Understanding Memory Resource Management in VMware® ESX™ Server

vmware®

Table of Contents

1. Introduction

VMware® ESX™ 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 ESX, 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. More details can be found in "Memory Resource Management in VMware ESX Server" [1].

ESX 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 ESX architecture will help you understand the concepts presented in this paper.

In order to quickly monitor virtual machine memory usage, the VMware vSphere™ Client exposes two memory statistics in the resource summary: **Consumed Host Memory** and **Active Guest Memory**.

Figure 1: Host and Guest Memory usage in vSphere Client
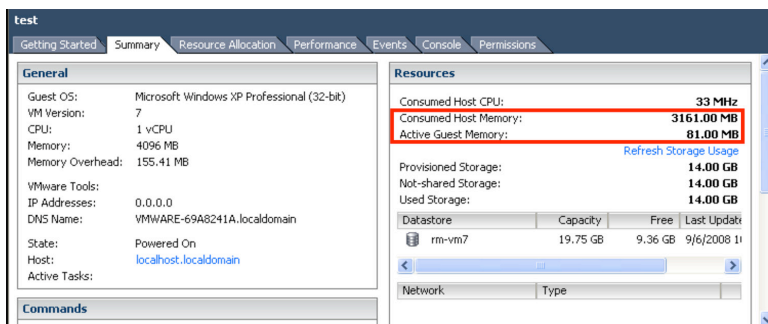
Consumed Host Memory usage is defined as the amount of host memory that is allocated to the virtual machine, Active Guest Memory is defined as the amount of guest memory that is currently being used by the guest operating system and its applications. These two statistics are quite useful for analyzing the memory status of the virtual machine and providing hints to address potential performance issues.

This paper helps answer these questions:

• Why is the Consumed Host Memory so high?

• Why is the Consumed Host Memory usage sometimes much larger than the Active Guest Memory?

• Why is the Active Guest 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 ESX. Understanding how ESX manages memory will also make the performance implications of changing ESX memory management parameters clearer.

The vSphere Client can also display performance charts for the following memory statistics: active, shared, consumed, granted, overhead, balloon, swapped, swapped in rate, and swapped-out rate. 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].

The rest of the paper is organized as follows. Section 2 presents the overview of ESX memory management concepts. Section 3 discusses the memory reclamation techniques used in ESX. Section 4 describes how ESX allocates host memory to virtual machines when the host is under memory pressure. Section 5 presents and discusses the performance results for different memory reclamation techniques. Finally, Section 6 discusses the best practices with respect to host and guest memory usage.

2. ESX Memory Management Overview

2.1 Terminology

The following terminology is used throughout this paper.

- *Host physical memory*[1] 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.

2.2 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.

When running a virtual machine, the hypervisor creates a contiguous addressable memory space for the virtual machine. This memory space has the same properties as the virtual address space presented to the applications by the guest operating system. 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 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 ESX: guest virtual memory, guest physical memory, and host physical memory. Their relationships are illustrated in Figure 2 (a).

Figure 2: Virtual memory levels (a) and memory address translation (b) in ESX
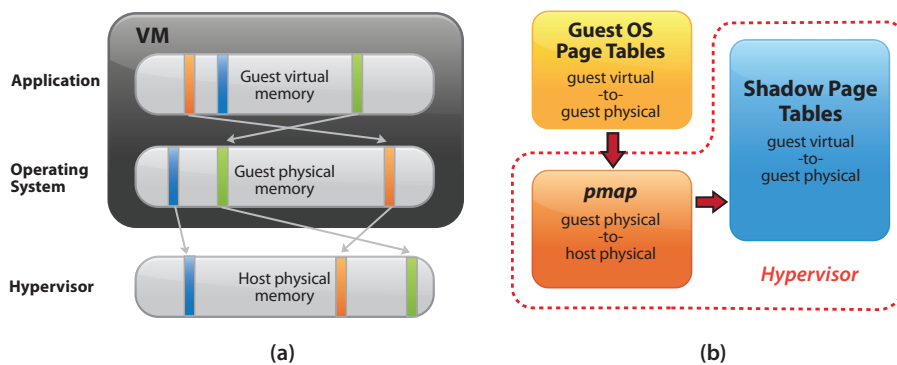
As shown in Figure 2 (b), in ESX, the address translation between guest physical memory and host physical memory is maintained by the hypervisor using a physical memory mapping data structure, or *pmap*, for each virtual machine. The hypervisor intercepts all virtual machine instructions that manipulate 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

---

[1] 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 [1].

