Understanding Memory Resource Management in VMware vSphere® 5.0

Performance Study

TECHNICAL WHITE PAPER

vmware®

Table of Contents

Overview

VMware ESXi™, a crucial component of VMware vSphere 5.0, is a hypervisor designed to efficiently manage hardware resources including CPU, memory, storage, and network among multiple, concurrent virtual machines. This paper describes the basic memory management concepts in ESXi, the configuration options available, and provides results to show the performance impact of these options. The focus of this paper is in presenting the fundamental concepts of these options. *

NOTE: "Swap to SSD Performance" on page 26 is new for vSphere 5.0.

Introduction

ESXi uses high-level resource management policies to compute a target memory allocation for each virtual machine (VM) based on the current system load and parameter settings for the virtual machine (shares, reservation, and limit[2]). The computed target allocation is used to guide the dynamic adjustment of the memory allocation for each virtual machine. In the cases where host memory is overcommitted, the target allocations are still achieved by invoking several lower-level mechanisms to reclaim memory from virtual machines.

This paper assumes a pure virtualization environment in which the guest operating system running inside the virtual machine is not modified to facilitate virtualization (often referred to as paravirtualization). Knowledge of ESXi architecture will help you understand the concepts presented in this paper.

The VMware vSphere Client exposes several memory statistics in the performance charts. Among them are charts for the following memory types: consumed, active, shared, granted, overhead, balloon, swapped, and compressed. A complete discussion about these metrics can be found in *Memory Performance Chart Metrics in the vSphere Client*[3] and *VirtualCenter Memory Statistics Definitions*[4].

Two important memory statistics are Consumed Memory and Active Memory. You can use the charts for these statistics to quickly monitor the host memory and virtual machine memory usage.

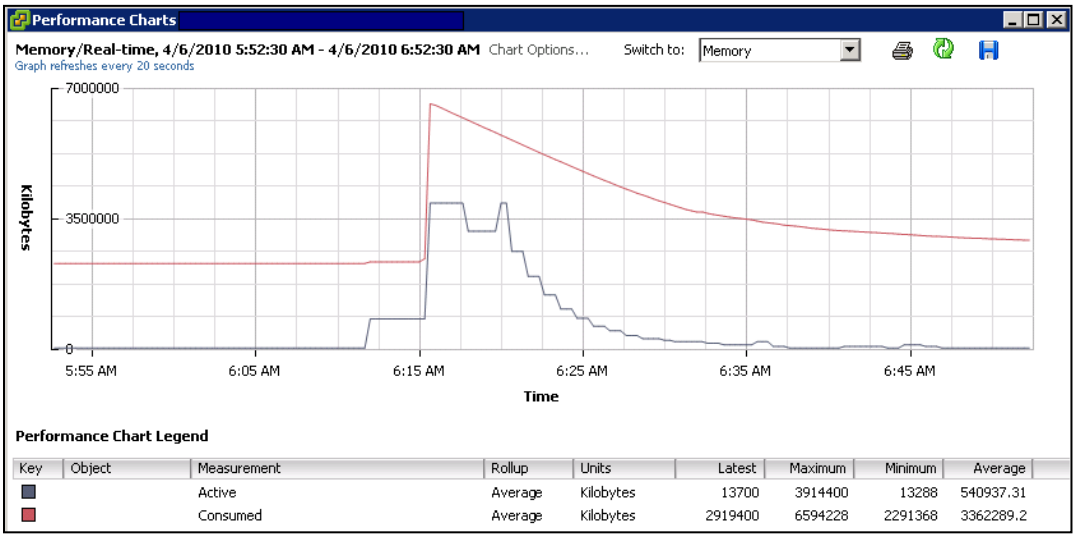

Figure 1. Host and Active Memory Usage in vSphere Client Performance Charts

Consumed Memory usage is defined as the amount of host memory that is allocated to the virtual machine, Active Memory is defined as the amount of guest memory that is currently being used by the guest operating

* More details can be found in *Memory Resource Management in VMware ESX Server*[1]

system and its applications. Use these two statistics to analyze the memory status of the virtual machine and look for hints to address potential performance issues.

This paper helps answer these questions:

- Why is the Consumed Memory so high?
- Why is the Consumed Memory usage sometimes much larger than the Active Memory?
- Why is the Active Memory different from what is seen inside the guest operating system?

These questions cannot be easily answered without understanding the basic memory management concepts in ESXi. Understanding how ESXi manages memory will also make the performance implications of changing ESXi memory management parameters clearer.

ESXi Memory Management Overview

Terminology

The following terminology is used throughout this paper.

- *Host physical memory*[†] refers to the memory that is visible to the hypervisor as available on the system.
- *Guest physical memory* refers to the memory that is visible to the guest operating system running in the virtual machine.
- *Guest virtual memory* refers to a continuous virtual address space presented by the guest operating system to applications. It is the memory that is visible to the applications running inside the virtual machine.
- *Guest physical memory* is backed by host physical memory, which means the hypervisor provides a mapping from the guest to the host memory.
- The memory transfer between the guest physical memory and the guest swap device is referred to as *guest-level paging* and is driven by the guest operating system. The memory transfer between guest physical memory and the host swap device is referred to as *hypervisor swapping*, which is driven by the hypervisor.

Memory Virtualization Basics

Virtual memory is a well-known technique used in most general-purpose operating systems, and almost all modern processors have hardware to support it. Virtual memory creates a uniform virtual address space for applications and allows the operating system and hardware to handle the address translation between the virtual address space and the physical address space. This technique not only simplifies the programmer's work, but also adapts the execution environment to support large address spaces, process protection, file mapping, and swapping in modern computer systems.

The ESXi hypervisor creates a contiguous addressable memory space for a virtual machine when it runs. This memory space has the same properties as the virtual address space that the guest operating system presents to the applications running on it. This allows the hypervisor to run multiple virtual machines simultaneously while protecting the memory of each virtual machine from being accessed by others. Therefore, from the view of the application running inside the virtual machine, the ESXi hypervisor adds an extra level of address translation that maps the guest physical address to the host physical address. As a result, there are three virtual memory layers in ESXi: guest virtual memory, guest physical memory, and host physical memory. Their relationships are illustrated in Figure 2(a).

[†] The terms host physical memory and host memory are used interchangeably in this paper. They are also equivalent to the term machine memory used in *Memory Resource Management in VMware ESX Server*[1].

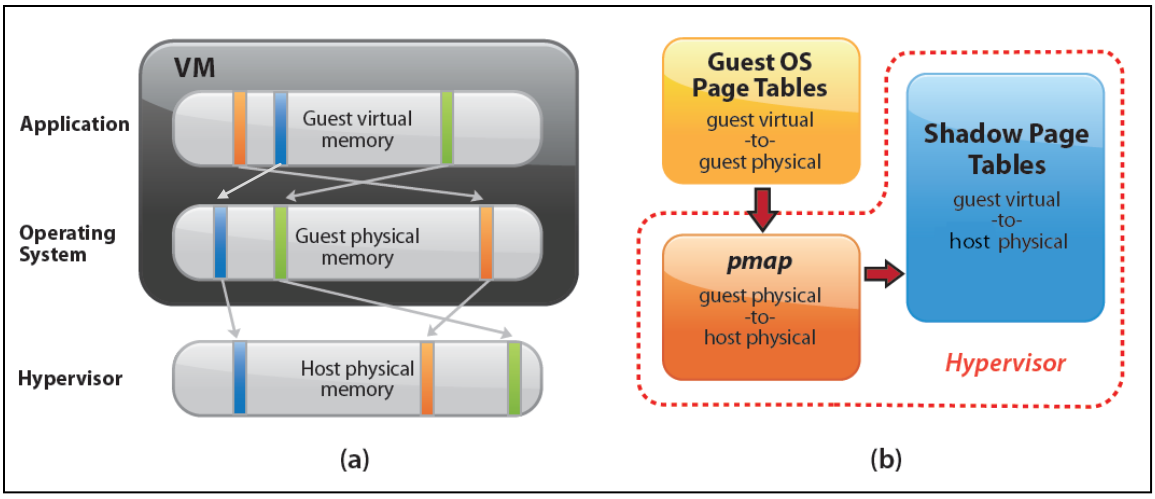

Figure 2. Virtual Memory Levels (a) and Memory Address Translation (b) in ESXi

As shown in Figure 2(b), in ESXi, the address translation between guest physical memory and host physical memory is maintained by the hypervisor using a physical memory mapping data structure, that is, *pmap*, for each virtual machine. The hypervisor intercepts all virtual machine instructions that manipulate the hardware translation lookaside buffer (TLB) contents or guest operating system page tables, which contain the virtual to physical address mapping. The actual hardware TLB state is updated based on the separate shadow page tables, which contain the guest virtual to host physical address mapping. The shadow page tables maintain consistency with the guest virtual to guest physical address mapping in the guest page tables and the guest physical to host physical address mapping in the *pmap* data structure. This approach removes the virtualization overhead for the virtual machine's normal memory accesses because the hardware TLB will cache the direct guest virtual to host physical memory address translations read from the shadow page tables. Note that the extra level of guest physical to host physical memory indirection is extremely powerful in the virtualization environment. For example, ESXi can easily remap a virtual machine's host physical memory to files or other devices in a manner that is completely transparent to the virtual machine.

Recently, third generation AMD Opteron and Intel Xeon 5500 series processors have provided hardware support for memory virtualization by using two layers of page tables in hardware. One layer stores the guest virtual to guest physical memory address translation, and the other layer stores the guest physical to host physical memory address translation. These two page tables are synchronized using processor hardware. Support for hardware memory virtualization eliminates the overhead required to keep shadow page tables in synchronization with guest page tables in software memory virtualization. For more information about hardware-assisted memory virtualization, see *Performance Evaluation of Intel EPT Hardware Assist*[5] and *Performance Evaluation of AMD RVI Hardware Assist*[6].

Memory Management Basics in ESXi

Prior to describing how ESXi manages memory for virtual machines, it is useful to first understand how the application, guest operating system, hypervisor, and virtual machine manage memory at their respective layers.

- An application starts and uses the interfaces provided by the operating system to explicitly allocate or de-allocate virtual memory during its execution.
- In a non-virtualized environment, the operating system assumes it owns all physical memory in the system. The hardware does not provide interfaces for the operating system to explicitly "allocate" or "free" physical memory. The operating system establishes the definitions of "allocated" or "free" physical memory. Different operating systems have different implementations to realize this abstraction. One example is that the operating system maintains an "allocated" list and a "free" list, so whether or not a physical page is free depends on which list the page currently resides in.

- Because a virtual machine runs an operating system and several applications, the virtual machine memory management properties combine both application and operating system memory management properties. Like an application, when a virtual machine first starts, it has no pre-allocated physical memory. Like an operating system, the virtual machine cannot explicitly allocate host physical memory through any standard interface. The hypervisor also creates the definitions of "allocated" and "free" host memory in its own data structures. The hypervisor intercepts the virtual machine's memory accesses and allocates host physical memory for the virtual machine on its first access to the memory. In order to avoid information leaking among virtual machines, the hypervisor always writes zeroes to the host physical memory before assigning it to a virtual machine.
- Virtual machine memory deallocation acts just like it does in an operating system. The guest operating system frees a piece of physical memory by adding these memory page numbers to the guest free list, but the data of the "freed" memory may not be modified at all. As a result, when a particular piece of guest physical memory is freed, the mapped host physical memory will usually not change its state and only the guest free list will be changed.

