008; Barr et al. 2010; Wang et al. 2011; Ahn et al. 2012]. However, our further analysis reveals that HAP performs relatively poorly in 64-bit applications not only because of the larger TLB miss penalty but also because of far fewer VM exits for 64-bit applications using SP compared to the 32-bit version. Figuring out the root cause of the discrepancy between 64-bit and 32-bit applications is necessary to propose a substantial optimization for SP, expecting that SP can be a viable solution for all cases.

After tracing the workloads in a VM using SP, we find that around 11 out of 56 possible reasons for VM exits manifest themselves. The great difference between 32-bit and 64-bit applications is that there are fewer `EXCEPTION_NMIs` (NMIs) in the latter. NMI indicates that software caused an exception or a nonmaskable interrupt occurred. One major type of `EXCEPTION_NMI` results from page faults. When comparing system behaviors of the applications with different amounts of NMIs, we confirm that most of those in the 32-bit applications result from page faults in the heap. Repeated dynamic memory release and reallocation in a 32-bit process is much more notable than in a 64-bit process due to the limited virtual address space of 32-bit systems. We observed that the majority of these operations neither read/write to disk nor load data/text into a specified virtual address. Thus, the page faults are not from limited hardware resources, and it is possible to reduce them via a software solution.

2.4. Traditional Memory Allocator

The default and popular dynamic memory management algorithm for C/C++ is `DLmalloc` by Doug Lea [Lea]. It makes use of two encapsulated system calls (`mmap` and `sbrk`) to set up memory mappings for the user process. Virtual memory obtained via `sbrk` is continuous in address space. Virtual-to-physical memory mappings for it will not be cleared when the user gives up the corresponding memory via `free`, unless it results in the top chunk of the heap becoming sufficiently large (see Section 3.3). On the contrary, `mmap` gets a discrete virtual memory area. It will be released to the OS as soon as `free` is called. Once a user process requests dynamic memory, a memory reservation in the heap will be checked first. If no reserved memory chunk can satisfy the allocation request, one of these two system calls, `sbrk` and `mmap`, will take over and complete the memory allocation.

`Ptmalloc` [Gloger] is an extension of the Lea allocator for multithreaded applications. It introduces the concept of *subarena*. Multiple arenas could take charge of concurrent dynamic memory allocations in parallel programs to support scalability, while in the Lea allocator, there is only one arena, named the *main arena*. The major difference between the main arena and a subarena exists in the structure of the heap. The heap of the main arena is continuous in process address space, while in a subarena it is a list composed of multiple blocks. Each block is named as a *subheap*. If an object allocation

Table I. VM-Exit Frequency of 32-Bit and 64-Bit SPEC INT

32-bit	VM exits/million inst		64-bit	HAP	SP
	HAP	SP			
400.perlbench	0.7	1.0	400.perlbench	0.7	1.2
429.bzip2	0.9	1.6	429.bzip2	0.9	1.6
403.gcc	0.8	12.2	403.gcc	1.0	6.7
429.mcf	2.7	5.4	429.mcf	3.1	8.7
445.gobmk	1.1	1.5	445.gobmk	0.7	1.6
456.hmmmer	1.0	1.1	456.hmmmer	0.9	1.0
458.sjeng	0.7	1.7	458.sjeng	1.0	0.9
462.libquantum	0.6	1.3	462.libquantum	0.7	1.6
464.h264ref	0.7	0.9	464.h264ref	0.5	1.1
471.omnetpp	1.6	2.1	471.omnetpp	1.5	2.7
473.astar	1.2	2.0	473.astar	1.5	1.8
483.xalancbmk	0.8	1.8	483.xalancbmk	1.1	1.4

Note: Table I shows that *403.gcc* exhibits much higher VM-exit frequency in SP.

cannot be satisfied by the current subheaps in the subarenas, a new subheap will be allocated via *mmap* and appended to a subarena based on a round-robin algorithm. The unused portion of the subheap will be reserved for later allocations.

3. ORIGIN OF SHADOW PAGING OVERHEAD

Reducing the frequency of VM exits is considered to be the most important optimization for system virtualization. Although HAP has been introduced to eliminate VM exits caused by page faults, CR3 loads, and TLB invalidations, it also increases the TLB miss penalty. Thus, the experimental results demonstrating that SP outperforms HAP in almost all tested 64-bit applications are contrary to the objectives of hardware designers. In our opinion, tracing the root cause of VM exits for SP would enlighten us with new optimization opportunities for SP so it can match the performance of HAP even in the few pathological cases shown in Section 2.2. This section focuses on an analysis of VM exits for SPEC CPU2006 integer applications to understand the source of VM exits. We will present the results for Parsec and cloud benchmarks in Section 5.

3.1. Frequency of VM Exit

Since the most significant performance bottleneck of SP exists on VM exits, we speculate that the VM-exit frequency might explain the discrepancy between 32-bit and 64-bit applications. In order to measure the VM-exit frequency for 32-bit and 64-bit applications, we count the number of instructions executed in the VM and the number of VM exits at the same time. Table I shows that both 32-bit and 64-bit SPEC INT applications result in higher VM-exit frequency in SP than in HAP. Apparently, HAP does effectively reduce VM exits. However, it can outperform SP only when it reduces the VM-exit frequency significantly. In particular, 32-bit *403.gcc* incurs 15 times more VM exits in SP than in HAP, while the difference between SP and HAP in the 64-bit version is cut in half. Even though the frequency of VM exits for *403.gcc* in SP is still high in the 64-bit version, SP is able to match the performance of HAP.

Since the 32-bit and 64-bit versions of these applications are compiled from the same source code and run on the same operating system, the excessive VM exits in the 32-bit applications might be related to the manner of executing system calls, built-in libraries, or memory management in the kernel. Figuring out the exact cause is a first step to reduce VM exits for SP.