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, ESX 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, some new generation CPUs, such as 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. Hardware support 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]

2.3 Memory Management Basics in ESX

Prior to talking about how ESX 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 deallocate the virtual memory during the execution.

• In a non-virtual 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 interfaces. 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 an operating system, such that 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 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 host physical memory for this virtual machine any more. This means that the following always holds true:

VM's host memory usage <= VM's guest memory size + VM's overhead memory

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 the vSphere Resource Management Guide [2].
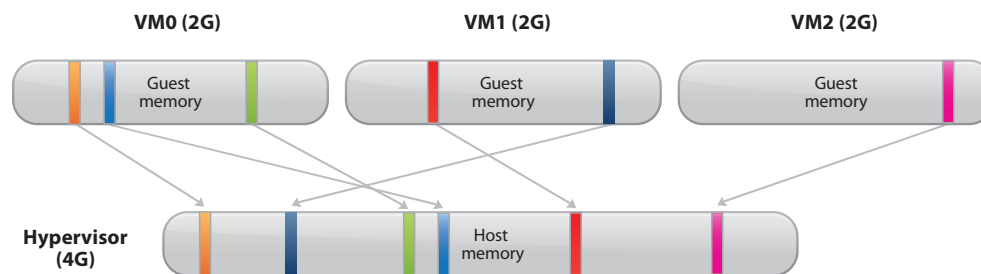
3. Memory Reclamation in ESX

3.1 Motivation

According to the above equation if the hypervisor cannot reclaim host physical memory upon virtual machine memory deallocation, it must reserve enough host physical memory to back all virtual machine's 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. ESX supports memory overcommitment from the very first version, due to two important benefits it provides:

• Higher memory utilization: With memory overcommitment, ESX 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 ESX.*

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

3.2 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 only keep 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 ESX, 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. ESX 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, ESX uses hashing to identify potentially identical pages. The detailed algorithm is illustrated in Figure 4.

*Figure 4: Content based page sharing in ESX*
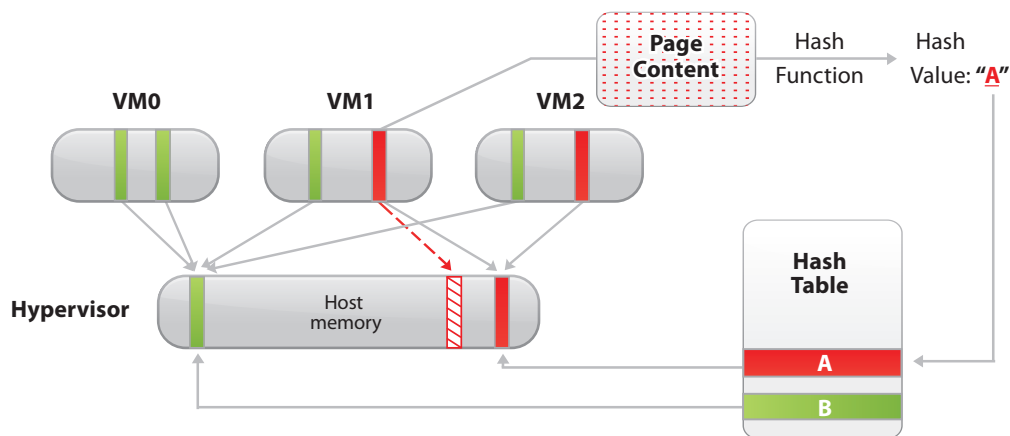
A hash value is generated based on the candidate guest physical page's content. The hash value is then used as a key to look up a global hash table, in which each entry records a hash value and the physical page number of a shared page. If the hash value of the candidate guest physical page matches an 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 sytem. 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 to this private copy the virtual machines affecting the guest physical page. 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 ESX, 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 ESX advanced settings. An example is shown in Figure 5.

