Study of Content-Based Sharing on the Xen Virtual Machine Monitor

Selvamuthukumar Senthilvelan, and Murugappan Senthilvelan

University of Wisconsin, Madison

ABSTRACT

Renewed interest in Virtual Machines (VM) due to inexpensive hardware has triggered numerous application areas such as server consolidation and management, testing and debugging of system software, rapid application deployment, data isolation, etc. It has been observed that there is a potential to save system memory by cutting down on redundant copies of code and data existing across multiple VMs running on top of a single physical machine. This paper investigates a transparent memory sharing mechanism called Content-Based sharing on the Xen Virtual Machine Monitor. The memory virtualization and the input/output (I/O) mechanisms of Xen have been studied in detail and discussed. The mechanism to identify memory pages that can be shared has been implemented and studied. Preliminary results indicate a good potential to share memory pages across domains running similar guest OSes.

1. INTRODUCTION

The concept of virtual machines (VM) was first developed in the 1960's as a part of Operating Systems research. A virtual machine was defined as a piece of software that emulates computer hardware. IBM pioneered most of the research on virtual machines to use the technology on their mainframe computers. The virtual machine was used to simulate multiple copies of the underlying mainframe hardware and run different operating systems and applications in each copy. These mainframe machines were used to build large custom applications which were tedious to design and build. The concept slowly faded in the 1980's due to lack of active research and lack of new application areas.

Interestingly, in the 1990's the virtual machine concept had a rebirth. The concept that was so far restricted to the mainframe computing domain transcended that barrier and spilled over into the desktop computing domain. The focus of business computing shifted from mainframe computers to PCs. Personal computer technology has been improving by leaps and bounds leading to affordable, reliable and more powerful hardware. Virtual machines were no longer constrained by prohibitive price of hardware, especially with the advancement and availability of affordable desktop memory.

1.1 Applications of Virtual Machines

The advent of virtual machines into the desktop domain suddenly sparked renewed interest and further research in this area, which triggered the use of this technology in areas never seen before. Virtual machine technology has found applications in diverse fields such as testing and debugging of system software, server consolidation and management, teaching system administration, networking and security, rapid application deployment, data isolation, etc.

In server consolidation, several lightly loaded servers can be bundled up using virtual machines and loaded on a single physical computer as shown in figure 1. This reduces the cost incurred on buying separate hardware for every server.

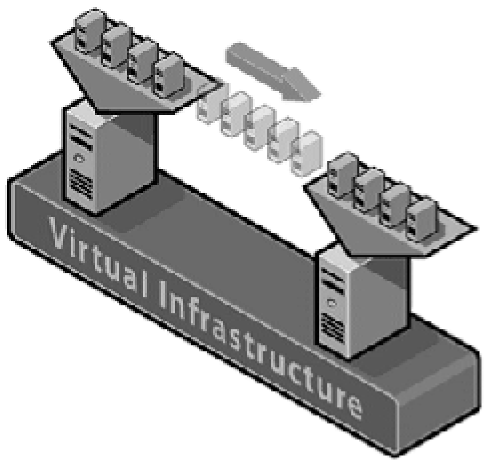

Figure1: Virtual Machine used for Server Consolidation and Migration [www.vmware.com]

Further to provide server maintenance or load balancing, certain virtual machines can be suspended, transferred across physical computer and later resumed without interruption. This helps to improve the availability of the server and also do regular maintenance on the computers without shutting down the servers running on virtual machines.

1.2 Implementing Server Consolidation

The major applications of virtual machines: server consolidation and creating test environments; require a system to host multiple applications and servers typically running on the same type of operating system on a shared machine. There are a number of ways to build such a system. A simplest and naive way of doing it would be to deploy one or more hosts running a standard operating system such as Linux or Windows, and then allow the users to install files and start processes – protection between applications being provided by conventional OS techniques. Experience has shown that in such type of systems, system administration becomes a time-consuming task due to complex configuration interactions between supposedly disjoint applications.

More importantly, such systems do not adequately support performance isolation; the scheduling priority, memory demand, network traffic and disk accesses of one process impact the performance of others. This may be acceptable when there is adequate provisioning and a closed user group, but not when resources are oversubscribed, or users uncooperative.