Table II. Ratio of VM Exits Derived from NMIs in SP

32-bit	Ratio	# of NMI	64-bit	Ratio	# of NMI
400.perlbench	45%	1,009,968	400.perlbench	47%	1,322,838
401.bzip2	61%	2,691,901	401.bzip2	60%	2,909,000
403.gcc	92%	14,023,355	403.gcc	84%	6,632,647
429.mcf	64%	1,580,447	429.mcf	54%	1,526,766
445.gobmk	44%	1,187,186	445.gobmk	29%	720,060
456.hmmmer	27%	954,918	456.hmmmer	22%	753,946
458.sjeng	53%	2,194,522	458.sjeng	40%	1,397,581
462.libquantum	59%	3,002,493	462.libquantum	36%	1,206,591
464.h264ref	35%	1,116,780	464.h264ref	46%	1,639,099
471.omnetpp	46%	728,662	471.omnetpp	45%	805,473
473.astar	49%	1,305,406	473.astar	50%	1,231,119
483.xalancbmk	56%	1,263,310	483.xalancbmk	52%	1,037,683

Note: Table II shows that the VM exits due to NMIs dominate most applications when using SP. The most outstanding application is 32-bit *403.gcc*, which performs poorly in SP.

3.2. VM-Exit Distribution

We analyze the distribution of different types of VM exits in SP. For both 32-bit and 64-bit SPEC INT, we observe that up to 92% of VM exits are caused by NMIs, as shown in Table II. Besides, the number of total VM exits in 32-bit *403.gcc* is much larger than that of the 64-bit version. We also list the total number of NMIs in Table II. We find that there are few NMIs in HAP for these applications. This observation serves as evidence to support the conjecture that the NMIs are mostly from page faults in the guest.

In Linux systems, page faults could be divided into two types, minor faults and hard faults. If a page fault does not need to read data from disk, it will be classified as a minor fault; otherwise, it is a hard fault. For HAP, minor faults could be resolved by modifying the corresponding PTEs in the guest page table without resorting to the VMM, but a hard fault is expensive for both SP and HAP because of I/O operations.

Finally, we identify the type of each page fault and the corresponding virtual address that causes it. The conclusion is that, with sufficient physical memory, most page faults are indeed minor faults that do not involve expensive I/Os. Further, we observe that 99% of minor faults in 32-bit applications take place in the heap.

3.3. Dynamic Memory Allocation

The discovery of excessive minor page faults in 32-bit applications drives us to analyze the Lea allocator in the runtime library. Allocating the same amount of memory in 32-bit and 64-bit glibc might be satisfied by different means: unused space on the heap, expanding the heap via *sbrk*, and allocating a new chunk through *mmap*. As we monitor system calls used to set up virtual-to-physical address mappings for the heap, we find that they are invoked more often in 32-bit applications than their 64-bit counterparts. Our further investigation reveals that this phenomenon is related to two thresholds in glibc.

Two thresholds in glibc, *mmap_threshold* and *trim_threshold*, are used to control memory allocation and release, which can make a great difference in the number of page faults within a process. *mmap_threshold* first tells which system call will be triggered to complete a memory allocation when there is no sufficient unused space in the heap. If the size of the requested memory is larger than *mmap_threshold*, *mmap* will be executed; otherwise, the allocator will try to get memory through *sbrk*, which extends the heap. A higher *mmap_threshold* results in more dynamic memory allocations to be satisfied via *sbrk*, so that more memory can be reserved for reuse once freed by a

Fig. 7. Repeatedly allocate, free, and release memory in the heap of the main arena. These bars stand for the states of the heap in the main arena. The user operations from u1 to u4 cause unnecessary memory releases, which lead to minor faults.

process. The area created by *mmap* will be returned to the OS immediately when it is freed. Another important threshold, *trim_threshold*, determines how much memory could be reserved in the top chunk, which stores the freed memory that is obtained most recently via *sbrk*. A higher value of *trim_shreshold* will yield a larger top chunk that can be used for future allocations, reducing the need to destroy/rebuild page table entries in dynamic memory management.

Both thresholds could be assigned a value before the process starts and be automatically adjusted during execution. In the Lea allocator, *mmap_threshold* and *trim_threshold* will be tuned up when a large chunk of memory initially obtained via *mmap* is released and its size exceeds the current *mmap_threshold*. In the meantime, *trim_threshold* is also adjusted to be two times of *mmap_threshold*. Compared to a 64-bit application, a 32-bit process owns limited virtual address space (2^{32} bytes). Thus, the upper bound of *mmap_threshold* is set, by default, to a relatively smaller value in the 32-bit runtime library. Now *mmap* will be invoked much more frequently for memory allocations of 32-bit applications. Those memory chunks will return to the OS as soon as the user process calls *free*. Therefore, less memory can be reserved in the heap and more page faults happen when an allocation needs to rebuild virtual-to-physical address mappings. These faults are the minor faults that can cause excessive VM exits for applications with intensive memory operations.

We see a similar phenomenon for the top chunk in the *sbrk*-managed area. Figure 7 illustrates, in a simplified form, how unnecessary minor faults are caused by a small *trim_threshold*. In this example, the user process allocates a chunk for data *a*, and the heap grows with an invocation of *sbrk(a)* in glibc. The same action is taken when the user allocates another chunk for *b*. Then the user frees *a* and *b* consecutively. As is shown in state P4, the space for chunk *a* is reserved when *a* is freed. After *b* is freed, *a* and *b* are combined into the top chunk, as is shown in state P5. At the time, the size of the top chunk exceeds *trim_threshold*, so *sYSTRim* is triggered to release unused memory in the top chunk to the OS. Then, the state of the heap in the main arena goes back to state P1. If the same allocation pattern (from u1 to u4) happens multiple times in a process, the space for the top chunk in the same area gets released and allocated repeatedly. Obviously, releasing memory that will be reused soon is unnecessary. We observe that this type of memory allocation and release pattern actually occurs in 32-bit *403.gcc*.