Figure 5: Configure page sharing in vSphere Client
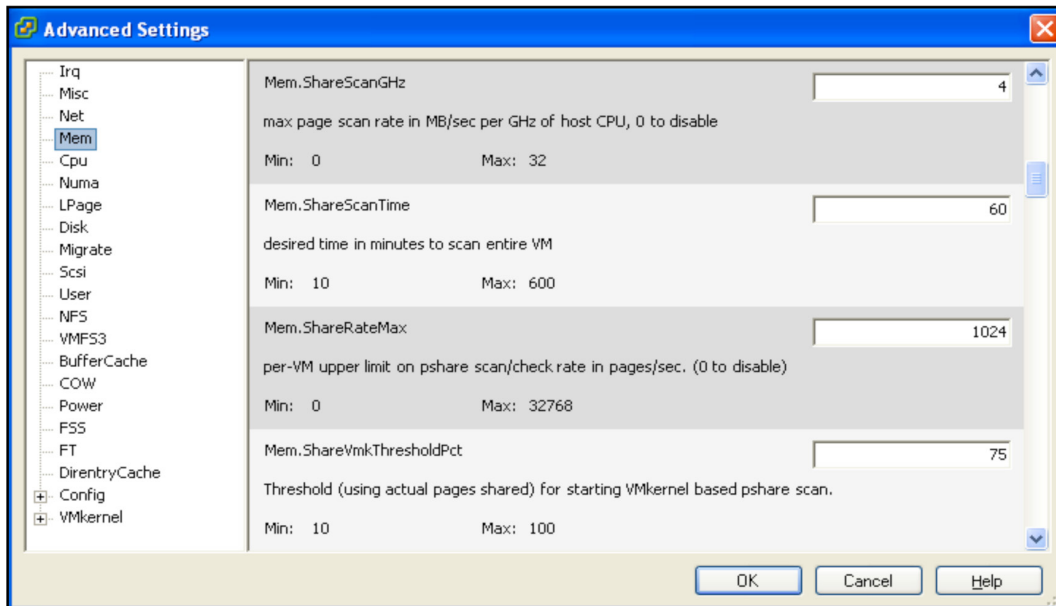
The default values of these three parameters are carefully chosen to provide sufficient sharing opportunities while keeping the CPU overhead negligible. In fact, ESX intelligently adjusts the page scan rate based on the amount of current shared pages. If the virtual machine's page sharing opportunity seems to be low, the page scan rate will be reduced accordingly and vice versa. This optimization further mitigates the overhead of page sharing.

3.3 Ballooning

Ballooning is a completely different memory reclamation technique compared to 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 ESX, 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 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 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 normal virtual machine memory allocation and allocate a new host physical page for the virtual machine. When the hypervisor decides to deflate 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 ESX
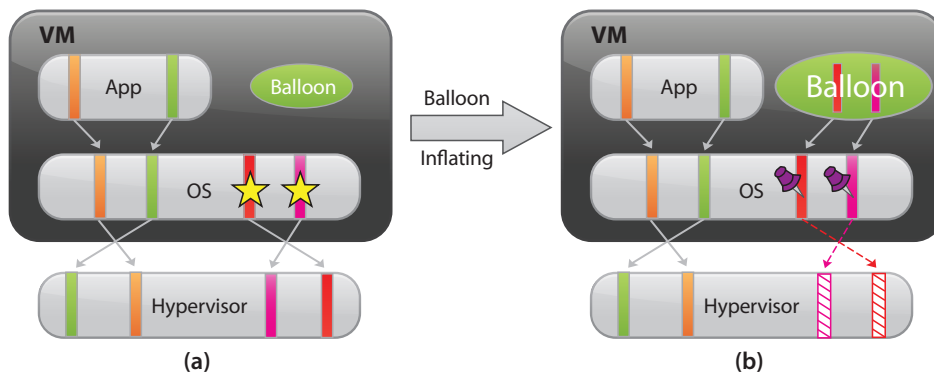
Typically, the hypervisor inflates the virtual machine balloon when it is under memory pressure. By inflating the balloon, a virtual machine consumes less physical memory on the host, but more physical memory inside the guest. As a result, the hypervisor offloads some of its memory overload to the guest 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. 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.

3.4 Hypervisor Swapping

As a last effort to manage excessively overcommitted physical memory, the hypervisor will swap the virtual machine's memory. Transparent page sharing has very little impact to performance and, as stated earlier, ballooning will only induce guest paging if the guest operating system is short of memory.

