Memory Resource Management in VMware ESX Server

Carl A. Waldspurger

VMware, Inc.
Palo Alto, CA 94304 USA
carl@vmware.com

Abstract

VMware ESX Server is a thin software layer designed to multiplex hardware resources efficiently among virtual machines running unmodified commodity operating systems. This paper introduces several novel ESX Server mechanisms and policies for managing memory. A *ballooning* technique reclaims the pages considered least valuable by the operating system running in a virtual machine. An *idle memory tax* achieves efficient memory utilization while maintaining performance isolation guarantees. *Content-based page sharing* and *hot I/O page remapping* exploit transparent page remapping to eliminate redundancy and reduce copying overheads. These techniques are combined to efficiently support virtual machine workloads that overcommit memory.

1 Introduction

Recent industry trends, such as server consolidation and the proliferation of inexpensive shared-memory multiprocessors, have fueled a resurgence of interest in server virtualization techniques. Virtual machines are particularly attractive for server virtualization. Each *virtual machine (VM)* is given the illusion of being a dedicated physical machine that is fully protected and isolated from other virtual machines. Virtual machines are also convenient abstractions of server workloads, since they cleanly encapsulate the entire state of a running system, including both user-level applications and kernel-mode operating system services.

In many computing environments, individual servers are underutilized, allowing them to be consolidated as virtual machines on a single physical server with little or no performance penalty. Similarly, many small servers can be consolidated onto fewer larger machines to simplify management and reduce costs. Ideally, system administrators should be able to flexibly overcommit memory, processor, and other resources in order to reap the benefits of statistical multiplexing, while still providing resource guarantees to VMs of varying importance.

Virtual machines have been used for decades to allow multiple copies of potentially different operating systems to run concurrently on a single hardware platform [8]. A *virtual machine monitor (VMM)* is a software layer that virtualizes hardware resources, exporting a virtual hardware interface that reflects the underlying machine architecture. For example, the influential VM/370 virtual machine system [6] supported multiple concurrent virtual machines, each of which believed it was running natively on the IBM System/370 hardware architecture [10]. More recent research, exemplified by Disco [3, 9], has focused on using virtual machines to provide scalability and fault containment for commodity operating systems running on large-scale shared-memory multiprocessors.

VMware *ESX Server* is a thin software layer designed to multiplex hardware resources efficiently among virtual machines. The current system virtualizes the Intel IA-32 architecture [13]. It is in production use on servers running multiple instances of unmodified operating systems such as Microsoft Windows 2000 Advanced Server and Red Hat Linux 7.2. The design of ESX Server differs significantly from VMware Workstation, which uses a *hosted* virtual machine architecture [23] that takes advantage of a pre-existing operating system for portable I/O device support. For example, a Linux-hosted VMM intercepts attempts by a VM to read sectors from its virtual disk, and issues a `read()` system call to the underlying Linux host OS to retrieve the corresponding data. In contrast, ESX Server manages system hardware directly, providing significantly higher I/O performance and complete control over resource management.

The need to run existing operating systems without modification presented a number of interesting challenges. Unlike IBM's mainframe division, we were unable to influence the design of the guest operating systems running within virtual machines. Even the Disco prototypes [3, 9], designed to run unmodified operating systems, resorted to minor modifications in the IRIX kernel sources.

This paper introduces several novel mechanisms and policies that ESX Server 1.5 [29] uses to manage memory. High-level resource management policies compute a target memory allocation for each VM based on specified parameters and system load. These allocations are achieved by invoking lower-level mechanisms to reclaim memory from virtual machines. In addition, a background activity exploits opportunities to share identical pages between VMs, reducing overall memory pressure on the system.

In the following sections, we present the key aspects of memory resource management using a bottom-up approach, describing low-level mechanisms before discussing the high-level algorithms and policies that coordinate them. Section 2 describes low-level memory virtualization. Section 3 discusses mechanisms for reclaiming memory to support dynamic resizing of virtual machines. A general technique for conserving memory by sharing identical pages between VMs is presented in Section 4. Section 5 discusses the integration of working-set estimates into a proportional-share allocation algorithm. Section 6 describes the high-level allocation policy that coordinates these techniques. Section 7 presents a remapping optimization that reduces I/O copying overheads in large-memory systems. Section 8 examines related work. Finally, we summarize our conclusions and highlight opportunities for future work in Section 9.

2 Memory Virtualization

A *guest* operating system that executes within a virtual machine expects a zero-based physical address space, as provided by real hardware. ESX Server gives each VM this illusion, virtualizing physical memory by adding an extra level of address translation. Borrowing terminology from Disco [3], a *machine address* refers to actual hardware memory, while a *physical address* is a software abstraction used to provide the illusion of hardware memory to a virtual machine. We will often use "physical" in quotes to highlight this deviation from its usual meaning.

ESX Server maintains a *pmap* data structure for each VM to translate "physical" page numbers (PPNs) to machine page numbers (MPNs). VM instructions that manipulate guest OS page tables or TLB contents are intercepted, preventing updates to actual MMU state. Separate *shadow page tables*, which contain virtual-to-machine page mappings, are maintained for use by the processor and are kept consistent with the physical-to-machine mappings in the *pmap*.[1] This approach permits ordinary memory references to execute without additional overhead, since the hardware TLB will cache direct virtual-to-machine address translations read from the shadow page table.

The extra level of indirection in the memory system is extremely powerful. The server can remap a "physical" page by changing its PPN-to-MPN mapping, in a manner that is completely transparent to the VM. The server may also monitor or interpose on guest memory accesses.

3 Reclamation Mechanisms

ESX Server supports overcommitment of memory to facilitate a higher degree of server consolidation than would be possible with simple static partitioning. Overcommitment means that the total size configured for all running virtual machines exceeds the total amount of actual machine memory. The system manages the allocation of memory to VMs automatically based on configuration parameters and system load.

Each virtual machine is given the illusion of having a fixed amount of physical memory. This *max size* is a configuration parameter that represents the maximum amount of machine memory it can be allocated. Since commodity operating systems do not yet support dynamic changes to physical memory sizes, this size remains constant after booting a guest OS. A VM will be allocated its maximum size when memory is not overcommitted.

3.1 Page Replacement Issues

When memory is overcommitted, ESX Server must employ some mechanism to reclaim space from one or more virtual machines. The standard approach used by earlier virtual machine systems is to introduce another level of paging [9, 20], moving some VM "physical" pages to a swap area on disk. Unfortunately, an extra level of paging requires a meta-level page replacement policy: the virtual machine system must choose not only the VM from which to revoke memory, but also which of its particular pages to reclaim.

In general, a meta-level page replacement policy must make relatively uninformed resource management decisions. The best information about which pages are least

[1] The IA-32 architecture has hardware mechanisms that walk in-memory page tables and reload the TLB [13].

valuable is known only by the guest operating system within each VM. Although there is no shortage of clever page replacement algorithms [26], this is actually the crux of the problem. A sophisticated meta-level policy is likely to introduce performance anomalies due to unintended interactions with native memory management policies in guest operating systems. This situation is exacerbated by diverse and often undocumented guest OS policies [1], which may vary across OS versions and may even depend on performance hints from applications [4].