Fig. 8. Structure of a subarena. This figure illustrates the organization of a subarena. It is composed of several subheaps. Except for the first subheap, each of the others owns a pointer pointed to the previous subheap.

The same story also exists for parallel programs. A subarena is composed of linked subheaps, as shown in Figure 8. *trim_threshold* is not applied to subheaps in subarenas for measuring whether to return reserved memory to the OS. The whole subheap will be released as long as it is only dominated by the top chunk after the process frees an object in its arena and the available memory in the previous subheap exceeds one page. Then the new top chunk is set in the previous subheap. The examination of releasable subheaps continues along the list until a subheap cannot meet the condition. Particularly, subheap 0 will not be released because it contains the meta-data for the arena.

4. DESIGN AND IMPLEMENTATION

The minor faults caused by the small *mmap* and *trim* thresholds in 32-bit applications can be reduced substantially. As is described by Ezolt [2001], *mmap_max* in glibc can be set to prohibit invoking *mmap*, so that all memory allocations will be satisfied by invoking *sbrk*. Further, *trim_threshold* can be set to prevent the top chunk from releasing. Now, all allocated memory would be kept in the heap for reuse until the process exits. We name this approach *simple SP*.

Simple SP causes a striking effect on VM-exit reduction. It could almost always outperform HAP for both 64-bit and 32-bit sequential applications, but its performance gain is accompanied by excessive physical memory retention. In addition, based on the analysis of Ptmalloc, simple SP has no effects on subheaps in parallel programs. We propose a new approach, adaptive SP, which can achieve the competitive performance of simple SP and work for both sequential and parallel programs without remarkable additional physical memory consumption compared to the original allocator.

Adaptive SP manages to improve the dynamic memory allocation algorithm in the guest library. The key idea is to recycle freed dynamic memory before returning it to the OS. This can decrease the operations of allocating physical pages and filling PTEs, since under most cases they are completed through the previous page faults in the same virtual memory area. However, we should not retain the physical memory for too long as it will increase the process's physical memory footprint and, worse yet, cause more expensive major faults when the physical memory is scarce. A compromise is to keep the memory in process if it is expected to be reused or return it to the OS if it will not be reused in the near future.

Memory released from the heap of a process to the OS is of three types: the top chunk of the main heap, a chunk of deallocated memory initially obtained using *mmap*, and a subheap released from a subarena for a multithreaded process. The first type of memory release can be controlled by adjusting *trim_threshold* based on the extension or shrinkage of the top chunk. If the memory in the top chunk is reallocated since its last release, we can predict that it will be reused later. So we should prevent the

top chunk from being trimmed. For the second type of memory release, since all large chunks of memory obtained through *mmap* are marked with *IS_MMAP*, they can be easily treated differently. We can accomplish reuse of that memory by holding them for a while after the process frees them and check whether they can satisfy the next *mmap* request of a large dynamic memory allocation. *Ptmalloc* introduces the third type of memory that has been described as subheap. Although subheaps are also obtained through *mmap*, they should be treated separately because of their unique characters, such as the uniform size and alignment requirement. Subheaps can also be put on hold when they are freed so they can be reused later.

4.1. Trimming the Top Chunk in the Main Arena

To track the history of the top chunk, two variables, *last_starting_address* and *last_top_chunk_size*, are introduced to record the starting address and the size of the top chunk, respectively. When unused memory in the top chunk exceeds the current *trim_threshold*, we take a sample point and compare the current starting address with *last_starting_address* of the last sample point in order to decide whether it is necessary to adjust *trim_threshold* or not.

If the current starting address is not lower than *last_starting_address* and the sizes of the top chunk in the last two sample points are almost the same, we can conclude that the memory space in the top chunk of the last sample point has been reused. We then predict that the memory in the current top chunk is likely to be reused in the near future. In this case, we tune up *trim_threshold* to prevent repeating operations of memory release and reallocation. On the other hand, if the current starting address is lower than the last sample, it suggests that the memory under the top chunk at the last sample point has been freed and combined into the top chunk before the current sample point. If such a pattern repeats, we may predict that the demand for dynamic memory in the process is diminishing, so we can release the unused memory in the top chunk and tune *trim_threshold* down to allow more aggressive releases of memory in the future.

To prevent the process from keeping too much free memory in the top chunk for a long period due to high *trim_threshold*, we introduce a timeout mechanism. We reset the timer each time *trim_threshold* is tuned up. The timer ticks each time a memory chunk in the heap is freed. When the timer expires, the value of *trim_threshold* will be reset to the value of two times of *mmap_threshold*.

4.2. Delayed unmap

Large dynamic memory chunks allocated via *mmap* are marked with *IS_MMAP*; in parallel applications, subheaps are also obtained via *mmap*. We propose a mechanism to delay the operation of *unmap* in order to make full use of the mapped virtual space.

A doubly linked list is introduced to store the delayed *unmap* chunks. When a memory chunk with *IS_MMAP* is freed, we do not unmap it. Instead, we add it to the list, sorted in descending order of chunk size. If the length of the list exceeds the max length, the memory chunk in the tail of the list will be returned to the OS. We always resort to the list for the next *mmap* request to maximize space reuse before actually invoking *mmap*. To avoid excessive physical memory retention, we release all but the first node in the list when the list fails to satisfy an *mmap* request. Querying the list is both first fit and largest fit as the list is sorted. This design is aimed at increasing the probability of successful reuse while avoiding retention of a large memory chunk for too long. We make the maximum length of the list adjustable, since different applications might have different memory demands. Experimental evaluation shows that maintaining two delayed *unmap* memory chunks performs the best and holding more chunks in the list does not lead to any notable performance gain.