One way to address this problem is to retrofit support for performance isolation to the operating system. This has been demonstrated with approaches such as resource containers [4]. One difficulty with such a approach is ensuring that all resource usage is accounted to the correct process - consider, for example, the complex interactions between applications due to buffer cache or page replacement algorithms. Performing multiplexing of resources at a low level can mitigate this problem, as demonstrated by the Exokernel [5]. Unintentional or undesired interactions between tasks are minimized.

Current day virtual machines utilize this basic approach to multiplex physical resources at the granularity of an entire operating system and are able to provide performance isolation between applications running on top of them. In contrast to process-level multiplexing this also allows a range of guest operating systems to gracefully coexist. This approach also allows individual users to run unmodified binaries, or collections of binaries, in a resource controlled fashion. Furthermore it provides an extremely high level of flexibility since the user can dynamically create the precise execution environment their software requires. Unfortunate configuration interactions between various services and applications are avoided.

1.3 Overhead of running virtual machines

There is a price to pay for the flexibility of virtual machines mentioned above - running a full OS is more heavyweight than running a process, both in terms of performance and in terms of resource consumption. The Virtual Machine Monitor (VMM) which controls the VMs also adds overhead as it occupies memory and other resources [1]. Physical memory, one of the most precious resources, of a system running similar guest OSes on top a VMM is poorly utilized because of redundant content.

Such systems present numerous opportunities for sharing memory between the guest OSes. For example, several VMs may be running instances of the same guest OS, have the same applications or components loaded, or contain common data. Exploiting these sharing opportunities would enable server workloads running in VMs to consume less memory than they would when run on separate physical machines. As a result, higher levels of over-commitment can be supported efficiently.

1.4 Memory Sharing between domains

This paper studies the feasibility of implementing a transparent memory sharing mechanism called the Content-Based Sharing [6] on the Xen Virtual Machine Monitor [7]. A proactive policy for implementing cross domain (an instance of the guest OS running on top of the VMM) memory sharing has been investigated.

This technique intercepts every block I/O request from the guest OS running on the VMM. After the requested memory page has been fetched from the disk, it computes a hash on the contents of the fetched memory page and compares it with pre-computed hashes of pages that are already present in the memory. If there is a hash match, the contents of the pages are compared byte-by-byte to ensure an exact match and then the memory page is shared between the two domains. When sharing of memory pages takes place between two domains, the page is marked as a READONLY page and if either one of the domains requests a write on the shared page, a Copy-on-Write operation is performed and the requesting domain is provided with a writable copy of the page.

The mechanism to identify memory pages that can be shared has been implemented. The Copy-on-Write mechanism has not been implemented. Preliminary results indicate a good potential to share memory pages across domains running similar guest OSes.

Section 2 discusses the Xen VMM on which Content-Based Sharing (described in section 5) has been studied. Section 3 talks about memory subsystem virtualization and section 4 talks about Block I/O subsystem virtualization. Section 6 briefly outlines related work in this area and section 7 describes our approach / implementation. Section 8 discusses the results obtained and the challenges faced and section 9 provides the conclusions.

2. The Xen Virtual Machine Monitor

Xen is a paravirtualizing based virtual machine monitor, or 'hypervisor', for the x86 processor architecture. Xen was originally developed by the Systems Research Group at the University of Cambridge Computer Laboratory. Xen can efficiently partition a single physical system and execute multiple virtual machines, with close-to-native performance on these partitions. It allows each VM to run its own operating system and applications in an environment that provides protection, resource isolation and accounting.

2.1 Paravirtualization

In a traditional VMM the virtual hardware exposed is functionally identical to the underlying machine. This full virtualization allows the use of unmodified operating systems to be hosted on top of the VMM. Full virtualization on the x86 architecture can only be achieved at the cost of increased complexity and reduced performance. For example, effectively virtualizing the x86 MMU is very difficult and will result in reduced performance. To avoid the complexity, Xen presents a virtual machine abstraction that is similar but not identical to the underlying hardware. This approach called paravirtualization, promises improved performance although it requires modifications done to the guest operating system. Table 1 presents an overview of the paravirtualized x86 interface, for memory management, and device I/O of the system.