In the cases where ballooning and page sharing are not sufficient to reclaim memory, ESX employs hypervisor swapping to reclaim memory. To support this, when starting a virtual machine, 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.

Besides the limitation on the reclaimed memory size, 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 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. For example, the guest operating system will never page out its kernel pages since those pages are critical to ensure guest kernel performance. The hypervisor, however, cannot identify those guest kernel pages, so it may swap them out. In addition, the guest operating system reclaims the clean buffer pages by dropping them [7]. Again, since the hypervisor cannot identify the clean guest buffer pages, it will unnecessarily swap them out to the hypervisor swap device in order to reclaim the mapped host physical memory.

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. Note that it is impossible to find an algorithm to handle all these pathological cases properly. ESX attempts to mitigate the impact of interacting with guest operating system memory management by randomly selecting the swapped guest physical pages. Due to the potential high performance penalty, hypervisor swapping is the last resort to reclaim memory from a virtual machine.

3.5 When to Reclaim Host Memory[2]

ESX maintains four host free memory states: *high*, *soft*, *hard*, and *low*, which are reflected by four thresholds: 6 percent, 4 percent, 2 percent, and 1 percent of host memory respectively. Figure 7 shows how the host free memory state is reported in esxtop.

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

Figure 7: Host free memory state in esxtop
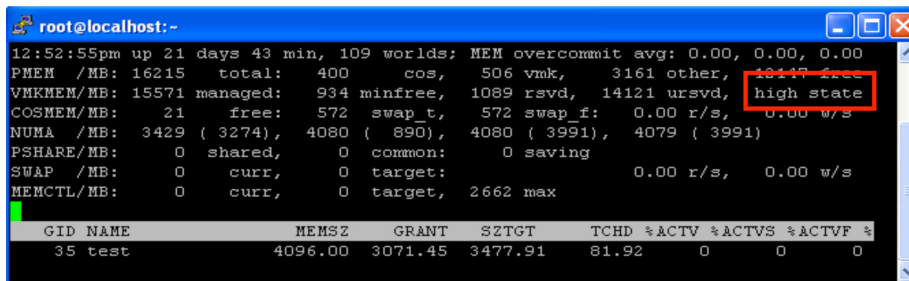
In the *high* state, the aggregate virtual machine guest memory usage is smaller than the host memory size. Whether or not host memory is overcommitted, the hypervisor will not reclaim memory through ballooning or swapping. (This is true only when the virtual machine memory limit is not set.)

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. Through swapping, the hypervisor should be able to quickly reclaim memory and bring the host memory state back to the *soft* state.

---

[2] *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 out of the topic of this paper. Most of the details can be found in the "Managing Resource Pools" section of the vSphere Resource Management Guide [2].*

In a rare case where host free memory drops below the *low* threshold, the hypervisor continues to reclaim memory through swapping, and additionally blocks the execution of all virtual machiens 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 to reclaim memory from the virtual machine until the virtual machines host memory usage falls back to its specified limit.

4. ESX Memory Allocation Management for Multiple Virtual Machines

This section describes how ESX allocates host memory to multiple virtual machines, especially when the host memory is overcommitted. ESX employs a share-based allocation algorithm to achieve efficient memory utilization while guaranteeing the performance isolation of memory. [1]

ESX 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 8.