For subheaps, we keep a separate doubly linked list for each subarena and a global list for the process. Memory chunks in these lists of subarenas do not need to be arranged in order when inserted, because all subheaps are of the same size. When a subheap is trimmed, it will be added to the head of the list for its subarena. The tail node will be released if the length surpasses the maximum limit. When a new subheap is requested in a subarena, the subheap in the head of the list, if it exists, for that arena will be reused. If there is no node available, our algorithm will turn to the global list, which collects the subheaps deleted from the lists of the subarenas. In other words, a released subheap will be held in the delay list of the subarena first and then in the global list before it is returned to the system. The subarena *delay* lists, as well as the global *subheap* list, use a lock to prevent contention among threads.

In most cases, a delayed *unmap* chunk will not impose an additional physical memory burden on the system, because it will be reused soon if the chunk size is large enough and released soon if it is too small as the list will soon exceed the maximum length. However, if the sizes of memory allocations exceed *mmap_threshold* and keep increasing, the head nodes of the delayed list will never be released. The resident set size of the process will be bigger in our approach than the original allocator. Fortunately, in practice, such a case is rare. Occasional increases are usually not substantial, as we control the sizes of the delay lists.

5. EVALUATION

This section evaluates our approach on a series of sequential and parallel workloads as well as two cloud computing applications. Our experiments on the cloud computing benchmarks and overcommitted VMs are performed on an Intel CORE I5 machine with 8GB of physical memory and four 2.80GHz cores. The remaining experiments are run on an Intel CORE I7 machine with 12GB of physical memory and four 2.80GHz cores with support of HyperThread (HT). All of our experiments are carried out on a hypervisor based on Xen-4.1.2, except that the overcommitted VMs are based on Xen-3.3.1. The guest system is built on CentOS5.9. The Linux kernel version is 2.6.32 and the GNU C library version is 2.9.

5.1. Benchmarks

SPEC CPU2006 [Henning et al. 2006] is chosen to represent the sequential workload. It contains integer and floating-point programs and benchmarks CPUs, memory systems, and C, C++, and FORTRAN compilers. Our work focuses on the Lea allocator that manages memory for C/C++, so we benchmark the whole integer set and part of the floating-point set in C/C++ (433.milc, 444.namd, 447.dealII, 450.soplex, 453.povray, 470.lbm, and 482.sphinx3). Parsec 2.1 [Bienia et al. 2009] is selected to measure our optimization for parallel programs. They are all implemented using C/C++. The applications in Parsec include floating-point operations, reads, writes, and synchronization primitives. In addition, they contain complex sharing patterns and interthread communications. Our evaluations on cloud applications are based on the Sector-Sphere platform [Gu et al. 2009], with the datasets from [Chen et al. 2012]. The two map-reduce applications are *knn* [Gillick et al. 2006] and *ann* [Liu et al. 2010]. *knn* is a map-reduce implementation of the k-nearest neighbor algorithm, and *ann* is a back-propagation algorithm for training the weights of a neural network. The size of the dataset for *knn* is 880M (10M records) and over 6G (8M records) for *ann*.

5.2. Experiment Setup

For SPEC CPU2006, we create a 64-bit guest VM with only one vcpu, and also appoint one for dom0. By enabling one core with HT in BIOS, we have two logical cores available. Each of them is attached to dom0 and the guest, respectively. In addition, we assign

Table III. VM Exits per Million Instructions of 32-Bit Applications

SPEC INT	VM exits/million inst			SPECFP	Original SP	Adaptive SP	Simple SP
	Original SP	Adaptive SP	Simple SP				
400.perlbench	1.00	1.00	1.00	433.milc	43.11	10.66	1.59
401.bzip2	1.64	1.61	1.53	444.namd	0.66	0.67	0.71
403.gcc	12.22	5.38	3.67	447.dealII	2.63	2.36	1.80
429.mcf	5.43	4.54	4.06	450.soplex	4.27	3.22	2.61
445.gobmk	1.51	1.12	1.14	453.povray	1.47	1.20	1.10
456.hmmer	1.10	1.06	1.15	470.lbm	1.72	1.67	1.65
458.sjeng	1.65	1.43	1.46	482.sphinx3	0.82	0.79	0.78
462.libquantum	1.33	0.97	0.69				
464.h264ref	0.86	0.77	0.76				
471.omnetpp	2.05	1.98	2.13				
473.astar	2.71	2.62	2.54				
483.xalancbmk	1.77	1.78	1.64				

Note: *443.milc* and *403.gcc* show remarkable reduction of VM-exit frequency with adaptive SP. Other applications show little variation of VM-exit frequency in different modes.

4GB of physical memory for the guest. For a fair comparison, the native OS is started with 4GB of physical memory and one core with HT disabled.

For the parallel programs in Parsec 2.1, we enable all cores and HT to offer as many as eight threads. The 64-bit guest VM is appointed exactly eight vcpus. Physical memory assigned to each guest is 6GB. We carry out all experiments multiple times to ease the effect of nondeterminism in parallel programs. To forbid automatic CPU frequency fluctuation caused by thermal management, we fix CPU frequency at 1.6GHz.

Our cloud computing environment is deployed in one machine, where the security server and master node reside in Domain 0. We create a Domain U guest with two cores and 4GB of memory to act both as a slave node and as a client end to prevent the overhead of network file transfer from disturbing our evaluation. The guest OS of the slave node is 32 bit due to compatibility of reading/writing 32-bit and 64-bit files in Sector-Sphere.