2.2 Structure of the Xen VMM

As shown in figure 2, a Xen system has multiple layers, the lowest and most privileged of which is Xen itself (called the hypervisor). Xen strives to separate policy from mechanism wherever possible. The resulting architecture is one in which the Xen hypervisor provides only basic control operations. These functions are exported and accessible through authorized domains. Xen in turn may host multiple *guest* operating systems, each of which is executed within a secure virtual machine (in Xen terminology, a *domain*). Domains are scheduled by Xen to make

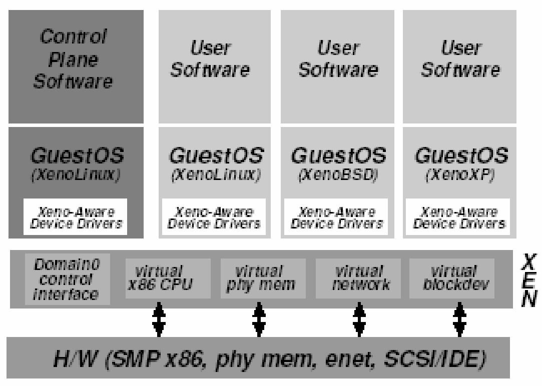

Figure 2: Structure of a machine running Xen

Memory Management	Segmentation	Cannot install fully-privileged segment descriptors and cannot overlap with the top end of the linear address space.
	Paging	Guest OS has direct read access to hardware page tables, but updates are batched and validated by the hypervisor. A domain may be allocated discontiguous machine pages.
Device I/O	Network, Disk, etc.	Virtual devices are elegant and simple to access. Data is transferred using asynchronous I/O rings. An event mechanism replaces hardware interrupts for notifications.

Table1: The paravirtualized x86 interface provided by Xen [3]

effective use of the available physical CPUs. Each guest OS manages its own applications, which includes responsibility for scheduling each application within the time allotted to the VM by Xen.

The first domain, *domain 0*, is created automatically when the system boots and has special management privileges. Domain 0 builds other domains and manages their virtual devices. It also performs administrative tasks such as suspending, resuming and migrating other virtual machines.

2.3 Control transfer mechanism in Xen

Two mechanisms exist for control interactions between Xen and an overlying domain: synchronous calls from a domain to Xen may be made using a “hypercall”, while notifications are delivered to domains from Xen using an asynchronous “event” mechanism.

The hypercall interface allows domains to perform a synchronous software trap into the hypervisor to perform a privileged operation, analogous to the use of system calls in conventional operating systems. An example of a hypercall is to request a set of pagetable updates, in which Xen validates and applies a list of updates, returning control to the calling domain when this is completed.

Communication from Xen to a domain is provided through an asynchronous event mechanism, which replaces the usual delivery mechanisms for device interrupts and allows lightweight notification of important events such as domain-termination requests. Similar to traditional Unix signals, there are only a small number of events, each acting to flag a particular type of occurrence. For instance, events are used to indicate that new data has been received over the network, or that a virtual disk request has completed.

2.4 Data transfer mechanism in Xen

The presence of a hypervisor means there is an additional protection domain between guest OSes and I/O devices, so it is crucial that a data transfer mechanism be provided that allows data to move vertically through the system with as little overhead as possible. The shared memory committed to device I/O is provided by the relevant domains wherever possible to prevent the crosstalk inherent in shared buffer pools; I/O buffers are protected during data transfer by pinning the underlying page frames within Xen.

Figure 3 shows the structure of I/O descriptor rings. A ring is a circular queue of descriptors allocated by a domain but accessible from within Xen. Descriptors do not directly contain I/O data; instead, I/O data buffers are allocated out-of-band by the guest OS and indirectly referenced by I/O descriptors. Access to each ring is based around two pairs of producer-consumer pointers: domains place requests on a ring, advancing a request producer pointer, and Xen removes these requests for handling, advancing an associated request consumer pointer. Responses are placed back on the ring similarly. There is no requirement that requests be processed in order: the guest OS associates a unique identifier with each request which is reproduced in the associated response. This allows Xen to unambiguously reorder I/O operations due to scheduling or priority considerations.