The fact that paging is transparent to the guest OS can also result in a *double paging* problem, even when the meta-level policy is able to select the same page that the native guest OS policy would choose [9, 20]. Suppose the meta-level policy selects a page to reclaim and pages it out. If the guest OS is under memory pressure, it may choose the very same page to write to its own virtual paging device. This will cause the page contents to be faulted in from the system paging device, only to be immediately written out to the virtual paging device.

3.2 Ballooning

Ideally, a VM from which memory has been reclaimed should perform as if it had been configured with less memory. ESX Server uses a *ballooning* technique to achieve such predictable performance by coaxing the guest OS into cooperating with it when possible. This process is depicted in Figure 1.

A small *balloon* module is loaded into the guest OS as a pseudo-device driver or kernel service. It has no external interface within the guest, and communicates with ESX Server via a private channel. When the server wants to reclaim memory, it instructs the driver to "inflate" by allocating pinned physical pages within the VM, using appropriate native interfaces. Similarly, the server may "deflate" the balloon by instructing it to deallocate previously-allocated pages.

Inflating the balloon increases memory pressure in the guest OS, causing it to invoke its own native memory management algorithms. When memory is plentiful, the guest OS will return memory from its free list. When memory is scarce, it must reclaim space to satisfy the driver allocation request. The guest OS decides which particular pages to reclaim and, if necessary, pages them out to its own virtual disk. The balloon driver communicates the physical page number for each allocated page to ESX Server, which may then reclaim the corresponding machine page. Deflating the balloon frees up

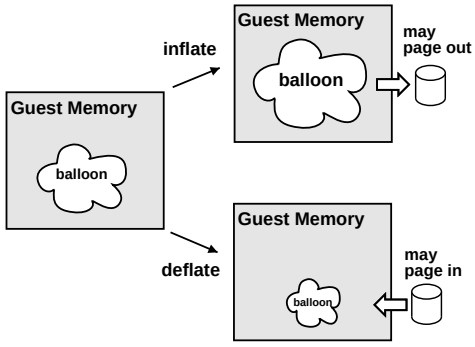

Figure 1: **Ballooning.** ESX Server controls a *balloon* module running within the guest, directing it to allocate guest pages and pin them in "physical" memory. The machine pages backing this memory can then be reclaimed by ESX Server. Inflating the balloon increases memory pressure, forcing the guest OS to invoke its own memory management algorithms. The guest OS may page out to its virtual disk when memory is scarce. Deflating the balloon decreases pressure, freeing guest memory.

memory for general use within the guest OS.

Although a guest OS should not touch any physical memory it allocates to a driver, ESX Server does not depend on this property for correctness. When a guest PPN is ballooned, the system annotates its *pmap* entry and deallocates the associated MPN. Any subsequent attempt to access the PPN will generate a fault that is handled by the server; this situation is rare, and most likely the result of complete guest failure, such as a reboot or crash. The server effectively "pops" the balloon, so that the next interaction with (any instance of) the guest driver will first reset its state. The fault is then handled by allocating a new MPN to back the PPN, just as if the page was touched for the first time.[2]

Our balloon drivers for the Linux, FreeBSD, and Windows operating systems poll the server once per second to obtain a target balloon size, and they limit their allocation rates adaptively to avoid stressing the guest OS. Standard kernel interfaces are used to allocate physical pages, such as `get_free_page()` in Linux, and `MmAllocatePagesForMdl()` or `MmProbeAndLockPages()` in Windows.

Future guest OS support for hot-pluggable memory cards would enable an additional form of coarse-grained ballooning. Virtual memory cards could be inserted into

[2]ESX Server zeroes the contents of newly-allocated machine pages to avoid leaking information between VMs. Allocation also respects cache coloring by the guest OS; when possible, distinct PPN colors are mapped to distinct MPN colors.

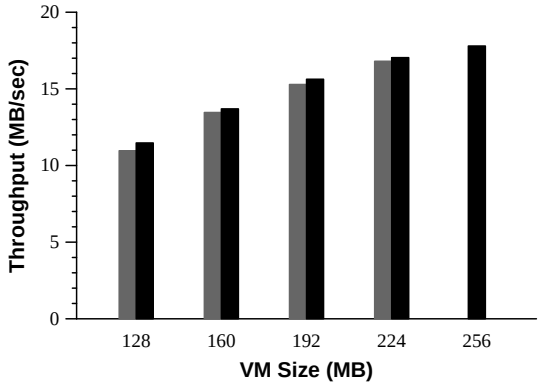

Figure 2: **Balloon Performance.** Throughput of single Linux VM running `dbench` with 40 clients. The black bars plot the performance when the VM is configured with main memory sizes ranging from 128 MB to 256 MB. The gray bars plot the performance of the same VM configured with 256 MB, ballooned down to the specified size.

or removed from a VM in order to rapidly adjust its physical memory size.

To demonstrate the effectiveness of ballooning, we used the synthetic `dbench` benchmark [28] to simulate fileserver performance under load from 40 clients. This workload benefits significantly from additional memory, since a larger buffer cache can absorb more disk traffic. For this experiment, ESX Server was running on a dual-processor Dell Precision 420, configured to execute one VM running Red Hat Linux 7.2 on a single 800 MHz Pentium III CPU.

Figure 2 presents `dbench` throughput as a function of VM size, using the average of three consecutive runs for each data point. The ballooned VM tracks non-ballooned performance closely, with an observed overhead ranging from 4.4% at 128 MB (128 MB balloon) down to 1.4% at 224 MB (32 MB balloon). This overhead is primarily due to guest OS data structures that are sized based on the amount of "physical" memory; the Linux kernel uses more space in a 256 MB system than in a 128 MB system. Thus, a 256 MB VM ballooned down to 128 MB has slightly less free space than a VM configured with exactly 128 MB.

Despite its advantages, ballooning does have limitations. The balloon driver may be uninstalled, disabled explicitly, unavailable while a guest OS is booting, or temporarily unable to reclaim memory quickly enough to satisfy current system demands. Also, upper bounds on reasonable balloon sizes may be imposed by various guest OS limitations.

3.3 Demand Paging

ESX Server preferentially uses ballooning to reclaim memory, treating it as a common-case optimization. When ballooning is not possible or insufficient, the system falls back to a paging mechanism. Memory is reclaimed by paging out to an ESX Server swap area on disk, without any guest involvement.

The ESX Server swap daemon receives information about target swap levels for each VM from a higher-level policy module. It manages the selection of candidate pages and coordinates asynchronous page outs to a swap area on disk. Conventional optimizations are used to maintain free slots and cluster disk writes.

A randomized page replacement policy is used to prevent the types of pathological interference with native guest OS memory management algorithms described in Section 3.1. This choice was also guided by the expectation that paging will be a fairly uncommon operation. Nevertheless, we are investigating more sophisticated page replacement algorithms, as well policies that may be customized on a per-VM basis.

4 Sharing Memory

Server consolidation presents numerous opportunities for sharing memory between virtual machines. For example, several VMs may be running instances of the same guest OS, have the same applications or components loaded, or contain common data. ESX Server exploits these sharing opportunities, so that server workloads running in VMs on a single machine often consume less memory than they would running on separate physical machines. As a result, higher levels of overcommitment can be supported efficiently.