Figure 8: Configure virtual machine memory allocation
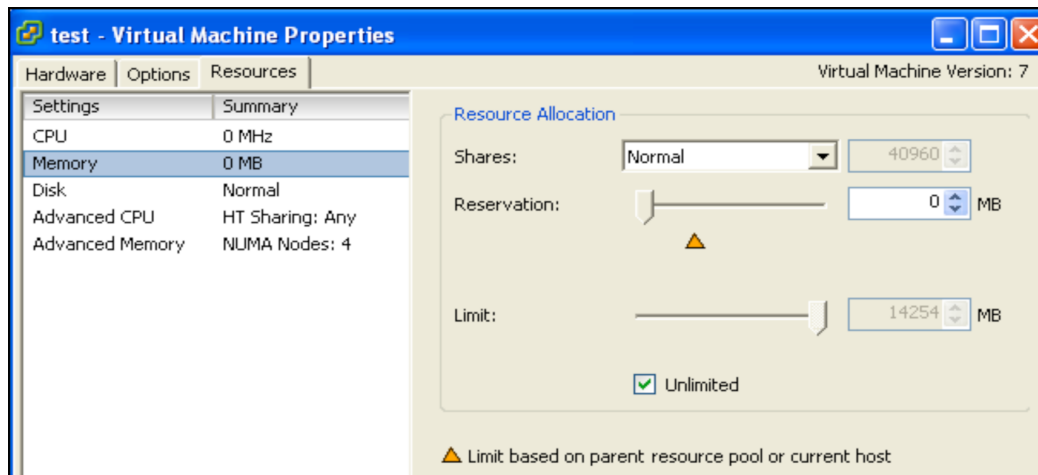
Limit is the upper bound of the amount of host physical memory allocated for a virtual machine. By default, limit is set to unlimited, which means a virtual machine's maximum allocated host physical memory is its specified virtual machine memory size (according to equation 1). Reservation is a guaranteed lower bound on the amount of host physical memory that 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, ESX 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[3]. 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 }

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, ESX 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 the lack of memory.

ESX 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]. The effectiveness of this algorithm relies on the accurate estimation of the virtual machine's working set size. ESX 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 also be found in *Memory Resource Management in VMware ESX Server* [1]. By default, ESX samples 100 guest physical pages for each 60-second period. The sampling rate can be adjusted by changing Mem.SamplePeriod in ESX advanced settings.

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

---

[3] *If a virtual machine is in a resource pool, the resource pool configuration is also taken into account when calculating the memory allocation target for the virtual machine. A detailed explanation of the resource pool is out of the topic of this paper. Most of the details can be found in the "Managing Resource Pools" section in the vSphere Resource Management Guide [2].*

5. Performance Evaluation

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

5.1 Experimental Environment

ESX RC build on a Dell PowerEdge 6950 system was installed and experiments were conducted against four workloads. The system hardware configurations and workload descriptions are summarized in Table 1 and Table 2 respectively.

Table 1. Server configurations

Processors	Type:	4 socket dual-core AMD Opteron 8222SE processors
	Frequency:	3GHz
	Last level cache:	1MB private L2 cache per core
	FSB:	1GHz
BIOS	Version:	1.1.2
	Virtualization technology:	Enabled
	Demand-based power management:	Disabled
DDR2 Memory	Size:	64G
	Speed:	667MHz
SAN Storage	LUN size:	1TB
	Protocol:	Fibre Channel

Table 2. Workload descriptions

SPECjbb2005	Heap size:	2.5GB
	Number of warehouse:	1
	Runtime:	10 minutes
	Number of runs:	3
	VM configuration:	1 vCPU, 4GB memory
Kernel Compile	Linux kernel version:	2.6.17
	Command:	"make –j 1 bzImage > /dev/null"
	Number of runs:	4
	VM configuration:	1 vCPU, 512MB memory
Swingbench	Database:	Oracle 11g
	Functional benchmark:	Order Entry
	Number of users:	30
	Runtime:	20 minutes
	Number of runs:	3
	VM configuration:	4 vCPUs, 4G memory
Exchange 2007	Server:	Microsoft Exchange 2007
	Loadgen client:	2000 heavy exchange users
	VM configuration:	4 vCPUs, 12G memory

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 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. In addition, for Swingbench and Exchange, the client applications were installed in a separate native machine.

5.2 Transparent Page Sharing Performance

In this experiment, two instances of workloads were run. The overall performance of workloads with page sharing enabled to those with page sharing disabled were compared. There was a focus 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**, 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 9: Performance impact of transparent page sharing

Figure 9 confirms that enabling page sharing introduces a negligible performance overhead in the default setting and only <1 percent 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 CoW pages is another source of page sharing. 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.

5.3 Ballooning vs. Swapping

In the following experiments, VM memory reclamation was enforced by changing each virtual machine's memory limit value from the default 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, ESX only uses ballooning to reclaim memory. The balloon driver was 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.