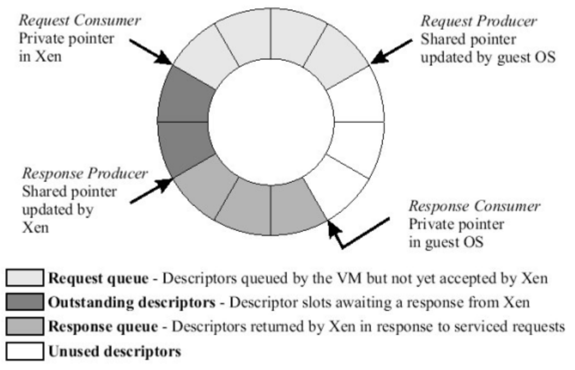
Request Consumer
Private pointer in Xen
Request Producer
Shared pointer updated by guest OS
Response Producer
Shared pointer updated by Xen
Response Consumer
Private pointer in guest OS
Request queue - Descriptors queued by the VM but not yet accepted by Xen
Outstanding descriptors - Descriptor slots awaiting a response from Xen
Response queue - Descriptors returned by Xen in response to serviced requests
Unused descriptors

Figure 3: The structure of asynchronous I/O rings which are used for data transfer between Xen and the guest OSes.

This structure is sufficiently generic to support a number of different devices. For example, a set of `requests' can provide buffers for network packet reception; subsequent `responses' then signal the arrival of packets into these buffers. Reordering is useful when dealing with disk requests as it allows them to be scheduled within Xen for efficiency, and the use of descriptors with out-of-band buffers makes implementing zero-copy transfer easy.

3. Memory subsystem Virtualization

Xen is responsible for managing the allocation of physical memory to domains, and for ensuring safe use of the paging and segmentation hardware.

3.1 Memory Allocation

Xen resides within a small fixed portion of physical memory; it also reserves the top 64MB of the virtual address space for every domain. The remaining physical memory is available for allocation to domains at a page granularity. Xen tracks the ownership and use of each page, which allows it to enforce secure partitioning between domains. Each domain has a maximum and current physical memory allocation. A guest OS may run a `balloon driver' to dynamically adjust its current memory allocation up to its limit [7].

3.2 Pseudo-Physical Memory

Since physical memory is allocated and freed on a page granularity, there is no guarantee that a domain will receive a contiguous stretch of physical memory as shown in figure 4. However, most operating systems do not have good support for operating in a fragmented physical address space. To aid porting such operating systems to run on top of Xen, distinction is made between machine memory and pseudo-physical memory.