4.1 Transparent Page Sharing

Disco [3] introduced *transparent page sharing* as a method for eliminating redundant copies of pages, such as code or read-only data, across virtual machines. Once copies are identified, multiple guest "physical" pages are mapped to the same machine page, and marked copy-on-write. Writing to a shared page causes a fault that generates a private copy.

Unfortunately, Disco required several guest OS modifications to identify redundant copies as they were created. For example, the `bcopy()` routine was hooked to

enable file buffer cache sharing across virtual machines. Some sharing also required the use of non-standard or restricted interfaces. A special network interface with support for large packets facilitated sharing data communicated between VMs on a virtual subnet. Interposition on disk accesses allowed data from shared, non-persistent disks to be shared across multiple guests.

4.2 Content-Based Page Sharing

Because modifications to guest operating system internals are not possible in our environment, and changes to application programming interfaces are not acceptable, ESX Server takes a completely different approach to page sharing. The basic idea is to identify page copies by their contents. Pages with identical contents can be shared regardless of when, where, or how those contents were generated. This general-purpose approach has two key advantages. First, it eliminates the need to modify, hook, or even understand guest OS code. Second, it can identify more opportunities for sharing; by definition, all potentially shareable pages can be identified by their contents.

The cost for this unobtrusive generality is that work must be performed to scan for sharing opportunities. Clearly, comparing the contents of each page with every other page in the system would be prohibitively expensive; naive matching would require O(n^2) page comparisons. Instead, hashing is used to identify pages with potentially-identical contents efficiently.

A hash value that summarizes a page's contents is used as a lookup key into a hash table containing entries for other pages that have already been marked copy-on-write (COW). If the hash value for the new page matches an existing entry, it is very likely that the pages are identical, although false matches are possible. A successful match is followed by a full comparison of the page contents to verify that the pages are identical.

Once a match has been found with an existing shared page, a standard copy-on-write technique can be used to share the pages, and the redundant copy can be reclaimed. Any subsequent attempt to write to the shared page will generate a fault, transparently creating a private copy of the page for the writer.

If no match is found, one option is to mark the page COW in anticipation of some future match. However, this simplistic approach has the undesirable side-effect of marking *every* scanned page copy-on-write, incurring unnecessary overhead on subsequent writes. As an optimization, an unshared page is not marked COW, but instead tagged as a special *hint* entry. On any future match with another page, the contents of the hint page are rehashed. If the hash has changed, then the hint page has been modified, and the stale hint is removed. If the hash is still valid, a full comparison is performed, and the pages are shared if it succeeds.

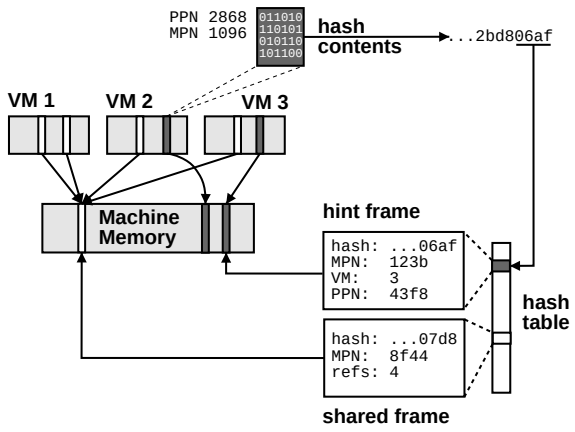

Figure 3: **Content-Based Page Sharing.** ESX Server scans for sharing opportunities, hashing the contents of candidate PPN 0x2868 in VM 2. The hash is used to index into a table containing other scanned pages, where a match is found with a hint frame associated with PPN 0x43f8 in VM 3. If a full comparison confirms the pages are identical, the PPN-to-MPN mapping for PPN 0x2868 in VM 2 is changed from MPN 0x1096 to MPN 0x123b, both PPNs are marked COW, and the redundant MPN is reclaimed.

Higher-level page sharing policies control when and where to scan for copies. One simple option is to scan pages incrementally at some fixed rate. Pages could be considered sequentially, randomly, or using heuristics to focus on the most promising candidates, such as pages marked read-only by the guest OS, or pages from which code has been executed. Various policies can be used to limit CPU overhead, such as scanning only during otherwise-wasted idle cycles.

4.3 Implementation

The ESX Server implementation of content-based page sharing is illustrated in Figure 3. A single global hash table contains frames for all scanned pages, and chaining is used to handle collisions. Each frame is encoded compactly in 16 bytes. A *shared frame* consists of a hash value, the machine page number (MPN) for the shared page, a reference count, and a link for chaining. A *hint frame* is similar, but encodes a truncated

hash value to make room for a reference back to the corresponding guest page, consisting of a VM identifier and a physical page number (PPN). The total space overhead for page sharing is less than 0.5% of system memory.

Unlike the Disco page sharing implementation, which maintained a backmap for each shared page, ESX Server uses a simple reference count. A small 16-bit count is stored in each frame, and a separate overflow table is used to store any extended frames with larger counts. This allows highly-shared pages to be represented compactly. For example, the empty *zero page* filled completely with zero bytes is typically shared with a large reference count. A similar overflow technique for large reference counts was used to save space in the early OOZE virtual memory system [15].

A fast, high-quality hash function [14] is used to generate a 64-bit hash value for each scanned page. Since the chance of encountering a false match due to hash aliasing is incredibly small[3] the system can make the simplifying assumption that all shared pages have unique hash values. Any page that happens to yield a false match is considered ineligible for sharing.

The current ESX Server page sharing implementation scans guest pages randomly. Although more sophisticated approaches are possible, this policy is simple and effective. Configuration options control maximum per-VM and system-wide page scanning rates. Typically, these values are set to ensure that page sharing incurs negligible CPU overhead. As an additional optimization, the system always attempts to share a page before paging it out to disk.

To evaluate the ESX Server page sharing implementation, we conducted experiments to quantify its effectiveness at reclaiming memory and its overhead on system performance. We first analyze a "best case" workload consisting of many homogeneous VMs, in order to demonstrate that ESX Server is able to reclaim a large fraction of memory when the potential for sharing exists. We then present additional data collected from production deployments serving real users.

We performed a series of controlled experiments using identically-configured virtual machines, each running Red Hat Linux 7.2 with 40 MB of "physical" memory. Each experiment consisted of between one and ten concurrent VMs running SPEC95 benchmarks for thirty minutes. For these experiments, ESX Server was running on a Dell PowerEdge 1400SC multiprocessor with two 933 MHz Pentium III CPUs.