The hypervisor knows when to allocate host physical memory for a virtual machine because the first memory access from the virtual machine to a host physical memory will cause a page fault that can be easily captured by the hypervisor. However, it is difficult for the hypervisor to know when to free host physical memory upon virtual machine memory deallocation because the guest operating system free list is generally not publicly accessible. Hence, the hypervisor cannot easily find out the location of the free list and monitor its changes.

Although the hypervisor cannot reclaim host memory when the operating system frees guest physical memory, this does not mean that the host memory, no matter how large it is, will be used up by a virtual machine when the virtual machine repeatedly allocates and frees memory. This is because the hypervisor does not allocate host physical memory on every virtual machine's memory allocation. It only allocates host physical memory when the virtual machine touches the physical memory that it has never touched before. If a virtual machine frequently allocates and frees memory, presumably the same guest physical memory is being allocated and freed again and again. Therefore, the hypervisor just allocates host physical memory for the first memory allocation and then the guest reuses the same host physical memory for the rest of the allocations. That is, if a virtual machine's entire guest physical memory (configured memory) has been backed by the host physical memory, the hypervisor does not need to allocate any more host physical memory for this virtual machine. This means that Equation 1 always holds true:

VM's host memory usage <= VM's guest memory size + VM's overhead memory **(1)**

Here, the virtual machine's overhead memory is the extra host memory needed by the hypervisor for various virtualization data structures besides the memory allocated to the virtual machine. Its size depends on the number of virtual CPUs and the configured virtual machine memory size. For more information, see *vSphere Resource Management*[2].

Memory Reclamation in ESXi

ESXi uses several innovative techniques to reclaim virtual machine memory, which are:

- Transparent page sharing (TPS)—reclaims memory by removing redundant pages with identical content
- Ballooning—reclaims memory by artificially increasing the memory pressure inside the guest
- Hypervisor swapping—reclaims memory by having ESXi directly swap out the virtual machine's memory
- Memory compression—reclaims memory by compressing the pages that need to be swapped out

The following sections describe these techniques and the motivation behind memory reclamation.

Motivation

According to Equation (1), if the hypervisor cannot reclaim host physical memory upon virtual machine memory deallocation, it must reserve enough host physical memory to back all virtual machines' guest physical memory (plus their overhead memory) in order to prevent any virtual machine from running out of host physical memory. This means that memory overcommitment cannot be supported. The concept of memory overcommitment is fairly simple: host memory is overcommitted when the total amount of guest physical memory of the running virtual machines is larger than the amount of actual host memory. ESXi supports memory overcommitment from the very first version, due to two important benefits it provides:

- Higher memory utilization: With memory overcommitment, ESXi ensures that host memory is consumed by active guest memory as much as possible. Typically, some virtual machines may be lightly loaded compared to others. Their memory may be used infrequently, so for much of the time their memory will sit idle. Memory overcommitment allows the hypervisor to use memory reclamation techniques to take the inactive or unused host physical memory away from the idle virtual machines and give it to other virtual machines that will actively use it.
- Higher consolidation ratio: With memory overcommitment, each virtual machine has a smaller footprint in host memory usage, making it possible to fit more virtual machines on the host while still achieving good performance for all virtual machines. For example, as shown in Figure 3, you can enable a host with 4G host physical memory to run three virtual machines with 2G guest physical memory each. Without memory overcommitment, only one virtual machine can be run because the hypervisor cannot reserve host memory for more than one virtual machine, considering that each virtual machine has overhead memory.

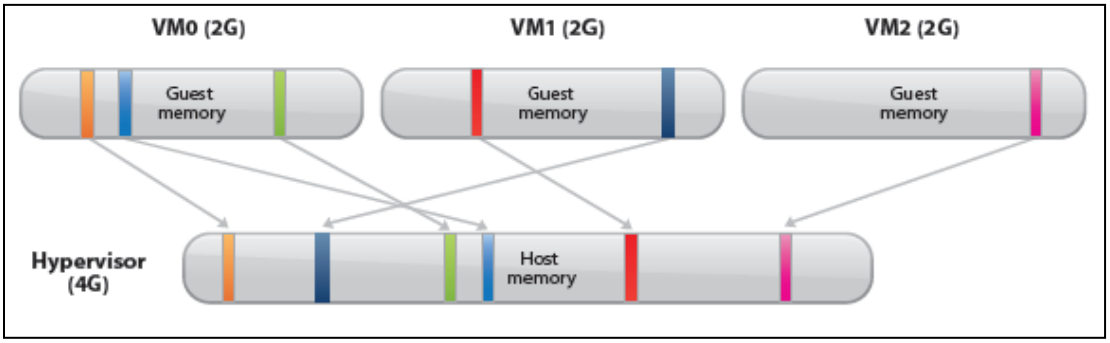

Figure 3. Memory Overcommitment in ESXi

In order to effectively support memory overcommitment, the hypervisor must provide efficient host memory reclamation techniques. ESXi leverages several innovative techniques to support virtual machine memory reclamation. These techniques are transparent page sharing, ballooning, memory compression, and hypervisor swapping.

Transparent Page Sharing (TPS)

When multiple virtual machines are running, some of them may have identical sets of memory content. This presents opportunities for sharing memory across virtual machines (as well as sharing within a single virtual machine). For example, several virtual machines may be running the same guest operating system, have the same applications, or contain the same user data. With page sharing, the hypervisor can reclaim the redundant copies and keep only one copy, which is shared by multiple virtual machines in the host physical memory. As a result, the total virtual machine host memory consumption is reduced and a higher level of memory overcommitment is possible.

In ESXi, the redundant page copies are identified by their contents. This means that pages with identical content can be shared regardless of when, where, and how those contents are generated. ESXi scans the content of guest physical memory for sharing opportunities. Instead of comparing each byte of a candidate guest physical page to other pages, an action that is prohibitively expensive, ESXi uses hashing to identify potentially identical pages. The detailed algorithm is illustrated in Figure 4.

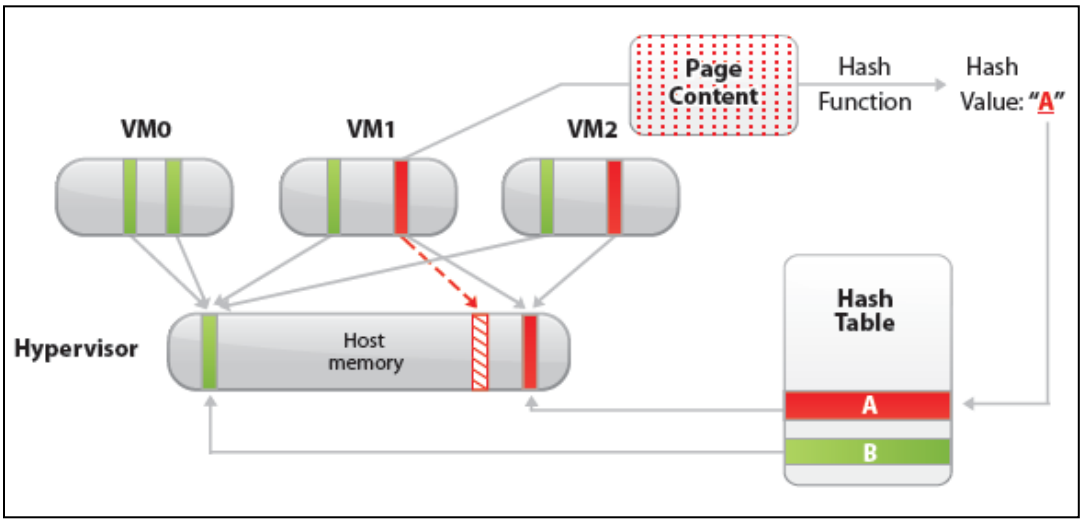

Figure 4. Content-Based Page Sharing in ESXi

A hash value is generated based on the candidate guest physical page's content. The hash value is then used as a key to look up a global hash table, in which each entry records a hash value and the physical page number of a shared page. If the hash value of the candidate guest physical page matches an existing entry, a full comparison of the page contents is performed to exclude a false match. Once the candidate guest physical page's content is confirmed to match the content of an existing shared host physical page, the guest physical to host physical mapping of the candidate guest physical page is changed to the shared host physical page, and the redundant host memory copy (the page pointed to by the dashed arrow in Figure 4) is reclaimed. This remapping is invisible to the virtual machine and inaccessible to the guest operating system. Because of this invisibility, sensitive information cannot be leaked from one virtual machine to another.

A standard copy-on-write (CoW) technique is used to handle writes to the shared host physical pages. Any attempt to write to the shared pages will generate a minor page fault. In the page fault handler, the hypervisor will transparently create a private copy of the page for the virtual machine and remap the affected guest physical page to this private copy. In this way, virtual machines can safely modify the shared pages without disrupting other virtual machines sharing that memory. Note that writing to a shared page does incur overhead compared to writing to non-shared pages due to the extra work performed in the page fault handler.

In VMware ESXi, the hypervisor scans the guest physical pages randomly with a base scan rate specified by `Mem.ShareScanTime`, which specifies the desired time to scan the virtual machine's entire guest memory. The maximum number of scanned pages per second in the host and the maximum number of per-virtual machine scanned pages, (that is, `Mem.ShareScanGHz` and `Mem.ShareRateMax` respectively) can also be specified in ESXi advanced settings. An example is shown in Figure 5.

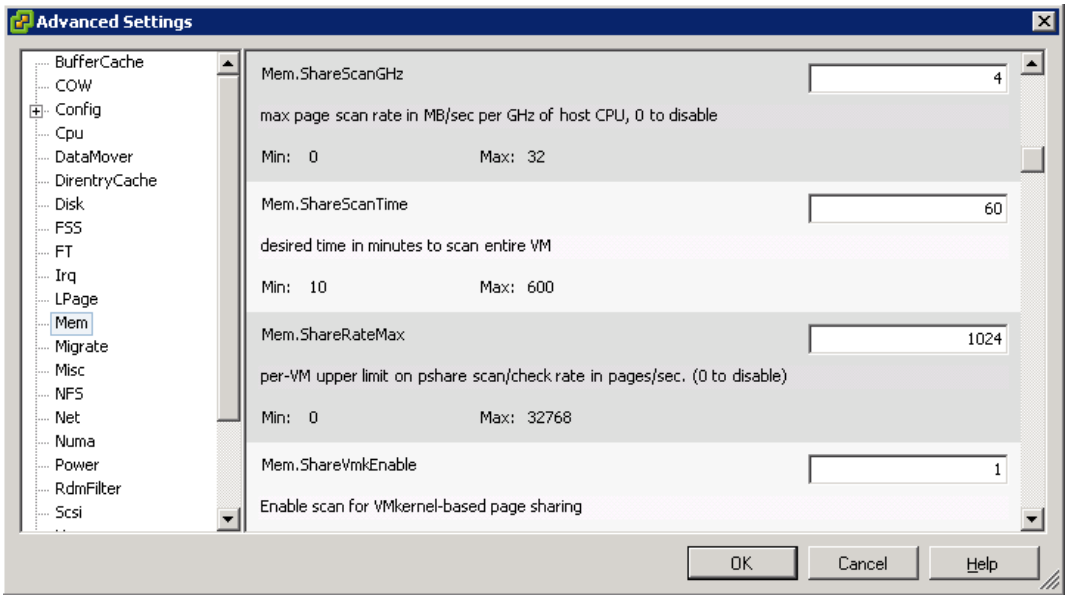

Figure 5. Configure Page Sharing in vSphere Client

The default values of these three parameters are carefully chosen to provide sufficient sharing opportunities while keeping the CPU overhead negligible. In fact, ESXi intelligently adjusts the page scan rate based on the amount of current shared pages. If the virtual machine's page sharing opportunity seems to be low, the page scan rate will be reduced accordingly and vice versa. This optimization further mitigates the overhead of page sharing.