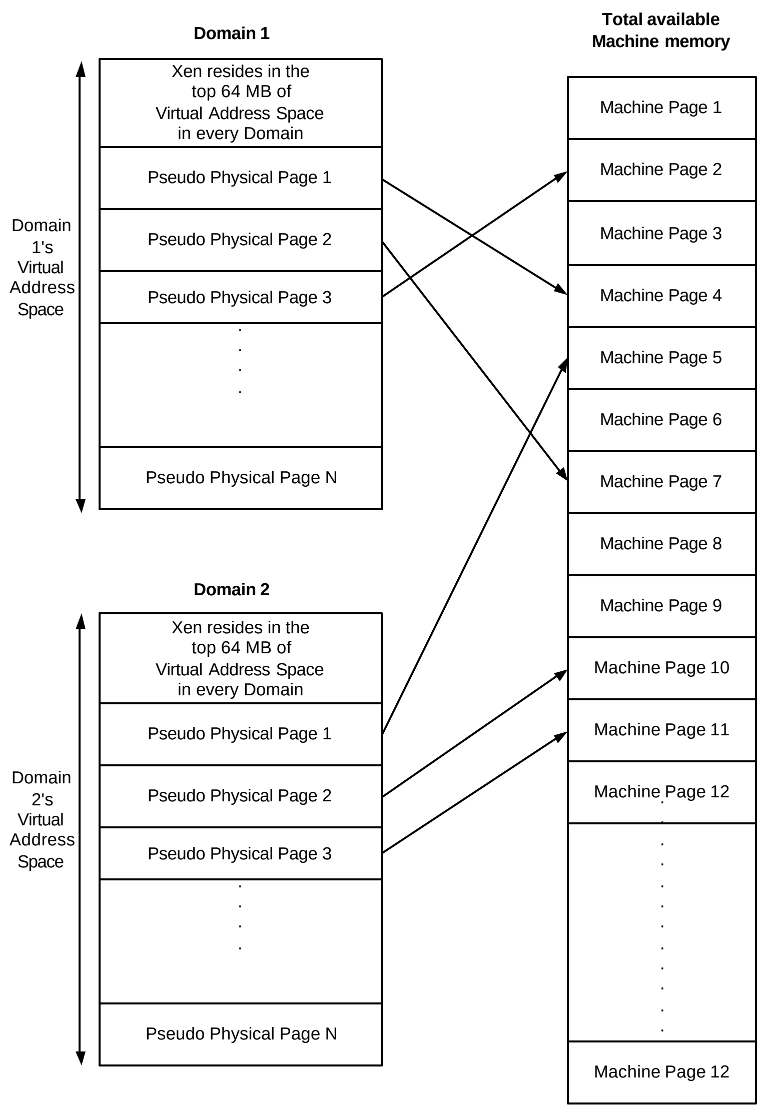

Figure 4: Memory mapping in Xen

Machine memory refers to the entire amount of memory installed in the machine, including that reserved by Xen. The machine memory comprises of a set of 4K machine page frames numbered consecutively starting from 0. Machine frame numbers are invariant within Xen or any domain.

Pseudo-physical memory, on the other hand, is a per-domain abstraction. It allows a guest operating system to consider its memory allocation to consist of a contiguous range of physical page frames starting at physical frame 0, despite the fact that the underlying machine page frames may be sparsely allocated and in any order.

To achieve this, Xen maintains a globally readable machine-to-physical table (figure 5) which records the mapping from machine page frames to pseudo-physical ones. Clearly the machine-to-physical table has size proportional to the amount of memory installed in the machine.

In addition to the normal per domain page table (figure 5), each domain is supplied with a physical-to-machine table (figure 5) which performs the inverse mapping from physical address to the machine address and a grant table to perform sharing of machine pages across domains. The page table and the physical-to-machine table have sizes proportional to the memory allocation of the given domain.

Architecture dependent code in guest operating systems can then use the machine-to-physical and physical-to-machine tables to provide the abstraction of pseudo-physical memory. In general, only certain specialized parts of the operating system (such as page table management) need to understand the difference between machine and pseudo-physical addresses.

3.3 Page Table Updates

In the default mode of operation, Xen enforces read-only access to page tables and requires guest operating systems to explicitly request any modifications via hypercalls. Xen validates all such requests and only applies updates that it deems safe. This is necessary to prevent domains from adding arbitrary mappings to their page tables. To minimize the number of hypercalls required, guest OSes can locally queue updates before applying an entire batch with a single hypercall (multicall) - this is particularly beneficial when creating new address spaces.

To aid validation, Xen associates a type and reference count with each memo ry page. A page has one of the following mutually-exclusive types at any point in time: page directory (PD), page table (PT), local descriptor table (LDT), global descriptor table (GDT), or writable (RW). A guest OS can create readable mappings of its own memory regardless of its current type. This mechanism is used to maintain the invariants required for safety; for example, a domain cannot have a writable mapping to any part of a page table as this would require the page concerned to simultaneously be of types PT and RW.

The type system is also used to track which frames have already been validated for use in page tables. When the guest OSes requests a frame to be allocated for page-table use Xen validates every entry in the frame. The page type is then pinned to PD or PT as appropriate, until a subsequent unpin request from the guest OS. This eliminates the need to validate the new page table on every context switch. A frame cannot be re-tasked until it is both unpinned and its reference count has reduced to zero.

Global Table:

Machine-to-Physical table - Read-Only access to all domains, modified through Hypercalls

Machine Page number	Pseudo-Physical Page Number

Domain Specific Tables:

Page Table - Read-Only access to all domains, modified through Hypercalls

Virtual Page number	Machine Page Number	12 bit flag field

Physical-to-Machine table - managed by each domain

Pseudo-Physical Page Number	Machine Page number

Grant table - Managed by each domain and modifiable by Xen Hypervisor

Domain ID	Machine Page Number	Flag for Read-Only Sharing	Flag to indicate sharing active/not

Figure 5: Data structures used in Xen to perform memory virtualization

4. Block I/O subsystem Virtualization

Only Domain0 has direct unchecked access to physical (IDE and SCSI) disks. All other domains access persistent storage through the abstraction of virtual block devices (VBDs), which are created and configured by management software running within Domain0. Allowing Domain0 to manage the VBDs keeps the mechanisms within Xen very simple.