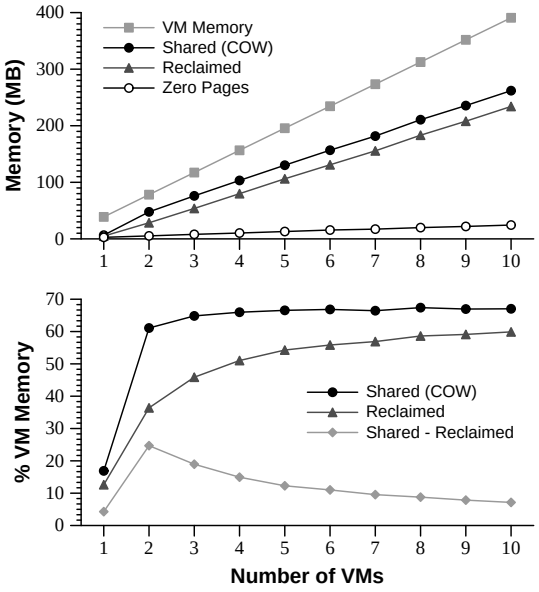

Figure 4: **Page Sharing Performance.** Sharing metrics for a series of experiments consisting of identical Linux VMs running SPEC95 benchmarks. The top graph indicates the absolute amounts of memory shared and saved increase smoothly with the number of concurrent VMs. The bottom graph plots these metrics as a percentage of aggregate VM memory. For large numbers of VMs, sharing approaches 67% and nearly 60% of all VM memory is reclaimed.

Figure 4 presents several sharing metrics plotted as a function of the number of concurrent VMs. Surprisingly, some sharing is achieved with only a single VM. Nearly 5 MB of memory was reclaimed from a single VM, of which about 55% was due to shared copies of the zero page. The top graph shows that after an initial jump in sharing between the first and second VMs, the total amount of memory shared increases linearly with the number of VMs, as expected. Little sharing is attributed to zero pages, indicating that most sharing is due to redundant code and read-only data pages. The bottom graph plots these metrics as a percentage of aggregate VM memory. As the number of VMs increases, the sharing level approaches 67%, revealing an overlap of approximately two-thirds of all memory between the VMs. The amount of memory required to contain the single copy of each common shared page (labelled Shared – Reclaimed), remains nearly constant, decreasing as a percentage of overall VM memory.

[3] Assuming page contents are randomly mapped to 64-bit hash values, the probability of a single collision doesn't exceed 50% until approximately $\sqrt{2^{64}} = 2^{32}$ distinct pages are hashed [14]. For a static snapshot of the largest possible IA-32 memory configuration with 2^{24} pages (64 GB), the collision probability is less than 0.01%.

		Total	Shared		Reclaimed	
	Guest Types	MB	MB	%	MB	%
A	10 WinNT	2048	880	42.9	673	32.9
B	9 Linux	1846	539	29.2	345	18.7
C	5 Linux	1658	165	10.0	120	7.2

Figure 5: **Real-World Page Sharing.** Sharing metrics from production deployments of ESX Server. (a) Ten Windows NT VMs serving users at a Fortune 50 company, running a variety of database (Oracle, SQL Server), web (IIS, Websphere), development (Java, VB), and other applications. (b) Nine Linux VMs serving a large user community for a nonprofit organization, executing a mix of web (Apache), mail (Majordomo, Postfix, POP/IMAP, MailArmor), and other servers. (c) Five Linux VMs providing web proxy (Squid), mail (Postfix, RAV), and remote access (ssh) services to VMware employees.

The CPU overhead due to page sharing was negligible. We ran an identical set of experiments with page sharing disabled, and measured no significant difference in the aggregate throughput reported by the CPU-bound benchmarks running in the VMs. Over all runs, the aggregate throughput was actually 0.5% *higher* with page sharing enabled, and ranged from 1.6% lower to 1.8% higher. Although the effect is generally small, page sharing does improve memory locality, and may therefore increase hit rates in physically-indexed caches.

These experiments demonstrate that ESX Server is able to exploit sharing opportunities effectively. Of course, more diverse workloads will typically exhibit lower degrees of sharing. Nevertheless, many real-world server consolidation workloads do consist of numerous VMs running the same guest OS with similar applications. Since the amount of memory reclaimed by page sharing is very workload-dependent, we collected memory sharing statistics from several ESX Server systems in production use.

Figure 5 presents page sharing metrics collected from three different production deployments of ESX Server. Workload A, from a corporate IT department at a Fortune 50 company, consists of ten Windows NT 4.0 VMs running a wide variety of database, web, and other servers. Page sharing reclaimed nearly a third of all VM memory, saving 673 MB. Workload B, from a nonprofit organization's Internet server, consists of nine Linux VMs ranging in size from 64 MB to 768 MB, running a mix of mail, web, and other servers. In this case, page sharing was able to reclaim 18.7% of VM memory, saving 345 MB, of which 70 MB was attributed to zero pages. Finally, workload C is from VMware's own IT department, and provides web proxy, mail, and remote access services to our employees using five Linux VMs ranging in size from 32 MB to 512 MB. Page sharing reclaimed about 7% of VM memory, for a savings of 120 MB, of which 25 MB was due to zero pages.

5 Shares vs. Working Sets

Traditional operating systems adjust memory allocations to improve some aggregate, system-wide performance metric. While this is usually a desirable goal, it often conflicts with the need to provide quality-of-service guarantees to clients of varying importance. Such guarantees are critical for server consolidation, where each VM may be entitled to different amounts of resources based on factors such as importance, ownership, administrative domains, or even the amount of money paid to a service provider for executing the VM. In such cases, it can be preferable to penalize a less important VM, even when that VM would derive the largest performance benefit from additional memory.

ESX Server employs a new allocation algorithm that is able to achieve efficient memory utilization while maintaining memory performance isolation guarantees. In addition, an explicit parameter is introduced that allows system administrators to control the relative importance of these conflicting goals.

5.1 Share-Based Allocation

In proportional-share frameworks, resource rights are encapsulated by *shares*, which are owned by *clients* that consume resources.[4] A client is entitled to consume resources proportional to its share allocation; it is guaranteed a minimum resource fraction equal to its fraction of the total shares in the system. Shares represent relative resource rights that depend on the total number of shares contending for a resource. Client allocations degrade gracefully in overload situations, and clients proportionally benefit from extra resources when some allocations are underutilized.

Both randomized and deterministic algorithms have been proposed for proportional-share allocation of space-shared resources. The dynamic *min-funding revocation* algorithm [31, 32] is simple and effective. When one client demands more space, a replacement algorithm selects a victim client that relinquishes some of its previously-allocated space. Memory is revoked from the

[4] Shares are alternatively referred to as *tickets* or *weights* in the literature. The term *clients* is used to abstractly refer to entities such as threads, processes, VMs, users, or groups.

client that owns the fewest shares per allocated page. Using an economic analogy, the *shares-per-page* ratio can be interpreted as a price; revocation reallocates memory away from clients paying a lower price to those willing to pay a higher price.

5.2 Reclaiming Idle Memory

A significant limitation of pure proportional-share algorithms is that they do not incorporate any information about active memory usage or working sets. Memory is effectively partitioned to maintain specified ratios. However, idle clients with many shares can hoard memory unproductively, while active clients with few shares suffer under severe memory pressure. In general, the goals of performance isolation and efficient memory utilization often conflict. Previous attempts to cross-apply techniques from proportional-share CPU resource management to compensate for idleness have not been successful [25].