5.3. Performance Evaluation

This section evaluates the performance of adaptive SP by comparing it with HAP, original SP, and simple SP, in terms of the effect on VM-exit frequency and execution time. We present the results for SPEC CPU, Parsec, and cloud applications separately.

Table III shows the effect of our approach on VM exit for 32-bit SPEC. Three applications show minor increases of VM exit with adaptive SP due to measurement bias [Mytkowicz et al. 2009]. Note that the frequencies of VM exit of *403.gcc* and *433.milc* have decreased to a great extent in both simple SP and adaptive SP. *403.gcc* and *433.milc* are the only two benchmarks where original SP performs remarkably poorer than HAP. The reduction in the number of VM exits leads to a significant performance improvement, as shown in Figure 9. We show the normalized execution times of SPEC with the 95% confidence interval on the top of each bar in Figure 9. The confidence intervals of adaptive SP and simple SP are overlapping for most benchmarks except *471.omnetpp* and *433.milc*. Adaptive SP is about 5% better than simple SP, on average, for *471.omnetpp*, while simple SP shows an edge of 2.5% for *400.perlbench*. The execution times of *433.milc* and *403.gcc* are improved by 30% and 5%, respectively, by adaptive SP compared to original SP. With the improvements, the performance of these two benchmarks in adaptive SP is comparable with that of HAP. When comparing the average time of multiple runs of each application, we observe that for 13 out of the 19

Fig. 9. Execution time of SPEC CPU with HAP, original SP, adaptive SP, and simple SP. If an application runs faster in original SP than in HAP, adaptive SP shows similar performance to original SP; otherwise, adaptive SP could eliminate performance degradation of original SP and achieve performance competitive to HAP. The difference between simple SP and adaptive SP is not significant.

Table IV. Page Faults of 32-Bit Parsec 2.1

Parsec 2.1 (32-bit)	# of minor page faults in guest		
	Original SP	Adaptive SP	Simple SP
blackscholes	156,628	156,618	156,638
Bodytrack	126,045	46,793	47,253
Facesim	85,165	80,044	70,013
Ferret	633,607	562,603	595,830
Freqmine	455,101	206,717	165,795
Swaptions	1,086	1,086	1,100
fluidanimate	48,882	48,822	48,800
Vips	488,167	436,227	479,905
x264	74,022	73,951	73,952

Note: Table IV shows that the adaptive memory allocator in the guest can reduce minor page faults in *bodytrack*, *ferret*, *freqmine*, and *vips*.

Table V. VM Exits per Million Instructions of 32-Bit Parsec

Parsec 2.1 (32-bit)	VM exits/million inst			
	HAP	Original SP	Adaptive SP	Simple SP
blackscholes	9.9	16.9	14.9	14.7
Bodytrack	11.4	14.1	10.6	11.6
Facesim	3.0	3.5	3.5	3.4
Ferret	2.4	7.5	7.3	7.3
Freqmine	8.1	10.3	9.3	9.1
Swaptions	2.1	2.3	2.3	2.3
fluidanimate	1.8	2.2	2.2	2.2
Vips	16.3	30.5	25.2	26.2
x264	19.7	31.5	29.0	29.2

Note: Table V shows that the VM-exit frequency of Parsec is relatively higher than SPEC. The frequencies of VM exit in *bodytrack* and *vips* have been substantively decreased with our approach.

SPEC applications, adaptive SP outperforms HAP. For four of the remaining six applications, they are the same. For *milc* and *namd* only, HAP shows an edge of merely 1%.

The results for Parsec 2.1 are shown in Table IV, Table V, and Figure 10. We notice that the experimental results vary in a relatively large range as a result of nondeterministic thread scheduling. We run each benchmark 20 times. The data presented in

Fig. 10. Execution time of Parsec with HAP, original SP, adaptive SP, and simple SP. Adaptive SP can outperform simple SP as for *bodytrack*. For *vips*, adaptive SP improves over original SP and matches the performance of HAP.

Table VI. VM_EXIT Derived from NMIs

		HAP	Original SP	Adaptive SP	Simple SP
Knn	NMIs	952	1,799,967	1,802,243	729,140
	VM_EXITS	1,008,531	4,635,176	4,621,628	2,900,690
Ann	NMIs	20,333	10,062,818	3,263,436	2,001,945
	VM_EXITS	4,341,304	20,812,226	10,992,439	9,822,777

Note: The number of VM exits in original SP is cut in half by adaptive SP for *ann*.

Tables IV and V are averages with the outliers removed. The result for *raytrace* is not selected because it is unstable on our machine, even with the native OS.

Table IV lists the total number of minor page faults in the guest for each workload in Parsec. The most notable minor fault reduction by adaptive SP is shown in *bodytrack*, *freqmine*, *vips*, and *Ferret*, with 63% for *bodytrack*, 55% for *freqmine*, and 11% for *vips* and *Ferret*. However, due to the difference in total executed instructions, the minor fault reduction in these four programs only translates to significant reduction in VM exits for *bodytrack* and *vips*, as shown in Table V. Complex sharing patterns and interthread communications exist in Parsec 2.1. VM exits due to EXCEPT_NMI is only around 20%, on average, of all VM exits, which is significantly lower than SPEC. Except for the exits due to minor page faults, the other exits are caused by other exceptions or nonremarkable interruptions in the guest because they cannot be eliminated by HAP either.

As shown in Table V, adaptive SP shows a lower VM-exit frequency than simple SP for *bodytrack*, *vips*, and *x264*. We monitor the arenas of all Parsec applications and find that only *bodytrack*, *facesim*, *ferret*, *vips*, and *x264* have ever created subarenas. The lower VM-exit frequencies in *bodytrack*, *vips*, and *x264* are because adaptive SP can benefit from caching subheaps in the subarenas.