The VBD is a split driver interface with the driver backend residing in Domain 0 (privileged domain), while the driver frontend is a part of every guest OS. The backend driver interfaces with the standard Linux block device drivers to perform block I/O. The frontend and backend of the VBD driver communicate through message passing over an event channel interface (described in section 2.3). A request/response message is sent over the event channel once the respective descriptors are placed in the shared memory ring. The sequence of a block I/O request from a guest OS is described in figure 6.

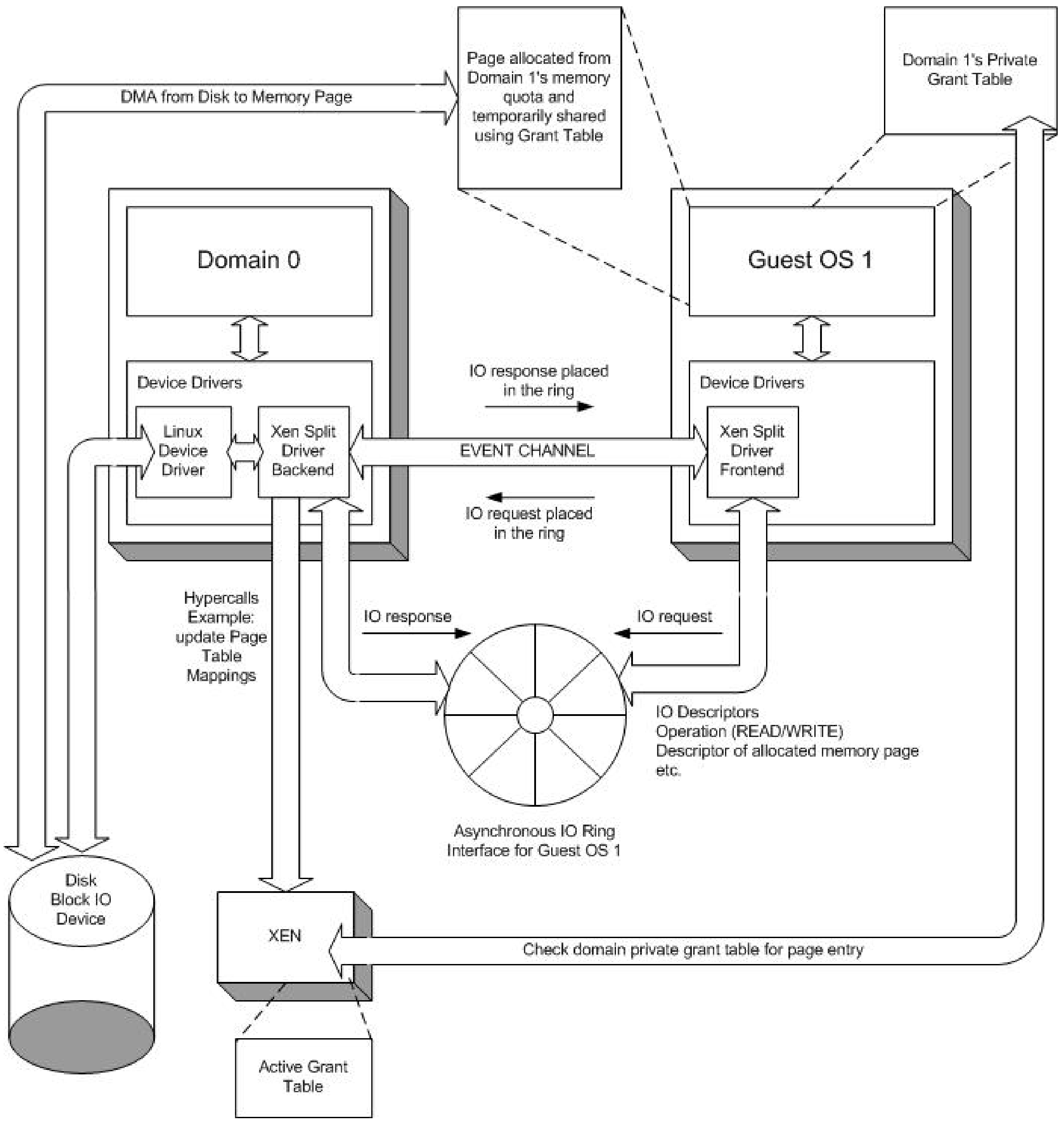
Page allocated from Domain 1's memory quota and temporarily shared using Grant Table
Domain 1's Private Grant Table
DMA from Disk to Memory Page
Domain 0
Guest OS 1
Device Drivers
Device Drivers
Linux Device Driver
Xen Split Driver Backend
Xen Split Driver Frontend
IO response placed in the ring
EVENT CHANNEL
IO request placed in the ring
Hypercalls Example: update Page Table Mappings
IO response
IO request
IO Descriptors Operation (READ/WRITE) Descriptor of allocated memory page etc.
Asynchronous IO Ring Interface for Guest OS 1
Disk Block IO Device
XEN
Check domain private grant table for page entry
Active Grant Table