In hardware-assisted memory virtualization (for example, Intel EPT Hardware Assist and AMD RVI Hardware Assist[6]) systems, ESXi will automatically back guest physical pages with large host physical pages (2MB contiguous memory region instead of 4KB for regular pages) for better performance due to less TLB misses. In such systems, ESXi will not share those large pages because:

- The probability of finding two large pages having identical contents is low
- The overhead of doing a bit-by-bit comparison for a 2MB page is much larger than for a 4KB page

However, ESXi still generates hashes for the 4KB pages within each large page. Since ESXi will not swap out large pages, the large page will be broken into small pages during host swapping so that these pre-generated hashes can be used to share the small pages before they are swapped out. In short, we may not observe any page sharing for hardware-assisted memory virtualization systems until host memory is overcommitted.

Ballooning

Ballooning is a completely different memory reclamation technique compared to transparent page sharing. Before describing the technique, it is helpful to review why the hypervisor needs to reclaim memory from virtual machines. Due to the virtual machine's isolation, the guest operating system is not aware that it is running inside a virtual machine and is not aware of the states of other virtual machines on the same host. When the hypervisor runs multiple virtual machines and the total amount of the free host memory becomes low, none of the virtual machines will free guest physical memory because the guest operating system cannot detect the host's memory shortage. Ballooning makes the guest operating system aware of the low memory status of the host.

In ESXi, a balloon driver is loaded into the guest operating system as a pseudo-device driver.[‡] It has no external interfaces to the guest operating system and communicates with the hypervisor through a private channel. The balloon driver polls the hypervisor to obtain a target balloon size. If the hypervisor needs to reclaim virtual

[‡] VMware Tools must be installed in order to enable ballooning. This is recommended for all workloads.

machine memory, it sets a proper target balloon size for the balloon driver, making it "inflate" by allocating guest physical pages within the virtual machine. Figure 6 illustrates the process of the balloon inflating.

In Figure 6 (a), four guest physical pages are mapped in the host physical memory. Two of the pages are used by the guest application and the other two pages (marked by stars) are in the guest operating system free list. Note that since the hypervisor cannot identify the two pages in the guest free list, it cannot reclaim the host physical pages that are backing them. Assuming the hypervisor needs to reclaim two pages from the virtual machine, it will set the target balloon size to two pages.

After obtaining the target balloon size, the balloon driver allocates two guest physical pages inside the virtual machine and pins them, as shown in Figure 6 (b). Here, "pinning" is achieved through the guest operating system interface, which ensures that the pinned pages cannot be paged out to disk under any circumstances. Once the memory is allocated, the balloon driver notifies the hypervisor about the page numbers of the pinned guest physical memory so that the hypervisor can reclaim the host physical pages that are backing them. In Figure 6 (b), dashed arrows point at these pages. The hypervisor can safely reclaim this host physical memory because neither the balloon driver nor the guest operating system relies on the contents of these pages. This means that no processes in the virtual machine will intentionally access those pages to read/write any values. Thus, the hypervisor does not need to allocate host physical memory to store the page contents. If any of these pages are re-accessed by the virtual machine for some reason, the hypervisor will treat it as a normal virtual machine memory allocation and allocate a new host physical page for the virtual machine. When the hypervisor deflates the balloon—by setting a smaller target balloon size—the balloon driver deallocates the pinned guest physical memory, which releases it for the guest's applications.

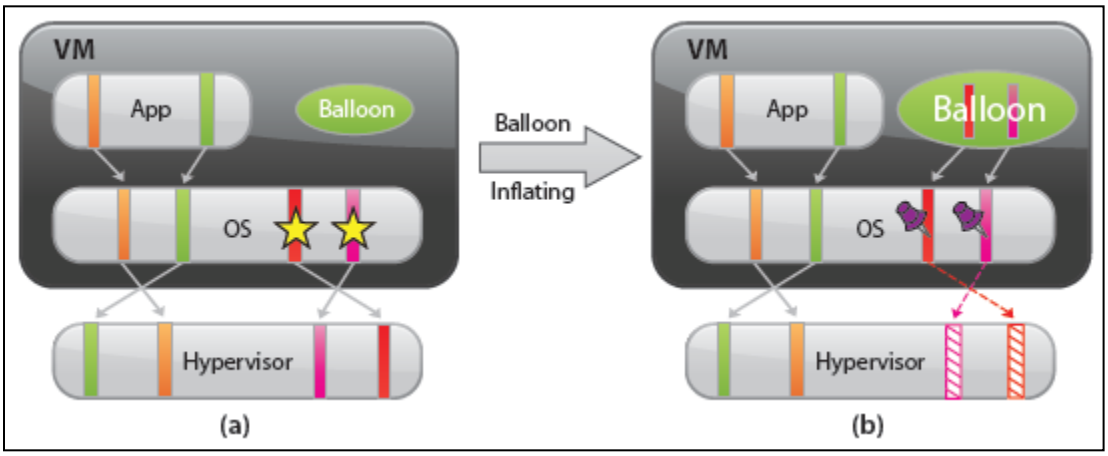

Figure 6. Inflating the Balloon in a Virtual Machine

Typically, the hypervisor inflates the virtual machine balloon when it is under memory pressure. By inflating the balloon, a virtual machine consumes less physical memory on the host, but more physical memory inside the guest. As a result, the hypervisor offloads some of its memory overload to the guest operating system while slightly loading the virtual machine. That is, the hypervisor transfers the memory pressure from the host to the virtual machine. Ballooning induces guest memory pressure. In response, the balloon driver allocates and pins guest physical memory. The guest operating system determines if it needs to page out guest physical memory to satisfy the balloon driver's allocation requests. If the virtual machine has plenty of free guest physical memory, inflating the balloon will induce no paging and will not impact guest performance. In this case, as illustrated in Figure 6, the balloon driver allocates the free guest physical memory from the guest free list. Hence, guest-level paging is not necessary. However, if the guest is already under memory pressure, the guest operating system decides which guest physical pages to be paged out to the virtual swap device in order to satisfy the balloon driver's allocation requests. The genius of ballooning is that it allows the guest operating system to intelligently make the hard decision about which pages to be paged out without the hypervisor's involvement.

For ballooning to work as intended, the guest operating system must install and enable the balloon driver, which is included in VMware Tools. The guest operating system must have sufficient virtual swap space configured for guest paging to be possible. Ballooning might not reclaim memory quickly enough to satisfy host memory demands. In addition, the upper bound of the target balloon size may be imposed by various guest operating system limitations.

Memory Compression

With memory compression, ESXi stores pages, which would otherwise be swapped out to disk through host swapping, in a compression cache located in the main memory. Memory compression outperforms host swapping because the next access to the compressed page only causes a page decompression, which can be an order of magnitude faster than the disk access.

ESXi determines if a page can be compressed by checking the compression ratio for the page. Memory compression occurs when the page's compression ratio is greater than 50%. Otherwise, the page is swapped out. Only pages that would otherwise be swapped out to disk are chosen as candidates for memory compression. This means ESXi will not proactively compress guest pages when host swapping is not necessary. In other words, memory compression does not affect workload performance when host memory is undercommitted.

Reclaiming Memory Through Compression

Figure 7 illustrates how memory compression reclaims host memory compared to host swapping. Assume ESXi needs to reclaim 8KB physical memory (that is, two 4KB pages) from a VM. With host swapping, two swap candidate pages, page A and B, are directly swapped to disk (Figure 7(a)). With memory compression, a swap candidate page is compressed and stored using 2KB[§] of space in a per-VM compression cache. Hence, each compressed page yields 2KB memory space for ESXi to reclaim. In order to reclaim 8KB physical memory, four swap candidate pages need to be compressed (Figure 7(b)).

The page compression is much faster than the normal page swap-out operation, which involves a disk I/O.

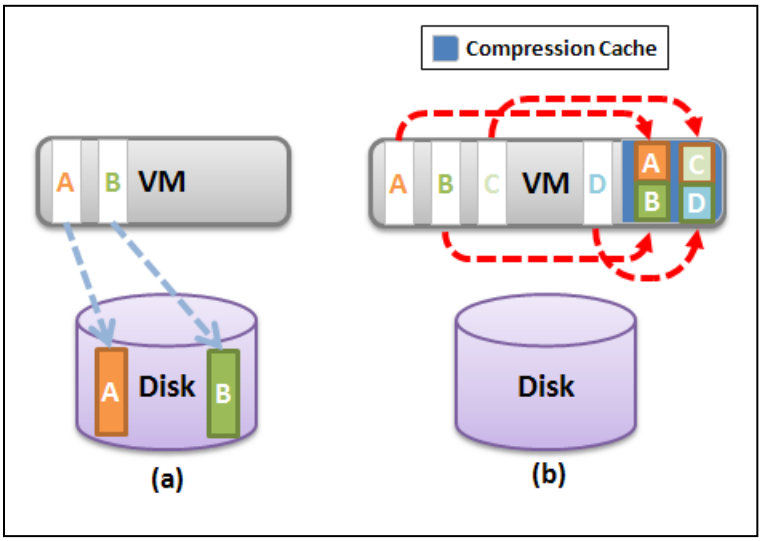

Figure 7. Host Swapping vs. Memory Compression in ESXi

[§] For more efficient usage of the compression cache, if a page's compression ratio is larger than 75%, ESXi will store the compressed page using a 1KB quarter-page space. In this example, we assume the page compression ratio is 50% for simplicity.

If any of the subsequent memory requests access a compressed page, the page is decompressed and pushed back to the guest memory. The page is then removed from the compression cache.

Managing Per-VM Compression Cache

The VM's guest memory usage accounts for the per-VM compression cache. The memory for the compression cache is not allocated separately as an extra overhead memory. The compression cache size starts with zero when host memory is undercommitted and grows when the virtual machine memory starts to be swapped out.

If the compression cache is full, one compressed page must be replaced in order to make room for a new compressed page. The page which has not been accessed for the longest time will be decompressed and swapped out. ESXi will not swap out compressed pages.

If the pages belonging to compression cache need to be swapped out under severe memory pressure, the compression cache size is reduced and the affected compressed pages are decompressed and swapped out.

The maximum compression cache size is important for maintaining good VM performance. If the upper bound is too small, a lot of replaced compressed pages need to be decompressed and swapped out. Any following swap-ins of those pages will hurt VM performance. However, since compression cache is accounted for by the VM's guest memory usage, a very large compression cache may waste VM memory and unnecessarily create host memory pressure, especially when most compressed pages would not be touched in the future. In vSphere 5.0, the default maximum compression cache size is conservatively set to 10% of configured VM memory size. This value can be changed through the vSphere Client in Advanced Settings by changing the value for `Mem.MemZipMaxPct`, which is shown in Figure 8.