The VM-exit reduction by adaptive SP in *bodytrack* and *vips* leads to a 38% and 25% improvement, respectively, compared to original SP. The improvement fills the only gap between HAP and SP for all Parsec applications. Adaptive SP beats simple SP by 25% for *bodytrack* because of the minor fault reduction in subarenas.

Tables VI and VII show the results for the two cloud computing applications. Adaptive SP is able to reduce the number of VM exits by nearly 47% for *ann* and is much closer to simple SP than original SP. This reduction leads to a 2% improvement in total

Table VII. Execution Time of HAP, Original SP, Adaptive SP, and Simple SP

		HAP	Original SP	Adaptive SP	Simple SP
Knn	Read data time	58	55	45	43
	Process time	150	150	149	150
	Total time	209	207	197	194
Ann	Read data time	415	433	419	421
	Process time	208	210	210	208
	Total time	625	645	632	631

Note: The total time of these two applications running on Sector-Sphere includes reading the data stream from the network or local slave nodes, map-reduce computation, and delivery time.

Fig. 11. Resident set size of 403.gcc in original SP, adaptive SP, and simple SP. This figure contains bube inputs provided in the reference run of 403.gcc. As shown in the picture, original SP consumes the least memory. For input of s04.i, simple SP needs 40K more physical memory pages than original SP and adaptive SP.

time compared to original SP and brings it within 1% of HAP, compared to the 3% gap between original SP and HAP. Although adaptive SP does not reduce the total number of VM exits in SP for *knn*, it improves over HAP and original SP by 5% in total execution time. We suspect the improvement comes from a better distribution of VM exits so some overhead is hidden.

5.4. Memory Consumption

As discussed in Section 4, both adaptive SP and simple SP can cause an increase in physical memory consumption. To evaluate the impact, we take *403.gcc* and *bodytrack* as representatives as adaptive SP is most effective on these two benchmarks. We collect the resident set size for every 5 seconds and show them in Figure 11 and Figure 12. Since *403.gcc* has nine separate input files in the reference set, we label them separately in Figure 11. As shown in Figure 11, the physical memory consumption of adaptive SP and simple SP is generally close to each other, but more than original SP.

An interesting observation from Figure 11 is that adaptive SP does not increase the peak memory demand of original SP, although it can stay in a high demand for a longer period of time. On the other hand, simple SP can increase the peak demand dramatically. For example, simple SP's resident set size is 40K pages, or 20% more than that of adaptive SP and original SP for *s04.i*. This increase can cause more expensive hard page faults when the machine memory is tight. Adaptive SP's lingering effect on the resident set stems from the nature of our approach. We delay the releases of unmap chunks and reserve them in the delayed unmap lists. If the nodes in the list are either reused or dropped soon, the lingering effects will be short.

Figure 12 shows the trend of resident set size of *bodytrack*. Adaptive SP is able to maintain the same resident size as original SP, with consistently less memory

Fig. 12. Resident set size of *bodytrack* in original SP, adaptive SP, and simple SP. This figure illustrates the resident set size for *bodytrack*. Simple SP will release subheaps from time to time, which is reflected in the decline of the resident set size. Adaptive SP consumes nearly the same amount of physical memory as original SP with less fluctuation.

Fig. 13. Overcommitted VM for 403.gcc. This figure compares the execution times of 403.gcc under overcommitted VMs.

Fig. 14. Overcommitted VM for 429.mcf. This figure compares the execution times of 429.mcf under overcommitted VMs.

consumption than simple SP. This result shows the overall advantage of our solution on balancing execution time and the physical memory footprint.

5.5. Overcommitted VMs

In a data center, a VM can be overcommitted in its resource allocation. Specifically for the study in this article, memory overcommitment can affect the performance of the memory virtualization schemes. To evaluate how the proposed adaptive shadow paging performs in an overcommitted VM, we run 403.gcc and 429.mcf, two representative benchmarks where SP and HAP each outperforms the other, in a set of VMs with decreasing allocation of the physical memory. Note that we use Xen-3.3.1, not Xen-4.1.2, in this set of experiments, so the VM-exit and page fault statistics are different from other sections.

Based on the Work Set Sizes (WSSs) of 403.gcc and 429.mcf, we reduce physical memory size until it is below the working set size and causes a significant performance drop due to major page faults.

Figures 13 and 14 show the execution times of 403.gcc and 429.mcf with respect to different VM physical memory sizes. As shown in Figure 13, for 403.gcc, adaptive SP is able to outperform original SP and match the performance of HAP until the memory size is dropped to 500MB. At this memory size, all three schemes observe around a 50% slowdown. Although HAP runs faster than adaptive SP, the matching performance of original SP and HAP confirms that the advantage of HAP over adaptive SP in an overcommitted system is not because of HAP's ability to handle VM exits. Rather, it is because of the increased major page faults by adaptive SP, which only occur when

Table VIII. 403.gcc Major Faults and VM-Exits

mem size	Original SP		Adaptive SP		HAP	
	# VM exits	# major flts	# VM exits	# major flts	# VM exits	# major flts
800MB	15,979,476	3,992	8,473,407	3,927	3,025,354	3,597
700MB	16,867,409	10,985	17,336,038	13,941	3,644,569	9,195
600MB	20,868,301	28,659	20,833,949	35,023	5,339,379	30,645
500MB	25,038,587	64,558	31,587,230	97,454	9,894,046	77,935

Table IX. 429.mcf Major Faults and VM-Exits

mem size	Original SP		Adaptive SP		HAP	
	# VM exits	# major flts	# VM exits	# major flts	# VM exits	# major flts
900MB	3,373,215	146	3,317,728	144	2,495,013	130
890MB	3,420,816	384	3,392,769	344	2,518,648	166
880MB	3,432,540	362	3,419,383	388	2,555,990	312
870MB	201,691,457	1,308,354	204,032,253	1,281,762	55,292,642	755,033

the memory is too tight and the Quality-of-Service (QoS) becomes unacceptable for all schemes. Table VIII shows the number of major faults and VM exits in different modes for 403.gcc. The number of major faults for adaptive SP increases more quickly than that of HAP and original SP when the physical memory is below the WSS. This also leads to increases in VM exits. As discussed in Section 5.4, although adaptive SP does not increase the WSS of a benchmark, it can cause the memory demand to linger at a peak point for a longer period of time. When the memory is overcommitted, this effect can result in more major faults. This phenomenon is more evident for 403.gcc. However, based on the execution times shown in Figures 13 and 14, we can conclude that it is the overhead of major faults that takes over as a major impact on performance when the memory is overcommitted.