ESX Server resolves this problem by introducing an *idle memory tax*. The basic idea is to charge a client more for an idle page than for one it is actively using. When memory is scarce, pages will be reclaimed preferentially from clients that are not actively using their full allocations. The tax rate specifies the maximum fraction of idle pages that may be reclaimed from a client. If the client later starts using a larger portion of its allocated memory, its allocation will increase, up to its full share.

Min-funding revocation is extended to use an adjusted shares-per-page ratio. For a client with S shares and an allocation of P pages, of which a fraction f are active, the adjusted shares-per-page ratio ρ is

$$\rho = \frac{S}{P \cdot (f + k \cdot (1 - f))}$$

where the idle page cost $k = 1/(1 - \tau)$ for a given tax rate $0 \leq \tau < 1$.

The tax rate τ provides explicit control over the desired policy for reclaiming idle memory. At one extreme, $\tau = 0$ specifies pure share-based isolation. At the other, $\tau \approx 1$ specifies a policy that allows all of a client's idle memory to be reclaimed for more productive uses.

The ESX Server idle memory tax rate is a configurable parameter that defaults to 75%. This allows most idle memory in the system to be reclaimed, while still providing a buffer against rapid working set increases, masking the latency of system reclamation activity such as ballooning and swapping.[5]

5.3 Measuring Idle Memory

For the idle memory tax to be effective, the server needs an efficient mechanism to estimate the fraction of memory in active use by each virtual machine. However, specific active and idle pages need not be identified individually.

One option is to extract information using native interfaces within each guest OS. However, this is impractical, since diverse activity metrics are used across various guests, and those metrics tend to focus on per-process working sets. Also, guest OS monitoring typically relies on access bits associated with page table entries, which are bypassed by DMA for device I/O.

ESX Server uses a statistical sampling approach to obtain aggregate VM working set estimates directly, without any guest involvement. Each VM is sampled independently, using a configurable sampling period defined in units of VM execution time. At the start of each sampling period, a small number n of the virtual machine's "physical" pages are selected randomly using a uniform distribution. Each sampled page is tracked by invalidating any cached mappings associated with its PPN, such as hardware TLB entries and virtualized MMU state. The next guest access to a sampled page will be intercepted to re-establish these mappings, at which time a touched page count t is incremented. At the end of the sampling period, a statistical estimate of the fraction f of memory actively accessed by the VM is $f = t/n$.

The sampling rate may be controlled to tradeoff overhead and accuracy. By default, ESX Server samples 100 pages for each 30 second period. This results in at most 100 minor page faults per period, incurring negligible overhead while still producing reasonably accurate working set estimates.

Estimates are smoothed across multiple sampling periods. Inspired by work on balancing stability and agility from the networking domain [16], we maintain separate exponentially-weighted moving averages with different gain parameters. A slow moving average is used to produce a smooth, stable estimate. A fast moving average

[5] The configured tax rate applies uniformly to all VMs. While the underlying implementation supports separate, per-VM tax rates, this capability is not currently exposed to users. Customized or graduated tax rates may be useful for more sophisticated control over relative allocations and responsiveness.

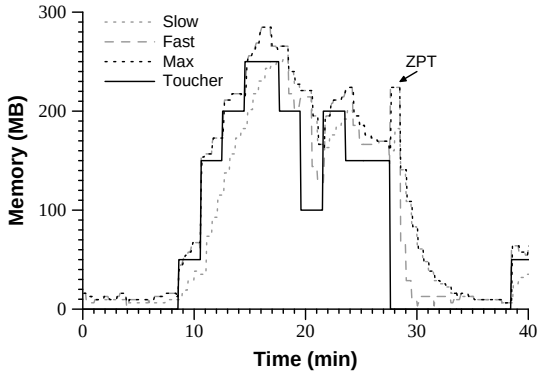

Figure 6: **Active Memory Sampling.** A Windows VM executes a simple memory *toucher* application. The solid black line indicates the amount of memory repeatedly touched, which is varied over time. The dotted black line is the sampling-based statistical estimate of overall VM memory usage, including background Windows activities. The estimate is computed as the *max* of *fast* (gray dashed line) and *slow* (gray dotted line) moving averages. The spike labelled *ZPT* is due to the Windows "zero page thread."

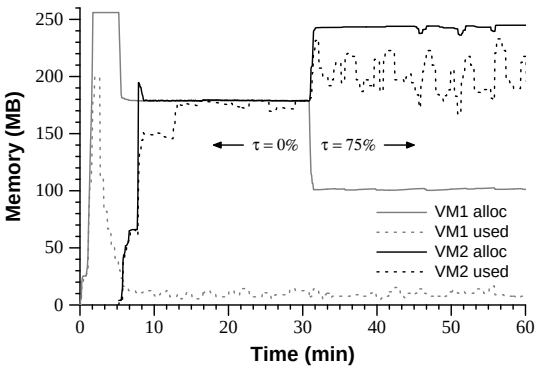

Figure 7: **Idle Memory Tax.** Two VMs with identical share allocations are each configured with 256 MB in an overcommitted system. VM1 (gray) runs Windows, and remains idle after booting. VM2 (black) executes a memory-intensive Linux workload. For each VM, ESX Server allocations are plotted as solid lines, and estimated memory usage is indicated by dotted lines. With an initial tax rate of 0%, the VMs each converge on the same 179 MB allocation. When the tax rate is increased to 75%, idle memory is reclaimed from VM1 and reallocated to VM2, boosting its performance by over 30%.

adapts quickly to working set changes. Finally, a version of the fast average that incorporates counts from the current sampling period is updated incrementally to reflect rapid intra-period changes.

The server uses the maximum of these three values to estimate the amount of memory being actively used by the guest. This causes the system to respond rapidly to increases in memory usage and more gradually to decreases in memory usage, which is the desired behavior. A VM that had been idle and starts using memory is allowed to ramp up to its share-based allocation quickly, while a VM that had been active and decreases its working set has its idle memory reclaimed slowly via the idle memory tax.

5.4 Experimental Results

This section presents quantitative experiments that demonstrate the effectiveness of memory sampling and idle memory taxation. Memory sampling is used to estimate the fraction of memory actively used by each VM. These estimates are then incorporated into the idle memory tax computations performed by the share-based memory allocation algorithm.

Figure 6 presents the results of an experiment designed to illustrate the memory sampling technique. For this experiment, ESX Server was running on a dual-processor Dell Precision 420, configured to execute one VM running Windows 2000 Advanced Server on a single 800 MHz Pentium III CPU.

A user-level *toucher* application allocates and repeatedly accesses a controlled amount of memory that is varied between 50 MB and 250 MB. An additional 10–20 MB is accessed by standard Windows background activities. As expected, the statistical estimate of active memory usage responds quickly as more memory is touched, tracking the fast moving average, and more slowly as less memory is touched, tracking the slow moving average.

We were originally surprised by the unexpected spike immediately after the toucher application terminates, an effect that does not occur when the same experiment is run under Linux. This is caused by the Windows "zero page thread" that runs only when no other threads are runnable, clearing the contents of pages it moves from the free page list to the zeroed page list [22].