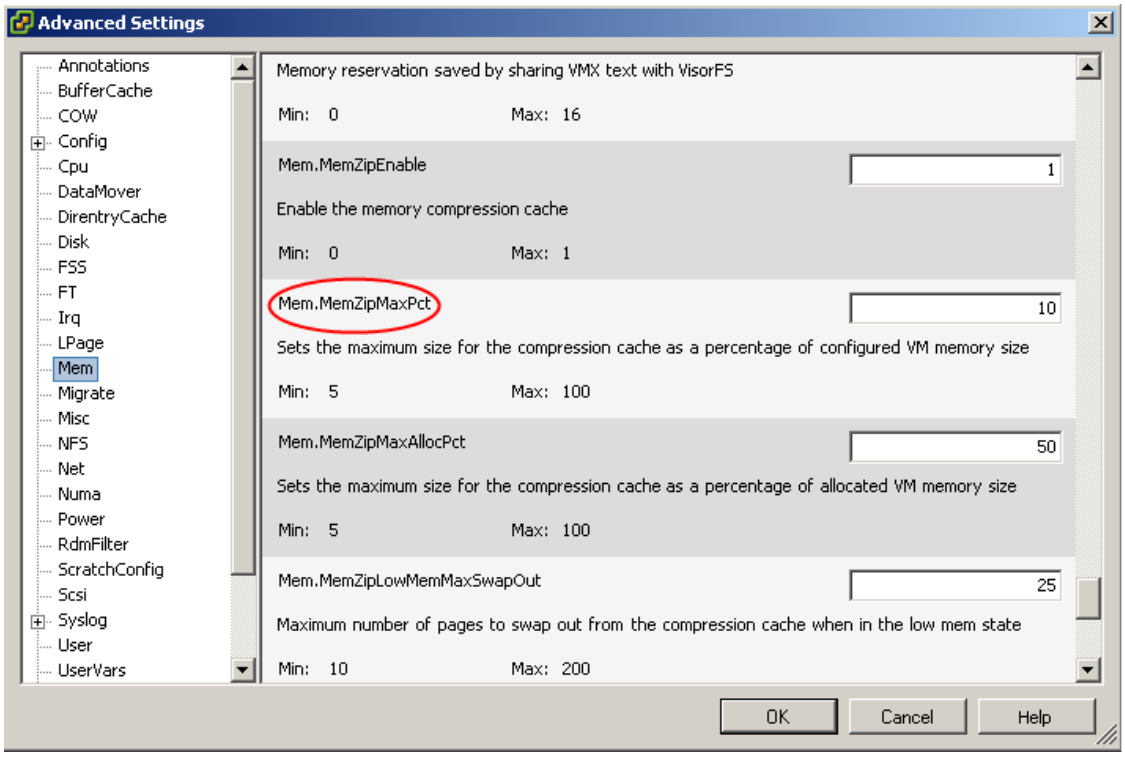

Figure 8. Change the Maximum Compression Cache Size in the vSphere Client

Hypervisor Swapping

In the cases where ballooning, transparent page sharing, and memory compression are not sufficient to reclaim memory, ESXi employs hypervisor swapping to reclaim memory. At virtual machine startup, the hypervisor creates a separate swap file for the virtual machine. Then, if necessary, the hypervisor can directly swap out guest physical memory to the swap file, which frees host physical memory for other virtual machines.

Both page sharing and ballooning take time to reclaim memory. The page-sharing speed depends on the page scan rate and the sharing opportunity. Ballooning speed relies on the guest operating system's response time for memory allocation.

In contrast, hypervisor swapping is a guaranteed technique to reclaim a specific amount of memory within a specific amount of time. However, hypervisor swapping is used as a last resort to reclaim memory from the virtual machine due to the following limitations on performance:

- Page selection problems: Under certain circumstances, hypervisor swapping may severely penalize guest performance. This occurs when the hypervisor has no knowledge about which guest physical pages should be swapped out, and the swapping may cause unintended interactions with the native memory management policies in the guest operating system.
- Double paging problems: Another known issue is the double paging problem. Assuming the hypervisor swaps out a guest physical page, it is possible that the guest operating system pages out the same physical page, if the guest is also under memory pressure. This causes the page to be swapped in from the hypervisor swap device and immediately to be paged out to the virtual machine's virtual swap device.
- Page selection and double-paging problems exist because the information needed to avoid them is not available to the hypervisor.
- High swap-in latency: Swapping in pages is expensive for a VM. If the hypervisor swaps out a guest page and the guest subsequently accesses that page, the VM will get blocked until the page is swapped in from disk. High swap-in latency, which can be tens of milliseconds, can severely degrade guest performance.

ESXi employs three methods to address the above limitations in order to improve hypervisor swapping performance:

- ESXi mitigates the impact of interacting with guest operating system memory management by randomly selecting the swapped guest physical pages.
- ESXi applies memory compression to reduce the amount of pages that need to be swapped out, while reclaiming the same amount of host memory.
- ESXi can be configured to swap to SSD (new in vSphere 5.0). If an SSD device is installed in the host, the user can choose to configure a host cache in the SSD (an example is shown in Figure 9). ESXi will then use the host cache to store the swapped out pages instead of putting them in the regular hypervisor swap file. Upon the next access to a page in the host cache, the page will be pushed back to the guest memory and then removed from the host cache. Since SSD read latency, which is normally around a few hundred microseconds, is much faster than typical disk access latency, this optimization significantly reduces the swap-in latency and hence greatly improves the application performance in high memory overcommitment scenarios.

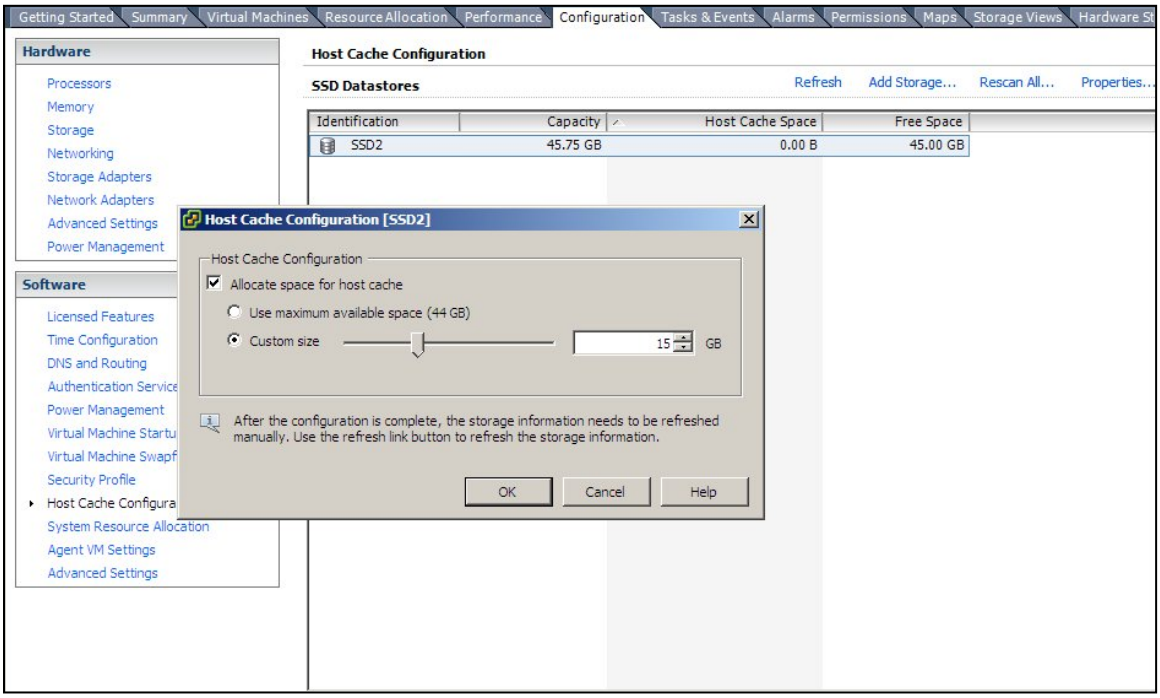

Figure 9. Configure Host Swap Cache through vSphere Client

When to Reclaim Host Memory**

ESXi maintains four host free memory states: high, soft, hard, and low, which are reflected by four thresholds. The threshold values are calculated based on host memory size. Figure 9 shows how the host free memory state is reported in esxtop. The "minfree" value represents the threshold for the high state.

By default, ESXi enables page sharing since it opportunistically "frees" host memory with little overhead. When to use ballooning or swapping (which activates memory compression) to reclaim host memory is largely determined by the current host free memory state.

** The discussions and conclusions made in this section may not be valid when the user specifies a resource pool for virtual machines. For example, if the resource pool that contains a virtual machine is specified as a small memory limit, ballooning or hypervisor swapping occur for the virtual machine even when the host free memory is in the high state. The detailed explanation of resource pool is beyond the scope of this paper. Most of the details can be found in the "Managing Resource Pools" section of the *vSphere Resource Management Guide*[2].

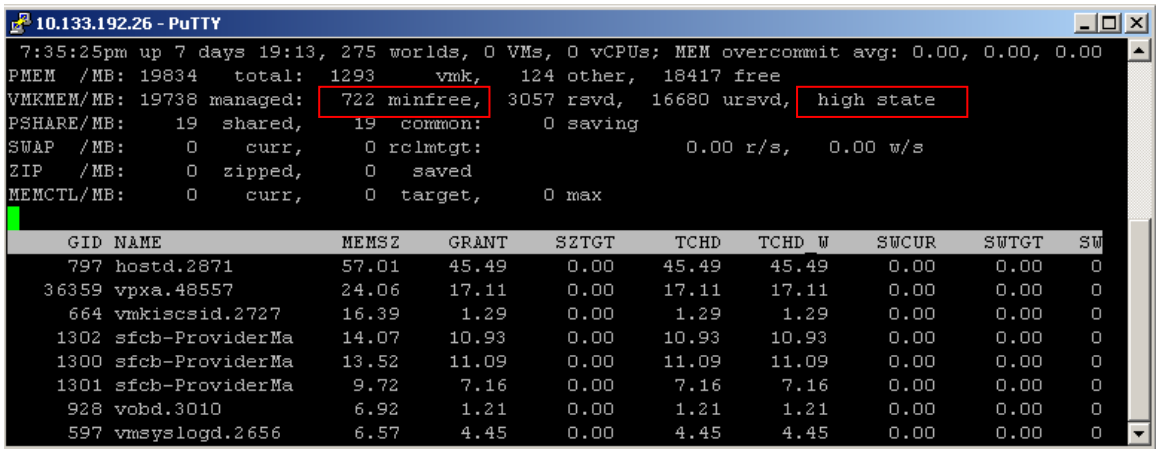

Figure 9. MinFree and Host Free Memory State in esxtop

In the high state, the aggregate virtual machine guest memory usage is smaller than the host memory size. Whether or not host memory is overcommitted, the hypervisor will not reclaim memory through ballooning or swapping. (This is true only when the virtual machine memory limit is not set.)

If host free memory drops towards the soft threshold, the hypervisor starts to reclaim memory using ballooning. Ballooning happens before free memory actually reaches the soft threshold because it takes time for the balloon driver to allocate and pin guest physical memory. Usually, the balloon driver is able to reclaim memory in a timely fashion so that the host free memory stays above the soft threshold.

If ballooning is not sufficient to reclaim memory or the host free memory drops towards the hard threshold, the hypervisor starts to use swapping in addition to using ballooning. During swapping, memory compression is activated as well. With host swapping and memory compression, the hypervisor should be able to quickly reclaim memory and bring the host memory state back to the soft state.

In a rare case where host free memory drops below the low threshold, the hypervisor continues to reclaim memory through swapping and memory compression, and additionally blocks the execution of all virtual machines that consume more memory than their target memory allocations.