Figure 6: Sequence of a block I/O request from a guest OS

Any block device accessible to the backend domain can be exported as a VBD interface to the guest OS. Each VBD is mapped to a device node in the guest OS, specified in the guest's startup configuration. Setting up of a VBD interface involves the initialization of the event channel and I/O ring mechanisms for communications between the split driver interface. When an application running on the guest OS requests a block I/O, the guest OS allocates a memory page from its virtual address space. Since Domain0 performs the actual data transfer, it should be capable of mapping this memory page into its virtual address space. Hence the guest OS uses the grant table mechanism to provide WRITE access to domain0 for the allocated memory page. Once the grant table entry is in place, the guest OS prompts the VBD frontend driver to place the I/O descriptors (all information required to perform the block I/O; eg. operation, address of allocated memory page, etc.) in the I/O ring and send a request message to the VBD backend driver over the event channel.

Once the backend driver receives the request from the frontend, the first step is to map the allocated memory page into Domain0's virtual address space. To accomplish this, the backend driver performs a hypercall into Xen which checks the requesting domain's grant table to validate the sharing. Once validated, an entry is made in the Active Grant table within Xen which enables Domain0 to map the allocated memory page.

A translation table is maintained within the hypervisor for each VBD; the entries within this table are installed and managed by Domain0 via a privileged control interface. Xen inspects the VBD identifier and offset and produces the corresponding sector address and physical device. Permission checks also take place at this time. The control is then passed on to the traditional linux block device drivers to perform a Zero-copy data transfer using DMA between the disk and pinned memory pages in the requesting domain. Once the data transfer is completed, the backend driver places the response descriptors in the I/O ring and intimates the frontend driver that the request has been completed.

Xen services batches of requests from competing domains in a simple round-robin fashion; these are then passed to a standard elevator scheduler before reaching the disk hardware. The low-level scheduling gives good throughput, while the batching of requests provides reasonably fair access.

5. Content-Based Sharing Mechanism

Page contents can be used to identify page copies present in the memory. Pages with identical contents can be shared regardless of when, where, or how those contents were generated. This approach simplifies the mechanism for identifying opportunities for sharing memory pages without modifying or understanding guest OS code. Content-Based sharing (figure 7) can be implemented proactively or lazily.

In the proactive approach every disk request is intercepted and checked for a sharing opportunity. This exploits maximum possible sharing but results in performance degradation since it is on the critical path of servicing a block I/O request. In the lazy approach idle cycles of the CPU are utilized to identify sharing by intercepting block I/O requests. This technique has no performance degradation but does not yield optimum sharing.

A naive way of identifying sharing opportunities is by direct comparison of the contents of each page with every other page in the system which would be prohibitively expensive and would require O(n^2)page comparisons. Instead, hashing is used to identify pages with potentially-identical contents efficiently.