Figure 7 presents experimental results that demonstrate the effectiveness of imposing a tax on idle memory. For this experiment, ESX Server was running on a Dell Precision 420 multiprocessor with two 800 MHz Pentium III CPUs and 512 MB RAM, of which approximately 360 MB was available for executing VMs.[6]

[6]Some memory is required for per-VM virtualization overheads, which are discussed in Section 6.2. Additional memory is required for ESX Server itself; the smallest recommended configuration is 512 MB.

Two VMs with identical share allocations are each configured with 256 MB "physical" memory. The first VM that powers on runs Windows 2000 Advanced Server, and remains idle after booting. A few minutes later, a second VM is started, running a memory-intensive dbench workload [28] under Red Hat Linux 7.2. The initial tax rate is set to $\tau = 0$, resulting in a pure share-based allocation. Despite the large difference in actual memory usage, each VM receives the same 179 MB allocation from ESX Server. In the middle of the experiment, the tax rate is increased to $\tau = 0.75$, causing memory to be reclaimed from the idle Windows VM and reallocated to the active Linux VM running dbench. The dbench workload benefits significantly from the additional memory, increasing throughput by over 30% after the tax rate change.

6 Allocation Policies

ESX Server computes a target memory allocation for each VM based on both its share-based entitlement and an estimate of its working set, using the algorithm presented in Section 5. These targets are achieved via the ballooning and paging mechanisms presented in Section 3. Page sharing runs as an additional background activity that reduces overall memory pressure on the system. This section describes how these various mechanisms are coordinated in response to specified allocation parameters and system load.

6.1 Parameters

System administrators use three basic parameters to control the allocation of memory to each VM: a *min* size, a *max* size, and memory *shares*. The *min* size is a guaranteed lower bound on the amount of memory that will be allocated to the VM, even when memory is overcommitted. The *max* size is the amount of "physical" memory configured for use by the guest OS running in the VM. Unless memory is overcommitted, VMs will be allocated their *max* size.

Memory *shares* entitle a VM to a fraction of physical memory, based on a proportional-share allocation policy. For example, a VM that has twice as many shares as another is generally entitled to consume twice as much memory, subject to their respective *min* and *max* constraints, provided they are both actively using their allocated memory.

6.2 Admission Control

An admission control policy ensures that sufficient unreserved memory and server swap space is available before a VM is allowed to power on. Machine memory must be reserved for the guaranteed *min* size, as well as additional *overhead* memory required for virtualization, for a total of *min* + *overhead*. Overhead memory includes space for the VM graphics frame buffer and various virtualization data structures such as the *pmap* and shadow page tables (see Section 2). Typical VMs reserve 32 MB for overhead, of which 4 to 8 MB is devoted to the frame buffer, and the remainder contains implementation-specific data structures. Additional memory is required for VMs larger than 1 GB.

Disk swap space must be reserved for the remaining VM memory; i.e. *max* − *min*. This reservation ensures the system is able to preserve VM memory under any circumstances; in practice, only a small fraction of this disk space is typically used. Similarly, while memory reservations are used for admission control, actual memory allocations vary dynamically, and unused reservations are not wasted.

6.3 Dynamic Reallocation

ESX Server recomputes memory allocations dynamically in response to various events: changes to system-wide or per-VM allocation parameters by a system administrator, the addition or removal of a VM from the system, and changes in the amount of free memory that cross predefined thresholds. Additional rebalancing is performed periodically to reflect changes in idle memory estimates for each VM.

Most operating systems attempt to maintain a minimum amount of free memory. For example, BSD Unix normally starts reclaiming memory when the percentage of free memory drops below 5% and continues reclaiming until the free memory percentage reaches 7% [18]. ESX Server employs a similar approach, but uses four thresholds to reflect different reclamation states: *high*, *soft*, *hard*, and *low*, which default to 6%, 4%, 2%, and 1% of system memory, respectively.

In the the *high* state, free memory is sufficient and no reclamation is performed. In the *soft* state, the system reclaims memory using ballooning, and resorts to paging only in cases where ballooning is not possible. In the *hard* state, the system relies on paging to forcibly reclaim memory. In the rare event that free memory transiently falls below the *low* threshold, the system con-

tinues to reclaim memory via paging, and additionally blocks the execution of all VMs that are above their target allocations.

In all memory reclamation states, the system computes target allocations for VMs to drive the aggregate amount of free space above the *high* threshold. A transition to a lower reclamation state occurs when the amount of free memory drops below the lower threshold. After reclaiming memory, the system transitions back to the next higher state only after significantly exceeding the higher threshold; this hysteresis prevents rapid state fluctuations.

To demonstrate dynamic reallocation we ran a workload consisting of five virtual machines. A pair of VMs executed a Microsoft Exchange benchmark; one VM ran an Exchange Server under Windows 2000 Server, and a second VM ran a load generator client under Windows 2000 Professional. A different pair executed a Citrix MetaFrame benchmark; one VM ran a MetaFrame Server under Windows 2000 Advanced Server, and a second VM ran a load generator client under Windows 2000 Server. A final VM executed database queries with Microsoft SQL Server under Windows 2000 Advanced Server. The Exchange VMs were each configured with 256 MB memory; the other three VMs were each configured with 320 MB. The *min* size for each VM was set to half of its configured *max* size, and memory shares were allocated proportional to the *max* size of each VM.

For this experiment, ESX Server was running on an IBM Netfinity 8500R multiprocessor with eight 550 MHz Pentium III CPUs. To facilitate demonstrating the effects of memory pressure, machine memory was deliberately limited so that only 1 GB was available for executing VMs. The aggregate VM workload was configured to use a total of 1472 MB; with the additional 160 MB required for overhead memory, memory was overcommitted by more than 60%.

Figure 8(a) presents ESX Server allocation states during the experiment. Except for brief transitions early in the run, nearly all time is spent in the *high* and *soft* states. Figure 8(b) plots several allocation metrics over time. When the experiment is started, all five VMs boot concurrently. Windows zeroes the contents of all pages in "physical" memory while booting. This causes the system to become overcommitted almost immediately, as each VM accesses all of its memory. Since the Windows balloon drivers are not started until late in the boot sequence, ESX Server is forced to start paging to disk. Fortunately, the "share before swap" optimization described in Section 4.3 is very effective: 325 MB of zero pages