In certain scenarios, host memory reclamation happens regardless of the current host free memory state. For example, even if host free memory is in the high state, memory reclamation is still mandatory when a virtual machine's memory usage exceeds its specified memory limit. If this happens, the hypervisor will employ ballooning and, if necessary, swapping and memory compression to reclaim memory from the virtual machine until the virtual machine's host memory usage falls back to its specified limit.

ESXi Memory Allocation Management for Multiple Virtual Machines

This section describes how ESXi allocates host memory to multiple virtual machines, especially when the host memory is overcommitted.

ESXi employs a share-based allocation algorithm to achieve efficient memory utilization for all virtual machines and to guarantee memory to those virtual machines which need it most[1].

ESXi provides three configurable parameters to control the host memory allocation for a virtual machine: Shares, Reservation, and Limit. The interface in the vSphere Client is shown in Figure 10.

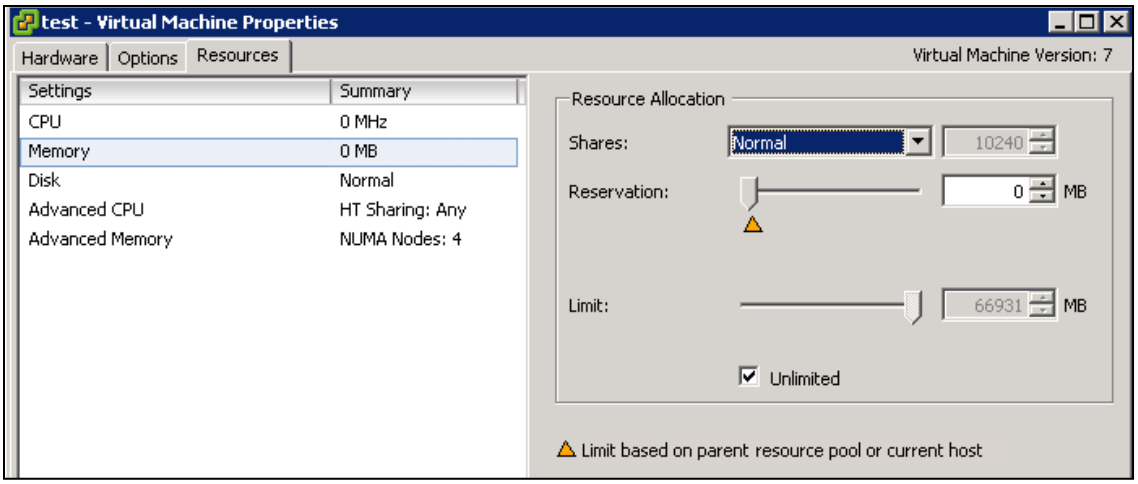

Figure 10. Configure Virtual Machine Memory Allocation

Limit is the upper bound of the amount of host physical memory allocated for a virtual machine. By default, Limit is set to unlimited, which means a virtual machine's maximum allocated host physical memory is its specified virtual machine memory size (according to Equation (1) on page 6). Reservation is a guaranteed lower bound on the amount of host physical memory the host reserves for a virtual machine even when host memory is overcommitted. Memory Shares entitle a virtual machine to a fraction of available host physical memory, based on a proportional-share allocation policy. For example, a virtual machine with twice as many shares as another is generally entitled to consume twice as much memory, subject to its limit and reservation constraints.

Periodically, ESXi computes a memory allocation target for each virtual machine based on its share-based entitlement, its estimated working set size, and its limit and reservation. Here, a virtual machine's working set size is defined as the amount of guest physical memory that is actively being used. When host memory is undercommitted, a virtual machine's memory allocation target is the virtual machine's consumed host physical memory size with headroom. The maximum memory allocation target is:

Maximum allocation target = min{ VM's memory size, VM's limit } **(2)**

When host memory is overcommitted, a virtual machine's allocation target is somewhere between its specified reservation and specified limit depending on the virtual machine's shares and the system load. If a virtual machine's host memory usage is larger than the computed allocation target, which typically happens in memory overcommitment cases, ESXi employs a ballooning or swapping mechanism to reclaim memory from the virtual machine in order to reach the allocation target. Whether to use ballooning or to use swapping is determined by the current host free memory state as described in previous sections.

Shares play an important role in determining the allocation targets when memory is overcommitted. When the hypervisor needs memory, it reclaims memory from the virtual machine that owns the fewest shares-per-allocated page.

A significant limitation of the pure proportional-share algorithm is that it does not incorporate any information about the actual memory usage of the virtual machine. As a result, some idle virtual machines with high shares can retain idle memory unproductively, while some active virtual machines with fewer shares suffer from lack of memory.

ESXi resolves this problem by estimating a virtual machine's working set size and charging a virtual machine more for the idle memory than for the actively used memory through an idle tax[1]. A virtual machine's shares-per-allocated page ratio is adjusted to be lower if a fraction of the virtual machine's memory is idle. Hence, memory will be reclaimed preferentially from the virtual machines that are not fully utilizing their allocated memory. The detailed algorithm can be found in *Memory Resource Management in VMware ESX Server*[1], in section 5.2,

"Reclaiming Idle Memory." The effectiveness of this algorithm relies on the accurate estimation of the virtual machine's working set size. ESXi uses a statistical sampling approach to estimate the aggregate virtual machine working set size without any guest involvement. At the beginning of each sampling period, the hypervisor intentionally invalidates several randomly selected guest physical pages and starts to monitor the guest accesses to them. At the end of the sampling period, the fraction of actively used memory can be estimated as the fraction of the invalidated pages that are re-accessed by the guest during the epoch. The detailed algorithm can be found in *Memory Resource Management in VMware ESX Server*[1], in section 5.3 "Measuring Idle Memory." By default, ESXi samples 100 guest physical pages for each 60-second period. The sampling rate can be adjusted by changing `Mem.SamplePeriod` through the vSphere Client in Advanced Settings.

By overpricing the idle memory and effective working set estimation, ESXi is able to efficiently allocate host memory under memory overcommitment while maintaining the proportional-share allocation.

Performance Evaluation

In this section, the performance of various ESXi memory reclamation techniques is evaluated. The purpose is to help users understand how individual techniques impact the performance of various applications.

Experimental Environment

ESX 4.0 RC was installed on a Dell PowerEdge 6950 system and experiments were conducted against SPECjbb, Kernel Compile, Swingbench, and Exchange workloads to evaluate page sharing, ballooning, and host swapping performance. ESX 4.1 RC was installed on a Dell PowerEdge R710 system to evaluate the memory compression feature using SharePoint and Swingbench workloads. ESXi 5.0 RTM was installed on a Dell PowerEdge R905 system to evaluate the swap to SSD feature using the Swingbench workload. The system hardware configurations and workload descriptions are summarized in Table 1 and Table 2 respectively.

PowerEdge 6950	4 socket dual core AMD Opteron 8222SE processors @ 3GHZ, 64GB memory
PowerEdge R710	2 quad-core Intel Xeon X5570 (Nehalem) processors @ 2.93GHz, with hyper-threading, 96GB memory
PowerEdge R905	4 socket quad-core AMD Opteron 8378 processors @2.4GHz, 72GB memory
SAN Storage	2TB Fibre Channel
SSD	Dell 50GB Solid State Drive SATA

Table 1. Server Configurations

SPECjbb2005	Parameters:	2.5GB Java heap, 1 warehouse, 10 minutes run
	VM configuration:	1 vCPU, 4GB memory
Kernel Compile	Parameters:	Linux 2.6.17 Kernel, run "make -j 1 bzimage > /dev/null"
	VM configuration:	1 vCPU, 512MB memory
Swingbench	Parameters:	Oracle 11g database, Order Entry functional benchmark, 20 minutes run
	VM configuration:	4 vCPUs, 5GB memory
Exchange 2007	Parameters:	Microsoft Exchange 2007 Server, 2000 heavy users driven by LoadGen client
	VM configuration:	4 vCPUs, 12GB memory
SharePoint 2007	Parameters:	SQL Server: SQL server 2008 SP1 with 200GB Sharepoint user data Web Server: Windows Server 2008 with Web Server and Query Server roles enabled Application Server: Index server
	VM configurations:	3 Web Server VMs (2 vCPUs, 4GB memory) 1 Application Server VM (2 vCPUs, 4GB memory) 1 SQL Server VM (2 vCPUs, 16GB memory)

Table 2. Workload Descriptions

The guest operating system running inside the SPECjbb, kernel compile, and Swingbench virtual machines was 64-bit Red Hat Enterprise Linux 5.2 Server. The guest operating system running inside the Exchange virtual machine and SharePoint VMs was Windows Server 2008.

For SPECjbb2005 and Swingbench, the throughput was measured by calculating the number of transactions per second. For kernel compile, the performance was measured by calculating the inverse of the compilation time. For Exchange, the performance was measured using the average Remote Procedure Call (RPC) latency. For SharePoint, the performance was measured using the number of requests per second. In addition—for Swingbench, Exchange, and SharePoint—the client applications were installed in a separate native machine.

Transparent Page Sharing Performance

In this experiment, two instances of workloads were run. The overall performance of workloads with page sharing enabled is compared to that with page sharing disabled. The focus is on evaluating the overhead of page scanning. Since the page scan rate (number of scanned pages per second) is largely determined by the `Mem.ShareScanTime`[††]

[††] This is an advanced setting, shown in Figure 5 on page 9.

, in addition to the default 60 minutes, the minimal `Mem.ShareScanTime` of 10 minutes was tested, which potentially introduces the highest page scanning overhead.

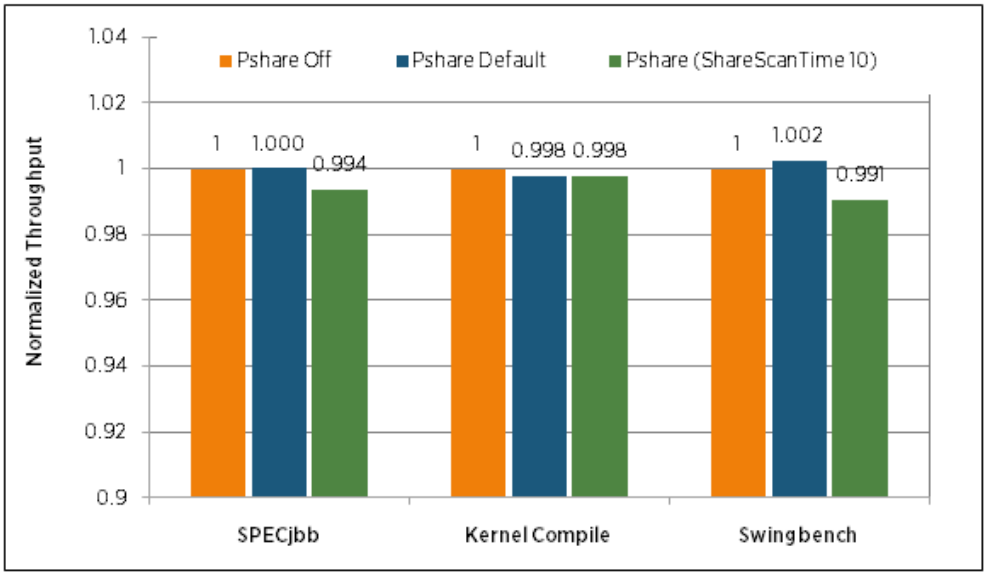

Figure 11. Performance Impact of Transparent Page Sharing

Figure 11 confirms that enabling page sharing introduces a negligible performance overhead in the default setting and only less than 1% overhead when `Mem.ShareScanTime` is 10 minutes for all workloads.