5.3.1 Linux Kernel Compile

Figure 10 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 10: Performance of kernel compile when using the ballooning vs. the swapping
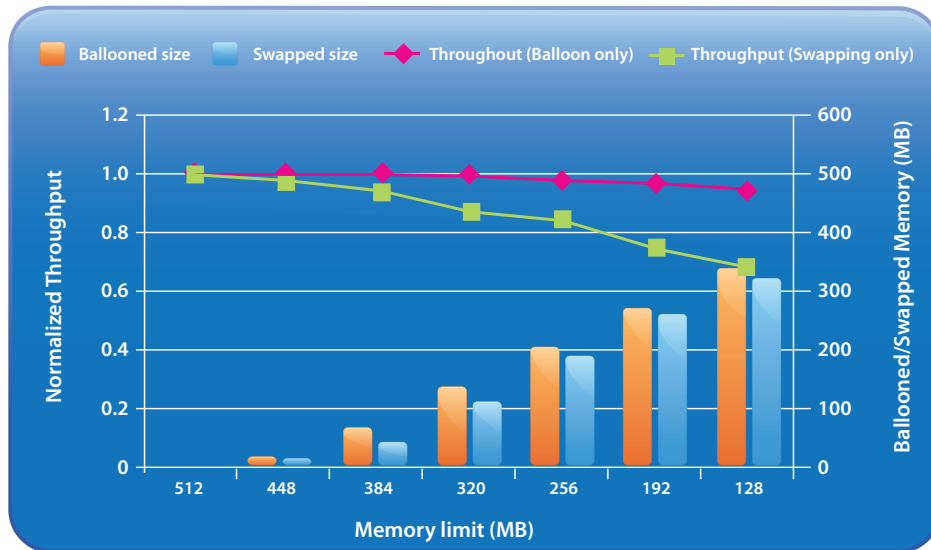
By using ballooning, the kernel compile virtual machine only suffers from 3 percent throughput loss even when the memory limit is as low as 128MB (1/4 of the configured virtual machine memory size). This is because the kernel compile workload has very little memory reuse and most of the guest physical 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 memory limit decreases. When the memory limit is 128MB, the throughput loss is about 34 percent in the swapping case. Balloon inflation is a better approach to memory reclamation from a performance perspective.

Figure 10, illustrates that as 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.

5.3.2 Oracle/Swingbench

Figure 11 presents the throughput of an Oracle database tested by the Swingbench workload with different memory limits when using ballooning vs. swapping. The throughput is normalized to the case where virtual machine memory is not reclaimed.

*Figure 11: Performance of Swingbench when using ballooning vs. swapping*
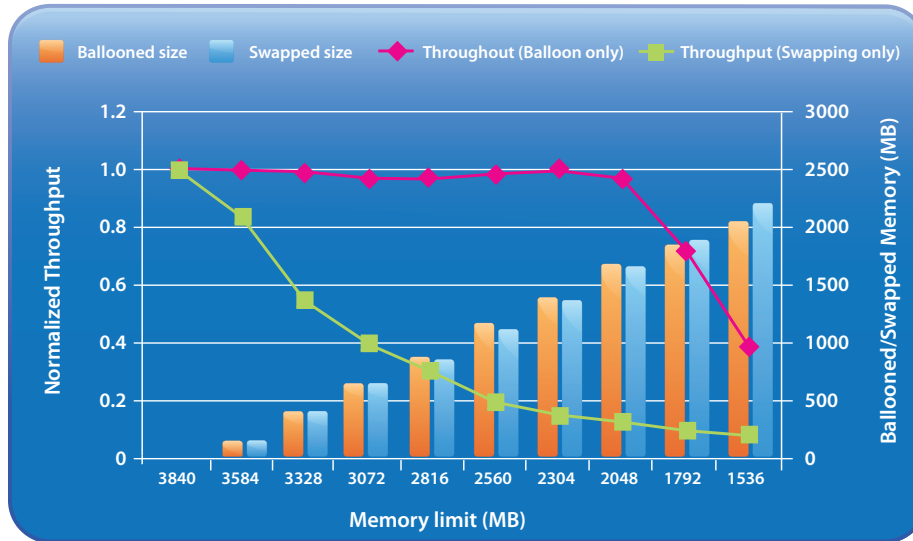
As in the kernel compile test, using ballooning barely impacts the throughput of the Swingbench virtual machine until the memory limit decreases below 2048MB. This occurs when the guest operating system starts to page out the physical pages that are heavily reused by the Oracle database.

In contrast to using ballooning, using swapping causes significant throughput penalty. The throughput loss is already 17 percent 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.