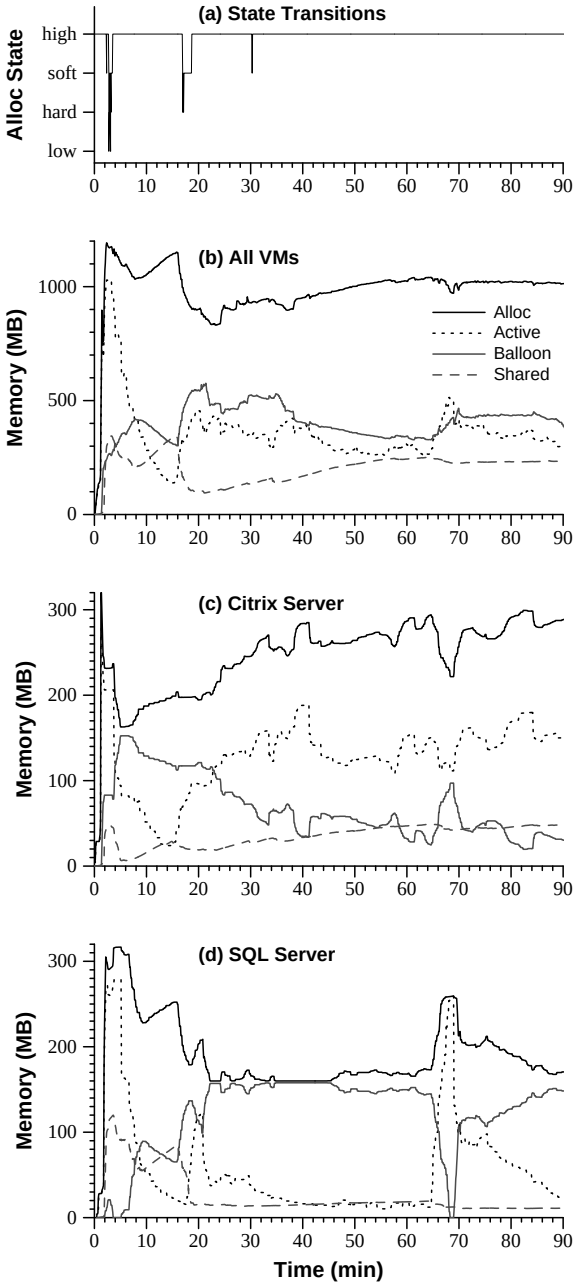

Figure 8: **Dynamic Reallocation.** Memory allocation metrics over time for a consolidated workload consisting of five Windows VMs: Microsoft Exchange (separate server and client load generator VMs), Citrix MetaFrame (separate server and client load generator VMs), and Microsoft SQL Server. (a) ESX Server allocation state transitions. (b) Aggregate allocation metrics summed over all five VMs. (c) Allocation metrics for MetaFrame Server VM. (d) Allocation metrics for SQL Server VM.

are reclaimed via page sharing, while only 35 MB of non-zero data is actually written to disk.[7] As a result of sharing, the aggregate allocation to all VMs approaches 1200 MB, exceeding the total amount of machine memory. Soon after booting, the VMs start executing their application benchmarks, the amount of shared memory drops rapidly, and ESX Server compensates by using ballooning to reclaim memory. Page sharing continues to exploit sharing opportunities over the run, saving approximately 200 MB.

Figures 8(c) and 8(d) show the same memory allocation data for the Citrix MetaFrame Server VM and the Microsoft SQL Server VM, respectively. The MetaFrame Server allocation tracks its active memory usage, and also grows slowly over time as page sharing reduces overall memory pressure. The SQL Server allocation starts high as it processes queries, then drops to 160 MB as it idles, the lower bound imposed by its *min* size. When a long-running query is issued later, its active memory increases rapidly, and memory is quickly reallocated from other VMs.

7 I/O Page Remapping

Modern IA-32 processors support a physical address extension (PAE) mode that allows the hardware to address up to 64 GB of memory with 36-bit addresses [13]. However, many devices that use DMA for I/O transfers can address only a subset of this memory. For example, some network interface cards with 32-bit PCI interfaces can address only the lowest 4 GB of memory.

Some high-end systems provide hardware support that can be used to remap memory for data transfers using a separate I/O MMU. More commonly, support for I/O involving "high" memory above the 4 GB boundary involves copying the data through a temporary *bounce buffer* in "low" memory. Unfortunately, copying can impose significant overhead resulting in increased latency, reduced throughput, or increased CPU load.

This problem is exacerbated by virtualization, since even pages from virtual machines configured with less than 4 GB of "physical" memory may be mapped to machine pages residing in high memory. Fortunately, this same level of indirection in the virtualized memory system can be exploited to transparently remap guest pages between high and low memory.

[7] This is the peak amount of memory paged to disk over the entire run. To avoid clutter, paging metrics were omitted from the graphs; the amount of data swapped to disk was less than 20 MB for the remainder of the experiment.

ESX Server maintains statistics to track "hot" pages in high memory that are involved in repeated I/O operations. For example, a software cache of physical-to-machine page mappings (PPN-to-MPN) associated with network transmits is augmented to count the number of times each page has been copied. When the count exceeds a specified threshold, the page is transparently remapped into low memory. This scheme has proved very effective with guest operating systems that use a limited number of pages as network buffers. For some network-intensive workloads, the number of pages copied is reduced by several orders of magnitude.

The decision to remap a page into low memory increases the demand for low pages, which may become a scarce resource. It may be desirable to remap some low pages into high memory, in order to free up sufficient low pages for remapping I/O pages that are currently "hot." We are currently exploring various techniques, ranging from simple random replacement to adaptive approaches based on cost-benefit tradeoffs.

8 Related Work

Virtual machines have been used in numerous research projects [3, 7, 8, 9] and commercial products [20, 23] over the past several decades. ESX Server was inspired by recent work on Disco [3] and Cellular Disco [9], which virtualized shared-memory multiprocessor servers to run multiple instances of IRIX.

ESX Server uses many of the same virtualization techniques as other VMware products. One key distinction is that VMware Workstation uses a hosted architecture for maximum portability across diverse desktop systems [23], while ESX Server manages server hardware directly for complete control over resource management and improved I/O performance.

Many of the mechanisms and policies we developed were motivated by the need to run existing commodity operating systems without any modifications. This enables ESX Server to run proprietary operating systems such as Microsoft Windows and standard distributions of open-source systems such as Linux.

Ballooning implicitly coaxes a guest OS into reclaiming memory using its own native page replacement algorithms. It has some similarity to the "self-paging" technique used in the Nemesis system [11], which requires applications to handle their own virtual memory operations, including revocation. However, few applications are capable of making their own page replacement decisions, and applications must be modified to participate in

an explicit revocation protocol. In contrast, guest operating systems already implement page replacement algorithms and are oblivious to ballooning details. Since they operate at different levels, ballooning and self-paging could be used together, allowing applications to make their own decisions in response to reclamation requests that originate at a much higher level.

Content-based page sharing was directly influenced by the transparent page sharing work in Disco [3]. However, the content-based approach used in ESX Server avoids the need to modify, hook, or even understand guest OS code. It also exploits many opportunities for sharing missed by both Disco and the standard copy-on-write techniques used in conventional operating systems.

IBM's MXT memory compression technology [27], which achieves substantial memory savings on server workloads, provided additional motivation for page sharing. Although this hardware approach eliminates redundancy at a sub-page granularity, its gains from compression of large zero-filled regions and other patterns can also be achieved via page sharing.

ESX Server exploits the ability to transparently remap "physical" pages for both page sharing and I/O page remapping. Disco employed similar techniques for replication and migration to improve locality and fault containment in NUMA multiprocessors [3, 9]. In general, page remapping is a well-known approach that is commonly used to change virtual-to-physical mappings in systems that do not have an extra level of "virtualized physical" addressing. For example, remapping and page coloring have been used to improve cache performance and isolation [17, 19, 21].