Page sharing sometimes improves performance because the virtual machine's host memory footprint is reduced so that it fits the processor cache better.

Besides page scanning, breaking copy-on-right (CoW) pages is another source of page sharing overhead. Unfortunately, such overhead is highly application dependent and it is difficult to evaluate it through configurable options. Therefore, the overhead of breaking CoW pages was omitted in this experiment.

Ballooning vs. Host Swapping

In the following experiments, virtual machine memory reclamation was enforced by changing each virtual machine's memory limit value from the default of unlimited, to values that are smaller than the configured virtual machine memory size. Page sharing was turned off to isolate the performance impact of ballooning or swapping. Since the host memory is much larger than the virtual machine memory size, the host free memory is always in the high state. Hence, by default, ESXi only uses ballooning to reclaim memory. Both ballooning and memory compression were turned off to obtain the performance of using swapping only. The ballooned and swapped memory sizes were also collected when the virtual machine ran steadily.

Linux Kernel Compile

Figure 12 presents the throughput of the kernel compile workload with different memory limits when using ballooning or swapping. This experiment was contrived to use only ballooning or swapping, not both. While this case will not often occur in production environments, it shows the performance penalty due to either technology on its own. The throughput is normalized to the case where virtual machine memory is not reclaimed.

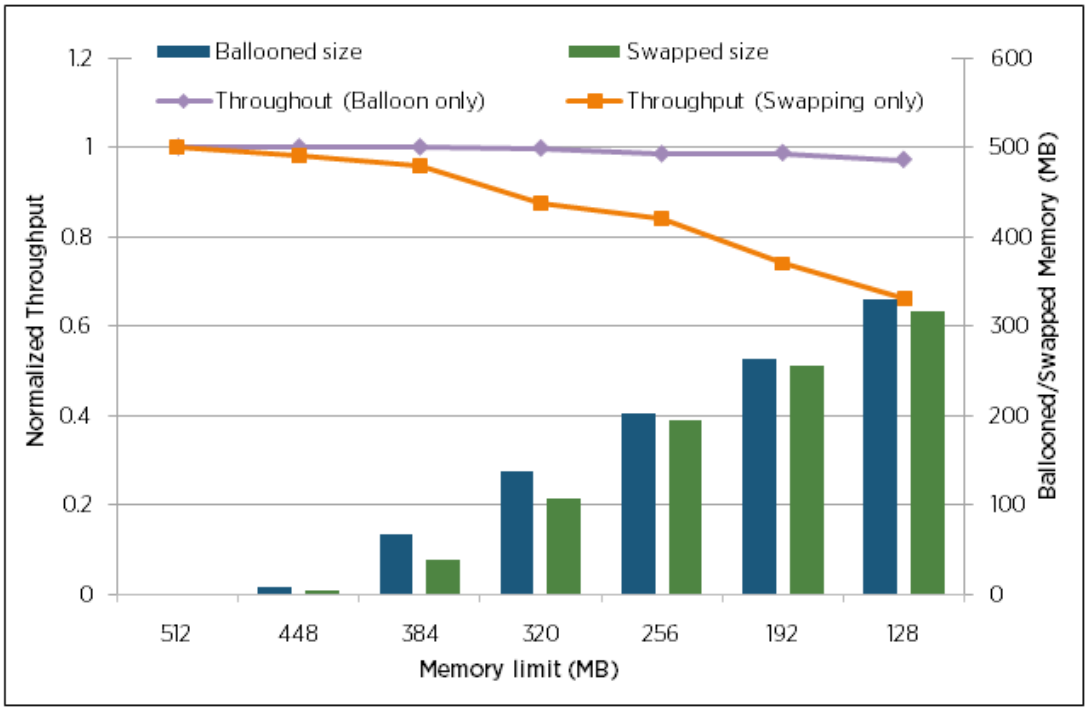

Figure 12. Performance of Kernel Compile when Using Ballooning vs. Swapping

By using ballooning, the kernel compile virtual machine only suffers from 3% throughput loss even when the memory limit is as low as 128MB (1/4 of the configured virtual machine memory size). This is because the kernel compile workload has very little memory reuse and most of the guest physical memory is used as buffer caches for the kernel source files. With ballooning, the guest operating system reclaims guest physical memory upon the balloon driver's allocation request by dropping the buffer pages instead of paging them out to the guest virtual swap device. Because dropped buffer pages are not reused frequently, the performance impact of using ballooning is trivial.

However, with hypervisor swapping, the selected guest buffer pages are unnecessarily swapped out to the host swap device and some guest kernel pages are swapped out occasionally, making the performance of the virtual machine degrade when the memory limit decreases. When the memory limit is 128MB, the throughput loss is about 34% in the swapping case. Balloon inflation is a better approach to memory reclamation from a performance perspective.

Figure 12 illustrates that as the memory limit decreases, the ballooned and swapped memory sizes increase almost linearly. There is a difference between the ballooned memory size and the swapped memory size. In the ballooning cases, when virtual machine memory usage exceeds the specified limit, the balloon driver cannot force the guest operating system to page out guest physical pages immediately unless the balloon driver has used up most of the free guest physical memory, which puts the guest operating system under memory pressure. In the swapping cases, however, as long as the virtual machine memory usage exceeds the specified limit, the extra amount of pages will be swapped out immediately. Therefore, the ballooned memory size is roughly equal to the virtual machine memory size minus the specified limit, which means that the free physical memory is included. The swapped memory size is roughly equal to the virtual machine host memory usage minus the specified limit. In the kernel compile virtual machine, since most of the guest physical pages are used to buffer the workload files, the virtual machine's effective host memory usage is close to the virtual machine memory size. Hence, the swapped memory size is similar to the ballooned memory size.

Oracle/Swingbench

Figure 13 presents the throughput of an Oracle database tested by the Swingbench workload with different memory limits when using ballooning versus swapping. The throughput is normalized to the case where virtual machine memory is not reclaimed.

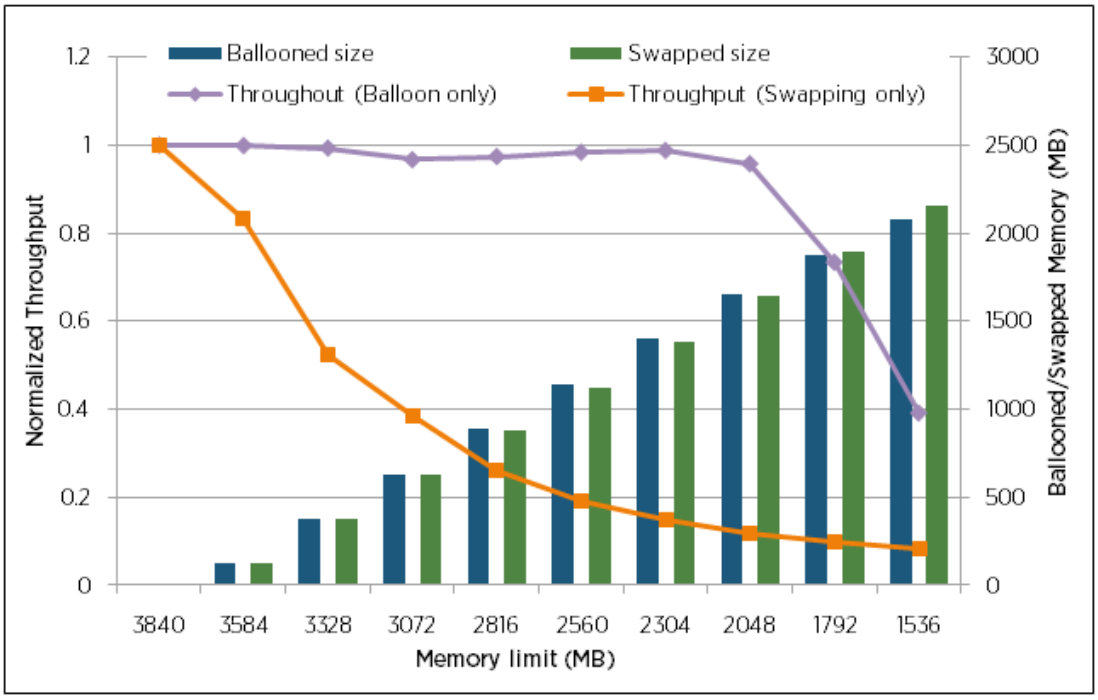

Figure 13. Performance of Swingbench when Using Ballooning vs. Swapping

As shown in the kernel compile test, using ballooning barely impacts the throughput of the Swingbench virtual machine until the memory limit decreases below 2048MB. This occurs when the guest operating system starts to page out the physical pages that are heavily reused by the Oracle database.

In contrast to ballooning, any amount of swapping causes significant throughput penalty. The throughput loss is already 17% when the memory limit is 3584MB. In hypervisor swapping, some guest buffer pages are unnecessarily swapped out and some guest kernel or performance-critical database pages are also unintentionally swapped out because of the random page selection policy. For the Swingbench virtual machine, the virtual machine host memory usage is very close to the virtual machine memory size, so the swapped memory size is very close to the ballooned memory size.

SPECjbb

Figure 14 presents the throughput of the SPECjbb workload with different memory limits when using ballooning versus swapping.

The throughput is normalized to the case where virtual machine memory is not reclaimed.

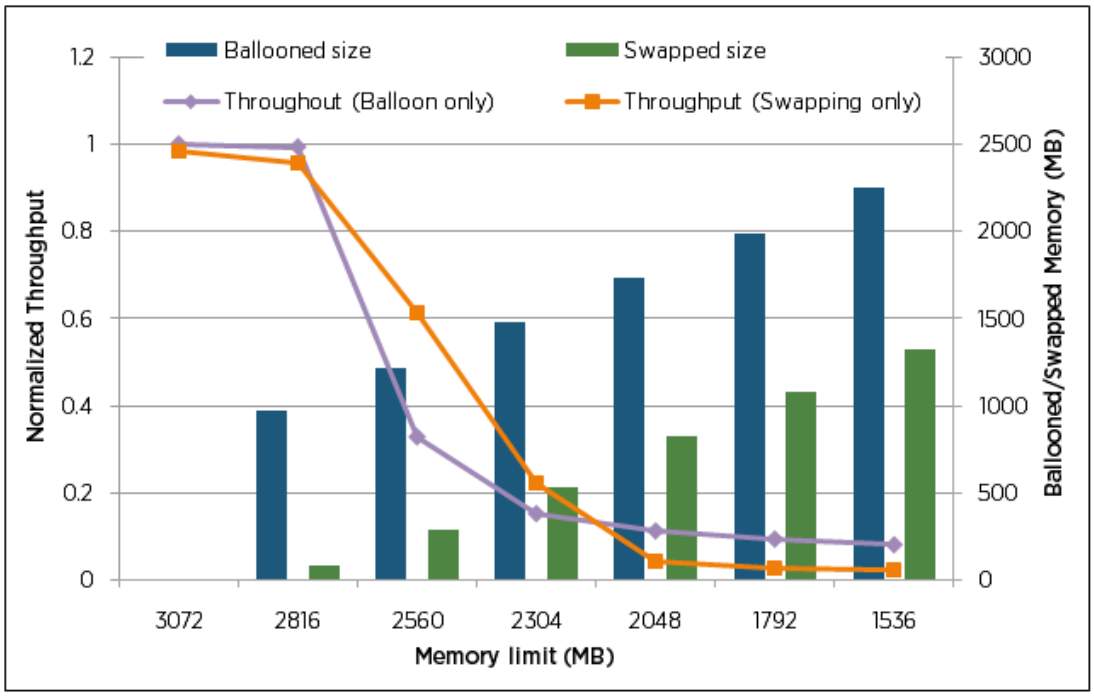

Figure 14. Performance of SPECjbb when Using Ballooning vs. Swapping

This figure shows that when the virtual machine memory limit decreases beyond 2816MB, the throughput of SPECjbb degrades significantly in both ballooning and swapping cases. When the memory limit is reduced to 2048MB, the throughput losses are 89% and 96% for ballooning and swapping respectively. Since the configured JVM heap size is 2.5GB, the actual virtual machine working set size is around 2.5GB plus guest operating system memory usage (about 300MB). When the virtual machine memory limit falls below 2816MB, the host memory cannot back the entire virtual machine's working set, so that virtual machine starts to suffer from guest-level paging in the ballooning cases or hypervisor swapping in the swapping cases.