For 429.mcf, when the memory size decreases from 880MB to 870MB, HAP, original SP, and adaptive SP all observe over 3.5 times of performance degradation. Adaptive SP and original SP both perform better than HAP when the memory is sufficient. However, when the memory size goes down to 880MB, the gap between original SP and HAP becomes slightly smaller. A further decrease of memory to 870MB causes 429.mcf to run 1,420 seconds in adaptive SP, 1,450 seconds in original SP, and 1,380 seconds in HAP. HAP indeed gains a slight edge over both adaptive SP and original SP. We attribute the advantage of HAP in the overcommitted system to a much higher number of major page faults for both original SP and adaptive SP that leads to a significant increase of VM exits as shown in Table IX.

For both 403.gcc and 429.mcf, adaptive SP is able to maintain its advantage even when the memory is slightly overcommitted and thus the QoS is still acceptable. It is interesting to see that original SP performs as well as HAP for 403.gcc when the memory is overcommitted. This result is consistent with our conclusion that a software-based memory virtualization scheme can match or outperform a hardware-based scheme in all circumstances. The result also suggests that there exists an improvement space for adaptive SP in an overcommitted system. At a minimum, we can monitor the major faults and turn off adaptive SP when we detect the memory is overcommitted.

5.6. Sensitivity Study

A high *trim_threshold* can cause an increase of resident set size. In order to prevent maintaining a high *trim_threshold* for too long, we tune *trim_threshold* down under two conditions in our implementation: there are successive decreases of the starting address of the top chunk, and the timer for the high *trim_threshold* has expired. The

experiments in this article tune down *trim_threshold* after two successive starting address decreases. Other thresholds will lead to performance degradation. The timer we use is set when tuning up *trim_threshold*. It will be counted down each time the user process calls *free*. We vary the initial timer from 5 to 15 for *403.gcc* and see limited change in its execution time. However, setting the timer too big or too small yields runtime increases for some applications.

In addition, we also constrain the length of doubly linked lists for delayed unmap chunks to prevent massive memory retention. In our experiments, we set the maximum length of the list for large memory chunks marked as *IS_MMAP* to 2; the max length of the delay list for each subarena is 1; and the max length of the global list is 1. Because of the relative small working set in the benchmarks, these thresholds can affect the resident set sizes but show little effect on the actual performance.

6. RELATED WORK

Since memory is one of the most frequently accessed devices on computers, memory virtualization overhead is critical for the overall performance of virtual machines. Adams and Agesen [2006] compare software-based and hardware-assisted hypervisor designs. They show that each approach has its own advantage. Wang et al. [2011] propose a selective memory virtualization approach that dynamically selects shadow paging and hardware-assisted paging based on VM application behavior. One disadvantage of hardware-assisted paging is the page walk latency to resolve a TLB miss. Bhargava et al. [2008] systematically discuss the design of hardware-assisted paging. They introduce page walk caches used in native OSs to virtualization systems, because spatial and temporal locality also exist in hardware-assisted page tables. To evaluate the effect of page walk caching, Barr et al. [2011] compare five designs of caching intermediate page table entries in multilevel page tables. Ahn et al. [2012] propose a flat nested page table to reduce unnecessary memory references for nested walks when a TLB miss occurs and a speculative inverted shadow paging supported by the flat nested page table.

Ezolt [2001] is the first to figure out the relationship between minor page faults and memory allocator thresholds. He proposes to reduce minor faults by disabling thresholds. He suggests that minor faults can be significantly reduced by forcing the dynamic memory allocator to use *sbrk* only and preventing the top chunk in the heap from releasing. Our evaluation shows that Ezolt's approach can be too aggressive and can dramatically increase the physical memory demand of an application. Our work on investigating the root cause of VM exits in VM with SP was initially inspired by Ezolt's work. Berger et al. [2002] compare custom memory allocators and a general-purpose allocator and encourage people to use the general-purpose Lea allocator instead of customized allocators. They also propose an allocator using a combination of regions and heaps, which outperforms other allocators with region-like semantics. For parallel applications, Berger et al. [2000] design the Hoard allocator that aims at improving speed and scalability while avoiding false sharing and memory blowup. Ferreira et al. [2009] present a survey on multiple open-source memory allocators for parallel applications. For parallel applications, we focus on improving the popular Ptmalloc. We leave the study on the effect of other allocators as our future work.

7. CONCLUSION

The performance of SP depends primarily on the frequency of VM exits. This article investigates the root cause of excessive VM exits in VM with SP and attributes it to the dynamic memory allocation algorithm implemented in glibc. We propose enhancements to the popular Lea allocator and Ptmalloc and make SP competitive with HAP in all the applications we run. In addition, SP often outperforms HAP significantly.