5.3.3 SPECjbb

Figure 12 presents the throughput of the SPECjbb workload with different memory limits when using ballooning vs. swapping. The throughput is normalized to the case where virtual machine memory is not reclaimed.

Figure 12: Performance of SPECjbb when using the ballooning vs. the swapping
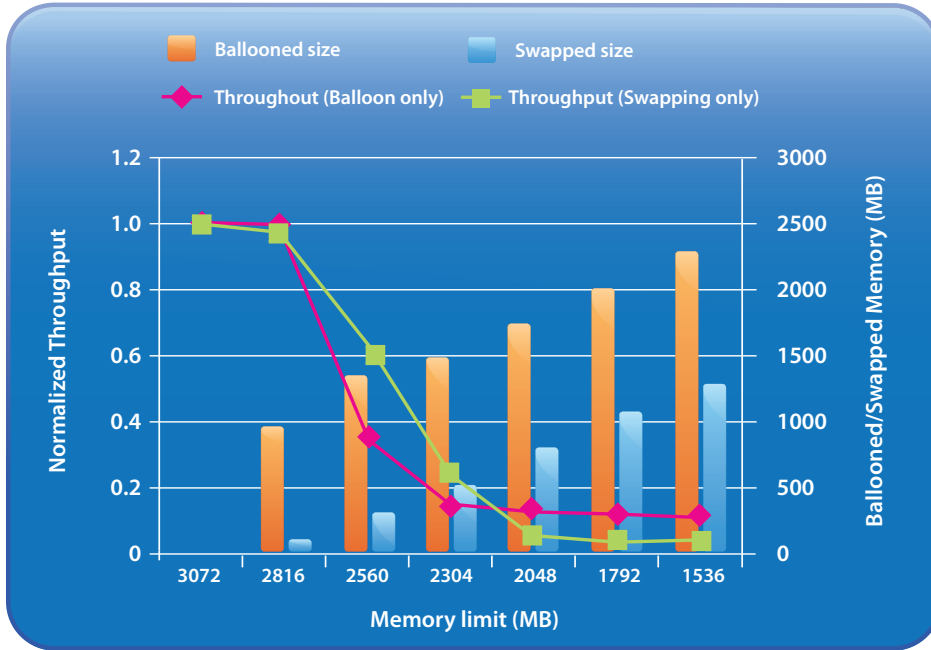
From this figure, it is seen that when the virtual machine memory limit decreases beyond 2816MB, the throughput of SPECjbb degrades significantly in both ballooning and swapping cases. When the memory limit is reduced to 2048MB, the throughput losses are 89 percent and 96 percent for ballooning and swapping 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 the virtual machine memory hit rate. In this instance, virtual machine memory hit rate is defined as the percentage of guest memory accesses that result in host physical memory hits. A higher memory hit rate means higher throughput for the SPECjbb workload. Since ballooning and host swapping similarly decrease memory hit rate, both guest level paging and hypervisor swapping largely hurt SPECjbb performance.

Surprisingly, when the memory limit is 2506MB or 2304MB, using swapping obtains higher throughput than that of using ballooning. 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 targets using an LRU-approximated policy. However, the SPECjbb workload often traverses all the allocated guest physical memory iteratively. For example, the JVM garbage collector periodically scans the entire heap to free memory. This behavior is categorized to a well-known LRU pathological case in which the memory hit rate drops dramatically even when the 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 hit rate reduce more gradually as the memory limit decreases. That 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, the virtual machine memory hit rate is equivalently low in both swapping and ballooning cases. Using swapping starts to cause worse performance compared to using

ballooning. Note that the above two configurations where swapping outperforms ballooning are rare pathological cases for ballooning performance. In most cases, using ballooning achieves much better performance compared to using swapping.

Since SPECjbb virtual machine's working set size (~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.

5.3.4 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 stable run. 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 13.