Since SPECjbb is an extremely memory intensive workload, its throughput is largely determined by how much of the working set resides in host memory. Since both ballooning (which may introduce guest swapping) and host swapping similarly decrease the amount of SPECjbb's working set in memory, both techniques largely hurt SPECjbb performance.

Surprisingly, when the memory limit is 2506MB or 2304MB, swapping yields higher throughput than ballooning does. This seems to be counterintuitive because hypervisor swapping typically causes a higher performance penalty. One reasonable explanation is that the random page selection policy used in hypervisor swapping largely favors the access patterns of the SPECjbb virtual machine. More specifically, with ballooning, when the guest operating system (Linux in this case) pages out guest physical pages to satisfy the balloon driver's allocation request, it chooses the pages using an LRU-approximated[‡‡] policy. However, JVM often scans the allocated guest physical memory (Java heap) during garbage collection. This behavior may fall into a well-known LRU pathological case in which the memory requests misses dramatically even when the allocated memory size is slightly smaller than the working set size. In contrast, when using hypervisor swapping, the swapped physical pages are randomly selected by the hypervisor, which makes the memory request miss increase gradually as the allocated host memory decreases. This is why using swapping achieves higher throughput when the memory limit is smaller than the virtual machine's working set size. However, when the memory limit drops to 2304MB, quite a few virtual machine memory requests miss in both the swapping and ballooning cases. Using swapping starts to cause worse performance compared to using ballooning. Note that the above two configurations where

[‡‡] LRU refers to "least recently used." LRU is a caching policy that determines how pages are discarded when the cache is full and a new page must be accepted into the cache. LRU specifies that the least recently used page is discarded.

swapping outperforms ballooning are rare pathological cases for ballooning performance. In most cases, using ballooning achieves much better performance compared to using swapping.

Since the virtual machine's working set size (approximately 2.8GB) is much smaller than the configured virtual machine memory size (4GB), the ballooned memory size is much higher than the swapped memory size.

Microsoft Exchange Server 2007

This section presents how ballooning and swapping impact the performance of an Exchange Server virtual machine. Exchange Sever is a memory intensive workload that is optimized to use all the available physical memory to cache the transactions for fewer disk I/Os.

The Exchange Server performance was measured using the average Remote Procedure Call (RPC) latency during two hours of stable performance. The RPC latency gauges the server processing time for an RPC from LoadGen (the client application that drives the Exchange server). Therefore, lower RPC latency means better performance. The results are presented in Figure 15.

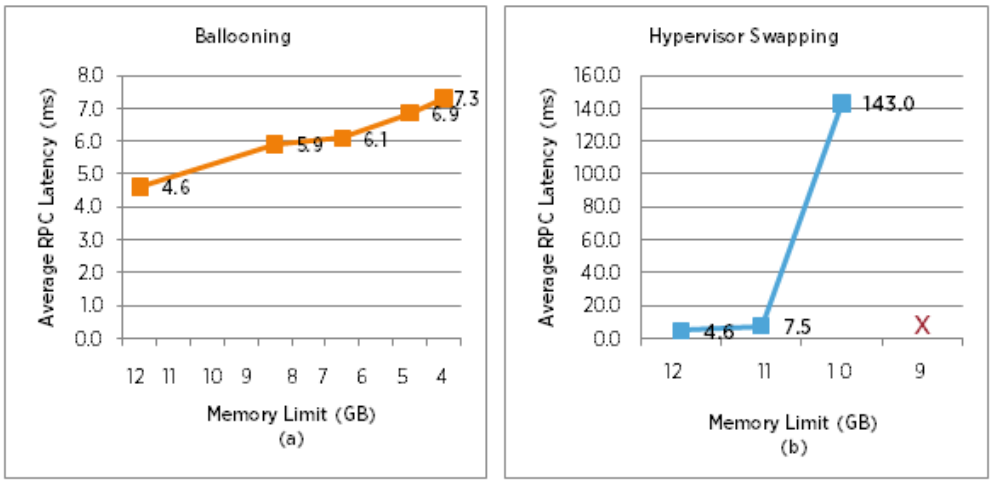

Figure 15. Average RPC Latency of Exchange when Using Ballooning vs. Using Swapping

Figure 15 (a), illustrates that when the memory limit decreases from 12GB to 3GB, the average RPC latency is gradually increased from 4.6ms to 7.3ms with ballooning. However, as shown in Figure 15 (b), the RPC latency is dramatically increased from 4.6ms to 143ms when solely swapping out 2GB host memory. When the memory limit is reduced to 9GB, hypervisor swapping makes the RPC latency too high, which resulted in the failure of the LoadGen application (due to timeout).

Overall, this figure confirms that using ballooning to reclaim memory is much more efficient than using hypervisor swapping for the Exchange Server virtual machine.

Memory Compression Performance

In this experiment, host memory overcommitment is enforced by reducing the host physical memory size instead of setting the VM memory limit. The maximum compression cache size is fixed at 10% of configured guest memory size.

SharePoint

SharePoint Sever is composed of five VMs with a total VM memory size of 32GB. The SQL server VM is given a full 16GB memory reservation because any memory reclamation through ballooning or swapping from this VM will make the SharePoint performance degrade significantly. No reservations are set for the other four VMs. In addition, all the memory reclamation techniques are enabled by default. Figure 16 presents how memory compression helps improve the SharePoint performance when host memory size is reduced. The performance result is normalized to the 36GB case where the host memory is undercommitted.

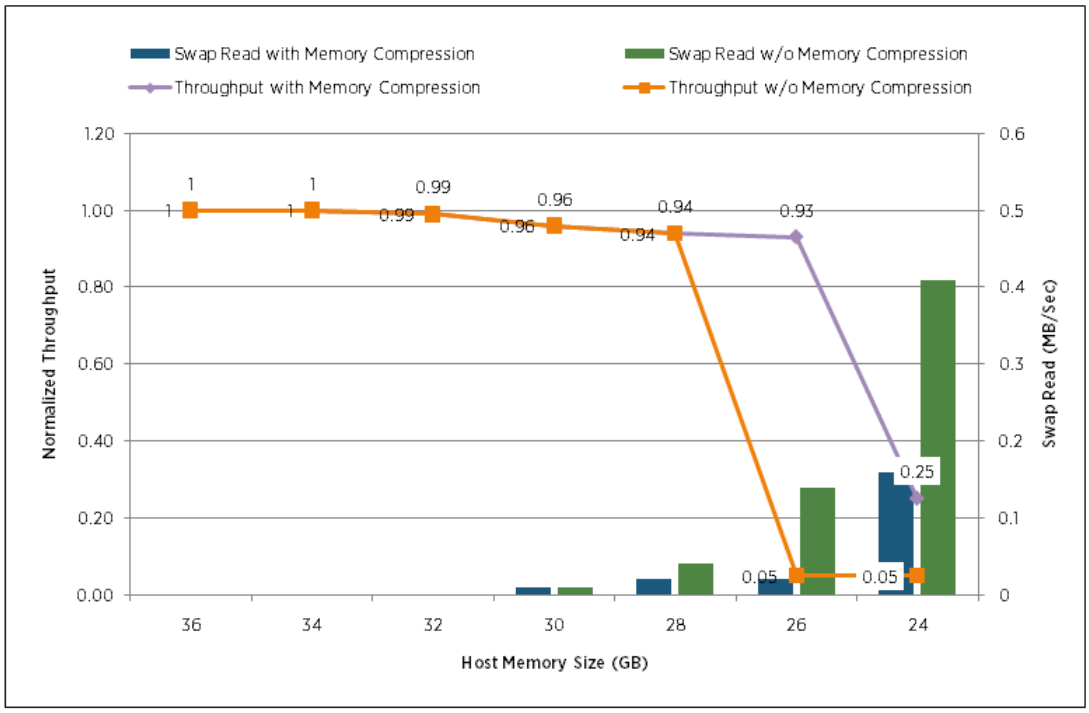

Figure 16. Throughput and Swap Read Rate of SharePoint with Different Host Memory Sizes

As shown in Figure 16, when host memory size is reduced from 36GB to 28GB, there is only a 6% performance degradation. This is because page sharing and ballooning efficiently reclaim most of the VM memory with little performance penalty. There are a few swapping activities when host memory is reduced to 28GB and 26GB, but they do not impact performance much.

However, when host memory is reduced to 26GB, ballooning and page sharing are not enough to reclaim host memory. ESXi has to swap out guest pages from VMs. Without memory compression, the performance loss is 95% because of the severe penalty of host swapping. Interestingly, by using memory compression, the performance loss is brought back from 95% to 6%. Such a huge improvement is mainly due to the significant reduction in the amount of swapped-in pages. As we can see, applying memory compression reduces the swap in (swap read) rate by around 85% since most of the missing pages can be found in the compression caches.

When host memory size is reduced to 24GB, even with memory compression, the throughput drops to 25% of the throughput in the 36GB case because the amount of swapped-in pages is considerably increased. Compression cache statistics showed that the compression caches of those VMs reached their maximum limit size in this case.

Swingbench

In this experiment, 16 Swingbench VMs were used to overcommit the host memory. None of the VMs had any memory reservation. The total VM configured memory size was 80GB. Figure 17 shows the normalized overall throughput of Swingbench with memory compression versus without memory compression when host memory size is reduced from 96GB to 50GB. The performance result is normalized to the 96GB case. Each of the 16 VMs had the same copy of the database installed, which led to artificial page sharing and made the impact of memory compression less obvious. So for benchmarking purposes, page sharing was temporarily turned off in this experiment.

From Figure 17, we can see that when host memory is reduced from 96GB to 70GB, the performance loss of Swingbench is less than 5% since ballooning efficiently reclaims the guest pages from the VMs without causing severe guest paging. However, when host memory is reduced to 60GB, the memory pressure is high enough to make the host start to swap out guest pages. Using memory compression can easily recover 14% of the lost throughput. Again, as shown in this figure, the improvement is mainly due to the significant reduction in host page swap-in (swap-read) operations. For the same reason, memory compression can bring the normalized throughput from 42% to 70% when host memory is only 50GB. In this case, even with memory compression, there still were considerable swap-in operations. Close observation found that although the compression caches of those VMs were not full, there were quite a few swapped-out pages that failed on compression because they could not be compressed into half or quarter pages. Hence, memory compression cannot completely eliminate host swapping.