The ESX Server mechanism for working-set estimation is related to earlier uses of page faults to maintain per-page reference bits in software on architectures lacking direct hardware support [2]. However, we combine this technique with a unique statistical sampling approach. Instead of tracking references to all pages individually, an aggregate estimate of idleness is computed by sampling a small subset.

Our allocation algorithm extends previous research on proportional-share allocation of space-shared resources [31, 32]. The introduction of a "tax" on idle memory solves a significant known problem with pure share-based approaches [25], enabling efficient memory utilization while still maintaining share-based isolation. The use of economic metaphors is also related to more explicit market-based approaches designed to facilitate decentralized application-level optimization [12].

9 Conclusions

We have presented the core mechanisms and policies used to manage memory resources in ESX Server [29], a commercially-available product. Our contributions include several novel techniques and algorithms for allocating memory across virtual machines running unmodified commodity operating systems.

A new ballooning technique reclaims memory from a VM by implicitly causing the guest OS to invoke its own memory management routines. An idle memory tax was introduced to solve an open problem in share-based management of space-shared resources, enabling both performance isolation and efficient memory utilization. Idleness is measured via a statistical working set estimator. Content-based transparent page sharing exploits sharing opportunities within and between VMs without any guest OS involvement. Page remapping is also leveraged to reduce I/O copying overheads in large-memory systems. A higher-level dynamic reallocation policy coordinates these diverse techniques to efficiently support virtual machine workloads that overcommit memory.

We are currently exploring a variety of issues related to memory management in virtual machine systems. Transparent page remapping can be exploited to improve locality and fault containment on NUMA hardware [3, 9] and to manage the allocation of cache memory to VMs by controlling page colors [17]. Additional work is focused on higher-level grouping abstractions [30, 31], multi-resource tradeoffs [24, 31], and adaptive feedback-driven workload management techniques [5].

Acknowledgements

Mahesh Patil designed and implemented the ESX Server paging mechanism. Mike Nelson and Kinshuk Govil helped develop the hot I/O page remapping technique. Vikram Makhija and Mahesh Patil provided invaluable assistance with quantitative experiments. We would also like to thank Keith Adams, Ole Agesen, Jennifer Anderson, Ed Bugnion, Andrew Lambeth, Beng-Hong Lim, Tim Mann, Michael Mullany, Paige Parsons, Mendel Rosenblum, Jeff Slusher, John Spragens, Pratap Subrahmanyam, Chandu Thekkath, Satyam Vaghani, Debby Wallach, John Zedlewski, and the anonymous reviewers for their comments and helpful suggestions.

References

[1] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. "Information and Control in Gray-Box Systems," *Proc. Symposium on Operating System Principles*, October 2001.

[2] Ozalp Babaoglu and William Joy. "Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits," *Proc. Symposium on Operating System Principles*, December 1981.

[3] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *ACM Transactions on Computer Systems*, 15(4), November 1997.

[4] Pei Cao, Edward W. Felten, and Kai Li. "Implementation and Performance of Application-Controlled File Caching," *Proc. Symposium on Operating System Design and Implementation*, November 1994.

[5] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, and Amin M. Vahdat. "Managing Energy and Server Resources in Hosting Centers," *Proc. Symposium on Operating System Principles*, October 2001.

[6] R. J. Creasy. "The Origin of the VM/370 Time-Sharing System," *IBM Journal of Research and Development*, 25(5), September 1981.

[7] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back, and Stephen Clawson. "Microkernels Meet Recursive Virtual Machines," *Proc. Symposium on Operating System Design and Implementation*, October 1996.

[8] Robert P. Goldberg. "Survey of Virtual Machine Research," *IEEE Computer*, 7(6), June 1974.

[9] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. "Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors," *Proc. Symposium on Operating System Principles*, December 1999.

[10] Peter H. Gum. "System/370 Extended Architecture: Facilities for Virtual Machines," *IBM Journal of Research and Development'*, 27(6), November 1983.

[11] Steven M. Hand. "Self-Paging in the Nemesis Operating System," *Proc. Symposium on Operating Systems Design and Implementation*, February 1999.

[12] Kieran Harty and David R. Cheriton. "Application-Controlled Physical Memory using External Page-Cache Management," *Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[13] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual. Volumes I, II, and III*, 2001.

[14] Bob Jenkins. "Algorithm Alley," *Dr. Dobbs Journal*, September 1997. Source code available from `http://burtleburtle.net/bob/hash/`

[15] Ted Kaehler. "Virtual Memory for an Object-Oriented Language," *Byte*, August 1981.

[16] Minkyong Kim and Brian Noble. "Mobile Network Estimation," *Proc. Seventh Annual International Conference on Mobile Computing and Networking*, July 2001.

[17] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. "OS-Controlled Cache Predictability for Real-Time Systems," *Proc. Third IEEE Real-Time Technology and Applications Symposium*, June 1997.

[18] Marshall K. McKusick, Keith Bostic, Michael J. Karels, and John S. Quaterman. *The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley, 1996.

[19] Theodore H. Romer, Dennis Lee, Brian N. Bershad and J. Bradley Chen. "Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware," *Proc. Symposium on Operating System Design and Implementation*, November 1994.

[20] L. H. Seawright and R. A. McKinnon. "VM/370: A Study of Multiplicity and Usefulness," *IBM Systems Journal*, 18(1), 1979.

[21] Timothy Sherwood, Brad Calder and Joel S. Emer. "Reducing Cache Misses Using Hardware and Software Page Placement," *Proc. International Conference on Supercomputing*, June 1999.

[22] David A. Solomon and Mark Russinovich. *Inside Microsoft Windows 2000*, Third Ed., Microsoft Press, 2001.

[23] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," *Proc. Usenix Annual Technical Conference*, June 2001.

[24] David G. Sullivan, Robert Haas, and Margo I. Seltzer. "Tickets and Currencies Revisited: Extensions to Multi-Resource Lottery Scheduling," *Proc. Seventh Workshop on Hot Topics in Operating Systems*, March 1999.

[25] David G. Sullivan and Margo I. Seltzer. "Isolation with Flexibility: A Resource Management Framework for Central Servers," *Proc. Usenix Annual Technical Conference*, June 2000.

[26] Andrew S. Tanenbaum. *Modern Operating Systems*, Prentice Hall, 1992.

[27] R. Tremaine, P. Franaszek, J. Robinson, C. Schulz, T. Smith, M. Wazlowski, and P. Bland. "IBM Memory Expansion Technology (MXT)," *IBM Journal of Research and Development'*, 45(2), March 2001.

[28] Andrew Tridgell. "dbench" benchmark. Available from `ftp://samba.org/pub/tridge/dbench/`, September 2001.

[29] VMware, Inc. *VMware ESX Server User's Manual Version 1.5*, Palo Alto, CA, April 2002.

[30] Carl A. Waldspurger and William E. Weihl. "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. Symposium on Operating System Design and Implementation*, November 1994.

[31] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*, Ph.D. thesis, Technical Report MIT/LCS/TR-667, September 1995.

[32] Carl A. Waldspurger and William E. Weihl. "An Object-Oriented Framework for Modular Resource Management," *Proc. Fifth Workshop on Object-Orientation in Operating Systems*, October 1996.