Figure 13: Average RPC latency of Exchange when using ballooning vs. using swapping
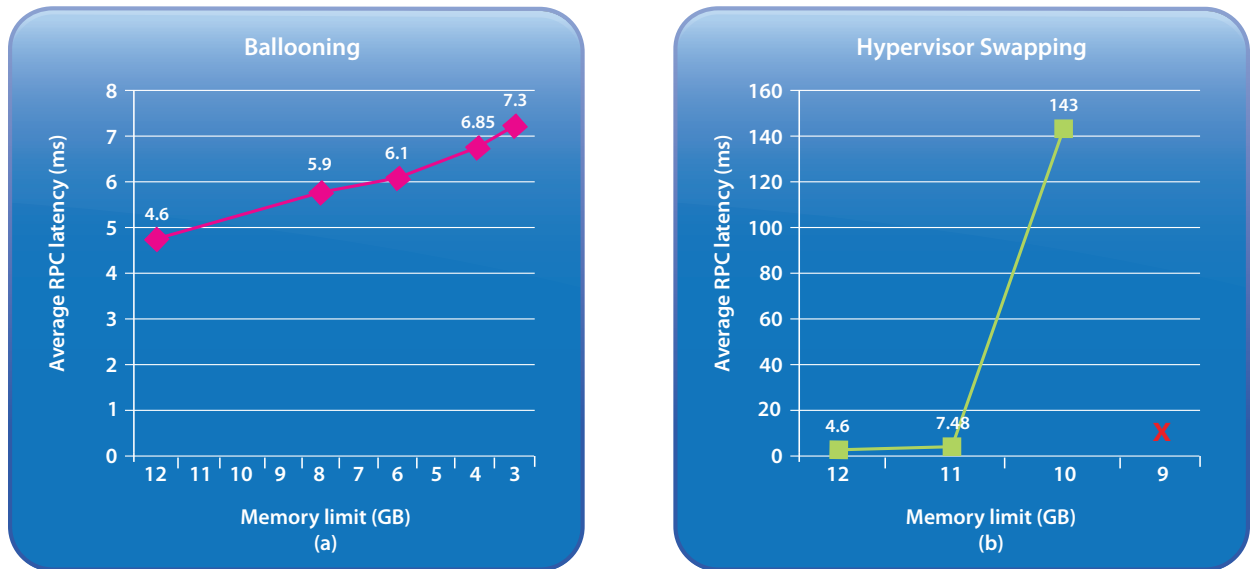

Figure 13 (a), illustrates that when the memory limit decreases from 12GB to 3GB, the average RPC latency is gradually increased from 4.6ms to 7.3ms with ballooning. However, as shown in Figure 13 (b), the RPC latency is dramatically increased from 4.6ms to 143ms when solely swapping out 2GB host memory. 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.

6. 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**. As described, page sharing is a lightweight technique which opportunistically reclaims the redundant host memory with trivial performance impact. In the cases where hosts are heavily overcommitted, using ballooning is generally more efficient and safer compared to using hypervisor swapping, based on the results presented in Section 5.3. These two techniques are enabled by default in ESX[4] and should not be disabled unless the benefits of doing so clearly outweigh the costs.

- **Carefully specify the memory limit and memory reservation**. The virtual machine memory allocation target is subject to the 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 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 machine's working sets, the bottom line is that the host memory should not be excessively overcommitted making the guests have to 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, the user can 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 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 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 may have plenty of free memory. Instead, the user may conservatively set a very large virtual machine memory size, which is fine in terms of virtual machine performance, but more virtual machine memory means that more overhead memory needs to be reserved for the virtual machine.

7. References

[1] Carl A. Waldspurger. "Memory Resource Management in VMware ESX Server". Proceeding of the fifth Symposium on Operating System Design and Implementation, Boston, Dec 2002.

[2] vSphere Resource Management Guide. VMware. http://www.vmware.com/pdf/vsphere4/r40/vsp_40_upgrade_guide.pdf

[3] Memory Performance Chart Statistics in the vSphere Client. http://communities.vmware.com/docs/DOC-10398

[4] VirtualCenter Memory Statistics Definitions. http://communities.vmware.com/docs/DOC-5230

[5] Performance Evaluation of Intel EPT Hardware Assist. VMware. http://www.vmware.com/resources/techresources/10006

[6] Performance Evaluation of AMD RVI Hardware Assist. VMware. http://www.vmware.com/resources/techresources/1079

[7] The buffer cache. http://tldp.org/LDP/sag/html/buffer-cache.html

---

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

Understanding Memory Resource Management in VMware ESX Server
Source: Technical Marketing, SD
Revision: 20090820

vmware®