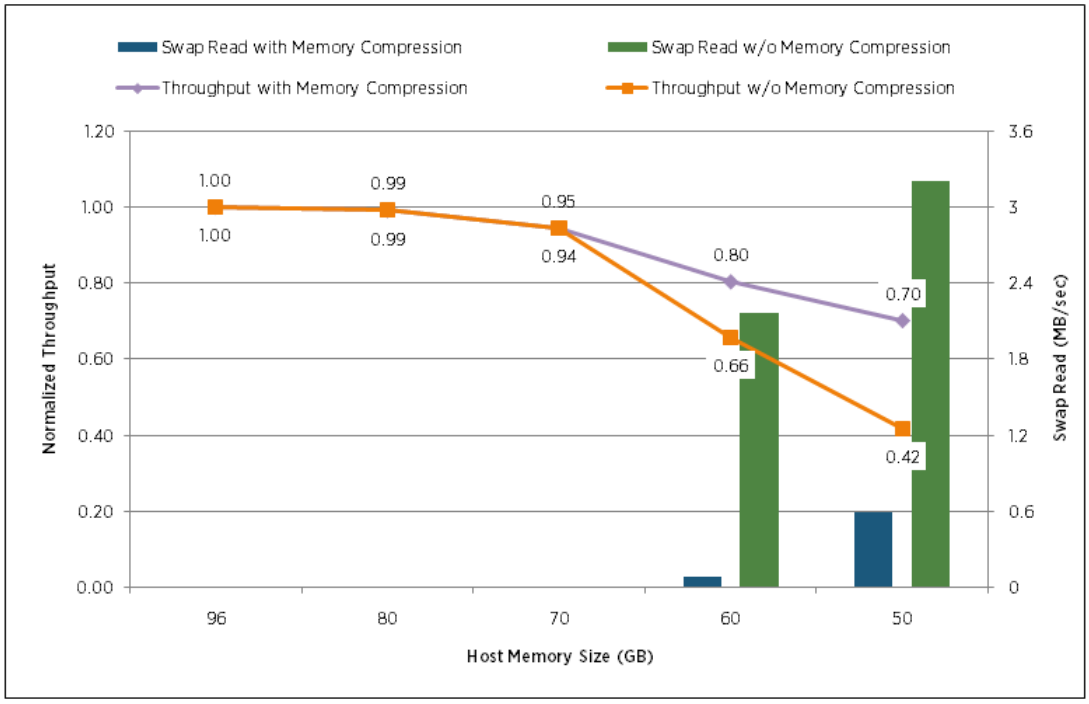

Figure 17. Throughput and Swap Read Rate of Swingbench with Different Host Memory Sizes

Memory compression itself introduces some CPU overhead when performing compression or decompression. This experiment shows that the physical CPU utilization is slightly increased by 1-2% when memory compression is enabled.

Swap to SSD Performance

In this experiment, 12 Swingbench VMs were used to overcommit the host memory. None of the VMs had any memory reservation. The total memory size of all the VMs was 60GB. Figure 18 shows the normalized overall throughput and the average fault-in[§§] latency of the Swingbench workload with a 40GB host cache configured in SSD versus without SSD installed when the host memory size is reduced from 72GB to 20GB. The throughput results are normalized to the 72GB case. All other memory reclamation techniques were enabled by default.

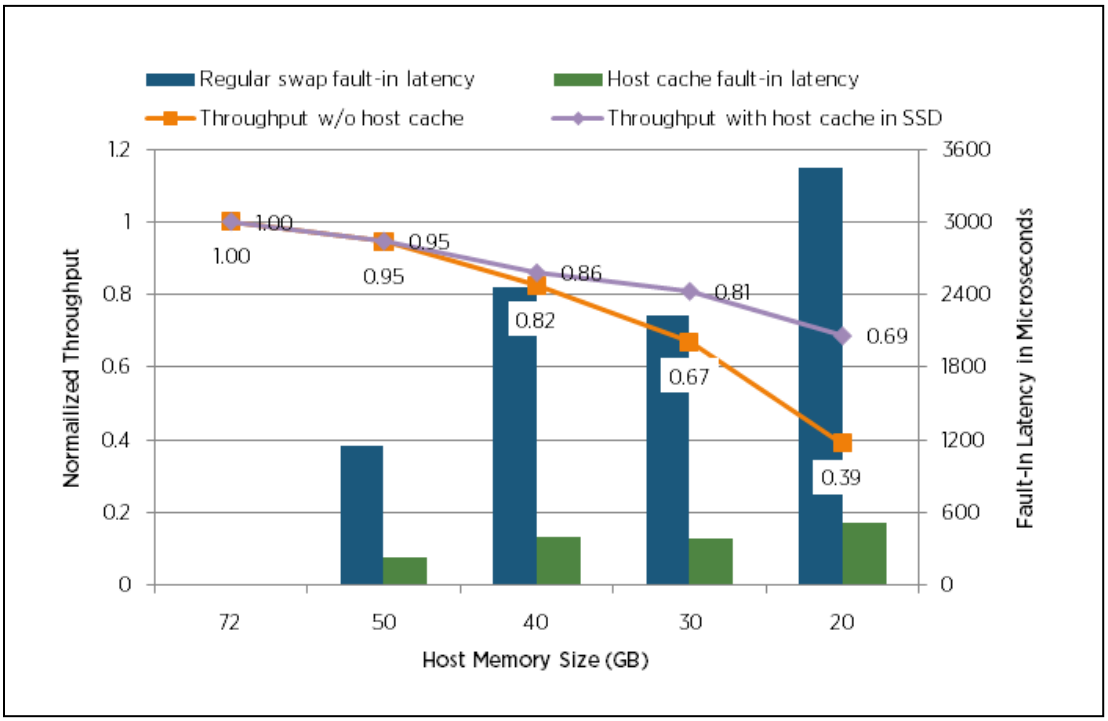

Figure 18. Throughput and Average Fault-In Latency of Swingbench with Different Host Memory Sizes

As shown in Figure 18, when host memory is above 40GB, other reclamation techniques—that is, page sharing, ballooning and memory compression—can efficiently reclaim memory with a small performance impact. When host memory is reduced to 30GB, the amount of memory that needs to be swapped is considerably high. The performance lost is 33% compared to the case where memory is undercommitted. However, by using the host cache in SSD, approximately 14% of the lost throughput is recovered. The improvement occurs because the average fault-in latency is significantly reduced from 2000 plus microseconds to a few hundred microseconds when storing the swapped-out pages in host cache. As the host memory is further reduced, more pages are stored in SSD and the benefit of using SSD becomes more significant. For example, when host memory is 20GB (almost three times memory overcommitment), swapping to host cache recovers approximately 30% of the lost performance.

[§§] The fault-in latency is the time it takes to swap a block of memory back in from disk.

Best Practices

Based on the memory management concepts and performance evaluation results presented in the previous sections, the following are some best practices for host and guest memory usage.

- Do not disable page sharing or the balloon driver. Page sharing is a lightweight technique which opportunistically reclaims redundant host memory with trivial performance impact. In the cases where hosts are heavily overcommitted, using ballooning is generally more efficient and safer than using hypervisor swapping, based on the results presented in "Ballooning vs. Host Swapping" on page 19. These two techniques are enabled by default and should not be disabled unless application testing shows that the benefits of doing so clearly outweigh the costs.
- Carefully specify memory limits and reservations. The virtual machine memory allocation target is subject to the VM's memory limit and reservation. If these two parameters are misconfigured, users may observe ballooning or swapping even when the host has plenty of free memory. For example, a virtual machine's memory may be reclaimed when the specified limit is too small or when other virtual machines reserve too much host memory, even though they may only use a small portion of the reserved memory. If a performance-critical virtual machine needs a guaranteed memory allocation, the reservation needs to be specified carefully because it may impact other virtual machines.
- Host memory size should be larger than guest memory usage. For example, it is unwise to run a virtual machine with a 2GB working set size in a host with only 1GB of host memory. If this is the case, the hypervisor has to reclaim the virtual machine's active memory through ballooning or hypervisor swapping, which will lead to potentially serious virtual machine performance degradation. Although it is difficult to tell whether the host memory is large enough to hold all of the virtual machines' working sets, the bottom line is that the host memory should not be excessively overcommitted because this state makes the guests continuously page out guest physical memory.
- Use shares to adjust relative priorities when memory is overcommitted. If the host's memory is overcommitted and the virtual machine's allocated host memory is too small to achieve a reasonable performance, adjust the virtual machine's shares to escalate the relative priority of the virtual machine so that the hypervisor will allocate more host memory for that virtual machine.
- Set an appropriate virtual machine memory size. The virtual machine memory size should be slightly larger than the average guest memory usage. The extra memory will accommodate workload spikes in the virtual machine. Note that the guest operating system only recognizes the specified virtual machine memory size. If the virtual machine memory size is too small, guest-level paging is inevitable, even though the host might have plenty of free memory. If the virtual machine memory size is set to a very large value, virtual machine performance will be fine, but more virtual machine memory means that more overhead memory needs to be reserved for the virtual machine.

Conclusion

vSphere 5.0 leverages the advantages of several memory reclamation techniques to allow users to overcommit host memory, With the support of memory overcommitment, vSphere 5.0 can achieve a remarkable VM consolidation ratio while maintaining reasonable performance. This paper presents how ESXi in vSphere 5.0 manages the host memory and how the memory reclamation techniques efficiently reclaim host memory without much impact on VM performance. A share-based memory management mechanism is used to enable both performance isolation and efficient memory utilization. The algorithm relies on a working set estimation technique which measures the idleness of a VM. Ballooning reclaims memory from a VM by implicitly causing the guest OS to invoke its own memory management technique. Transparent page sharing exploits sharing opportunities within and between VMs without any guest OS involvement. Memory compression reduces the amount of host swapped pages by storing the compressed format of the pages in a per-VM memory compression cache. Swap to SSD leverages SSD's low read latency to alleviate the host swapping penalty. Finally, a high level dynamic memory reallocation policy coordinates all these techniques to efficiently support memory overcommitment.

References

1. Waldspurger, Carl A. *Memory Resource Management in VMware ESX Server.* Proceeding of the Fifth Symposium on Operating System Design and Implementation, 2002. http://www.waldspurger.org/carl/papers/esx-mem-osdi02.pdf.

2. *vSphere Resource Management,* vSphere 5 Documentation Center. VMware, Inc., 2011. http://pubs.vmware.com/vsphere-50/topic/com.vmware.vsphere.resmgmt.doc_50/GUID-98BD5A8A-260A-494F-BAAE-74781F5C4B87.html.

3. *Memory Performance Chart Statistics in the vSphere Client.* VMware Community, 2009. http://communities.vmware.com/docs/DOC-10398.

4. *Understanding VirtualCenter Performance Statistics.* VMware Community, 2011. http://communities.vmware.com/docs/DOC-5230.

5. Bhatia, Nikhil. *Performance Evaluation of Intel EPT Hardware Assist.* VMware, Inc., 2009. http://www.vmware.com/resources/techresources/10006.

6. Bhatia, Nikhil. *Performance Evaluation of AMD RVI Hardware Assist.* VMware, Inc., 2009. http://www.vmware.com/resources/techresources/1079.

7. Wirzenius, Lars, Joanna Oja, Stephen Stafford, and Alex Weeks. "The buffer cache," *Linux System Administrator's Guide*, Version 0.9. The Linux Documentation Project. http://tldp.org/LDP/sag/html/buffer-cache.html.

About the Author

Fei Guo works at VMware in the Performance Engineering team where he specializes in memory resource management in vSphere. He has presented at VMworld and written several blogs on this subject.

Acknowledgements

The author thanks all members in the VMware performance team who contributed to and reviewed this paper. He also thanks the vSphere memory management developers for providing technical background for this paper.

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com