The disadvantage of HAP is mostly due to the long TLB miss latency. However, reducing TLB misses and latencies is challenging and might require new architectural advancements. Given the flexibility of our software-based solution and its striking performance, we suggest that SP should take precedence for memory virtualization in cloud computing where QoS and application performance are critical. In the future, we will experiment on other popular memory allocators. Furthermore, this article focuses on C/C++ applications. It is also interesting to investigate applications that rely on automatic memory management.

ACKNOWLEDGMENTS

This work is supported by Tencent Inc., the National Natural Science Foundation of China (Grant No. 61232008, 61170055, 61272158, 61328201), the National High Technology Research 863 Program of China (2012AA010905), the Research Fund for the Doctoral Program of Higher Education of China (20110001110101), and the Special Foundation of Industry Development for Biology, Internet, New Energy and New Material of Shenzhen (JC201104210107A). Zhenlin Wang is also supported by National Science Foundation Career CCF0643664.

We would also like to thank the State Key Laboratory of Computer Architecture, Institute of Computing Technology, China Academy of Sciences, for providing the Sector-Sphere data generator. Zhigang Wang, Wei Kuang, and Liang Yan all made efforts to facilitate experiment setups. Finally, we thank Linda Ott and all the anonymous reviewers who provided many insightful comments.

REFERENCES

- ADAMS, K. AND AGESEN, O. 2006. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGOPS Operating Systems Review* 40, 5, 2–13.
- AHN, J., JIN, S., AND HUH, J. 2012. Revisiting hardware-assisted page walks for virtualized systems. In *Proceedings of the 2012 39th Annual International Symposium on Computer Architecture (ISCA'12)*. IEEE, 476–487.
- BARR, T. W., COX, A. L., AND RIXNER, S. 2010. Translation caching: skip, don't walk (the page table). *ACM SIGARCH Computer Architecture News* 38, 3, 48–59.
- BARR, T. W., COX, A. L., AND RIXNER, S. 2011. SpecTLB: A mechanism for speculative address translation. In *Proceedings of the 2011 38th Annual International Symposium on Computer Architecture (ISCA'11)*. IEEE, 307–317.
- BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. 2002. Reconsidering custom memory allocation. *ACM SIGPLAN Notices* 37, 11, 1–12.
- BERGER, E. D., MCKINLEY, K. S., BLUMOFF, R. D., AND WILSON, P. R. 2000. Hoard: A scalable memory allocator for multithreaded applications. *ACM SIGPLAN Notices* 35, 11, 117–128.
- BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. 2008. Accelerating two-dimensional page walks for virtualized systems. *ACM SIGARCH Computer Architecture News* 36, 1, 26–35.
- BIENIA, C. AND LI, K. 2009. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*.
- CHEN, Y., FANG, S., ECKHOUT, L., TEMAM, O., AND WU, C. 2012. Iterative optimization for the data center. *ACM SIGARCH Computer Architecture News* 40, 1, 49–60.
- EZOLT, P. 2001. A study in Malloc: a case of excessive minor faults. In *Proceedings of the 5th Annual Linux Showcase and Conference*.
- FERREIRA, T. B., MATIAS, R., MACEDO, A., AND ARAUJO, L. B. 2011. An experimental study on memory allocators in multicore and multithreaded applications. In *Proceedings of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'11)*. IEEE, 92–98.
- GILLESPIE, M. 2009. Best Practices for Paravirtualization Enhancements from Intel® virtualization technology: EPT and VT-d. Retrieved November 26, 2013 from <http://software.intel.com/en-us/articles/best-practices-for-paravirtualization-enhancements-from-intel-virtualization-technology-ept-and-vt-d>.
- GILICK, D., FARIA, A., AND DeNERO, J. 2006. MapReduce: Distributed computing for machine learning. Retrieved November 26, 2013 from http://www.icsi.berkeley.edu/~arlo/publications/gillick_cs262a_proj.pdf.
- GLOGER, W. Wolfram Gloger's Malloc Homepage. Retrieved November 26, 2013 from <http://www.malloc.de/en/>.

- GU, Y. AND GROSSMAN, R. L. 2009. Sector and Sphere: The design and implementation of a high-performance data cloud. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 367, 1897, 2429–2445.
- HENNING, J. L. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4, 1–17.
- LEA, D. A Memory Allocator. Retrieved November 26, 2013 from <http://g.oswego.edu/dl/html/malloc.html>.
- LIU, Z., LI, H., AND MIAO, G. 2010. MapReduce-based backpropagation neural network over large scale mobile data. In *Proceedings of the 2010 6th International Conference on Natural Computation (ICNC'10)*. IEEE, 1726–1730.
- MENON, A., COX, A. L., AND ZWAENEPOEL, W. 2006. Optimizing network virtualization in Xen. In *Proceedings of the USENIX Annual Technical Conference*. 15–28.
- MYTKOWICZ, T., DIWAN, A., HAUSWIRTH, M., AND SWEENEY, P. F. 2009. Producing wrong data without doing anything obviously wrong! *ACM SIGPLAN Notices* 44, 3, 265–276.
- SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G., AND PRATT, I. 2008. Bridging the gap between software and hardware techniques for i/o virtualization. In *Proceedings of the USENIX'08 Annual Technical Conference*.
- TALLURI, M. AND HILL, M. D. 1994. Surpassing the TLB performance of superpages with less operating system support. 29, 11, 171–182.
- TALLURI, M., KONG, S., HILL, M. D., AND PATTERSON, D. A. 1992. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*. 415–424.
- WANG, X., ZANG, J., WANG, Z., LUO, Y., AND LI, X. 2011. Selective hardware/software memory virtualization. *ACM SIGPLAN Notices* 46, 7, 217–226.
- ZHAO, W., WANG, Z., AND LUO, Y. 2009. Dynamic memory balancing for virtual machines. *ACM SIGOPS Operating Systems Review* 43, 3, 37–47.

Received June 2013; revised November 2013; accepted November 2013