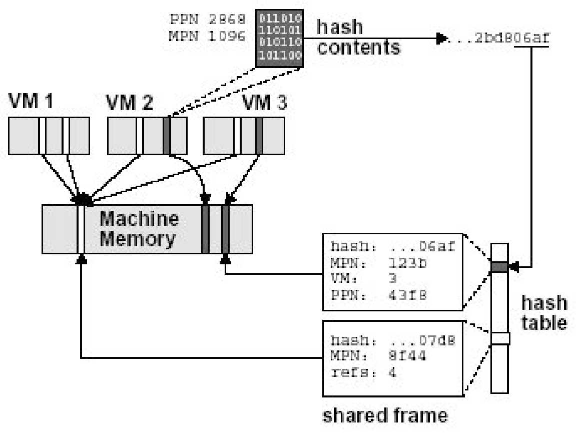

Figure 7: Content-Based Page Sharing mechanism scans for sharing opportunities, hashing the contents of candidate PPN 0x2868 in VM 2. The hash is used to index into a table containing other scanned pages, where a match is found associated with PPN 0x43f8 in VM 3. If a full comparison confirms the pages 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 0x1096 is reclaimed.

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 are already present in the memory. 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 due to hash aliasing. A successful match is followed by a byte-by-byte comparison of the page contents to verify that the pages are identical.

Once a match has been found with an existing 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 for the computed hash, store the computed hash and the corresponding page number in the hash table in anticipation of some future match. If a match was found in the hash table for the hash but the byte-by-byte comparison failed (page contents differ), then the information about the new page can either be discarded or chained in the hash table for determining future matches.

6. Related Work

Disco [8] implemented a transparent page sharing mechanism for eliminating redundant copies of pages across virtual machines. Once copies of code or read-only data were identified, multiple guest physical pages were mapped to the same machine page. Copy-on-write was performed to create a private writable copy of the shared page for the requesting domain. The downside was that to implement this page sharing mechanism, several guest OS modifications were required.

VMware introduced the Content-based sharing mechanism in the ESX server. The sharing mechanism was implemented without any changes to the guest OS since the design philosophy of the ESX server did not permit modifications to the guest OS and changes to the application programming interface. The sharing mechanism implemented was lazy type content-based sharing which would identify sharing opportunities by scanning the memory during CPU idle cycles.

7. Approach / Implementation

The approach taken by us to implement content-based sharing is as follows. The first step was to intercept the VBD request from the requesting VBD frontend driver (blkback.c). Once this request has been intercepted, follow this block I/O request to completion and compute the hash on the contents of the requested block. Use the computed hash to index into the hash table to find a matching entry. If no matching entry was found, insert the new information into the hash table. If a matching entry was found, compare the contents of the two pages byte-by-byte to ascertain an exact match. If the contents do not match, the new page is a hash alias and is discarded. If the contents of the two pages match exactly, the sharing count for that hash entry is incremented in the hash table. An entry is placed in the grant table of the domain that owns the initial machine page with which the new page matched and the page table entry for that page is marked as READONLY and the COPY_ON_WRITE bit (UNUSED1 flag bit) set. The requesting domain now maps the shared machine page into its virtual address space with the READONLY bit and the COPY_ON_WRITE bit set. The unused machine page allocated by the requested domain is reclaimed. This completes the sharing process. When either one of the sharing domains tries to perform a write on this shared page, this would cause a page fault and would be handled by the page fault handler within Xen (fault.c). If a page fault was triggered by a write protection, and the COPY_ON_WRITE bit for the page was set, then allocate a machine page and make a private writable copy of the shared data to the requesting domain. Update the sharing count in the hash table as needed. If the sharing count is zero, update the corresponding page table entry of the other domain to reset the READONLY bit.

The implementation was performed on a Pentium III machine with 512 MB RAM running Red Hat Linux 8.0. Xen was compiled and installed from the source distribution. Multiple user domains were created running Tiny Linux as the guest OS. A proactive approach to content-based sharing was chosen to be implemented as it is has the potential to exploit the maximum sharing possible and is comparatively straight forward to implement as CPU idle cycles need not be tracked.

The block request from unprivileged domains was intercepted and followed till completion. Upon completion, the block hash was computed and comparison done in the hash table to identify a hash match. On a hash match, a byte-by-byte comparison of the page contents was performed to determine an exact match. A system wide counter to track the number of shared pages was implemented to study the amount of page sharing. The modified code was compiled and tested. Due to the complexity involved in implementing the complete sharing mechanism and the time constraints of the course project, the copy-on-write mechanism was not implemented.

8. Results and Challenges

The memory sharing results were observed by varying the number of Tiny Linux domains running simultaneously on top of Xen. There was no memory sharing observed when multiple Tiny Linux domains were booted up. We believe this might be due to the reason that Tiny Linux is just a minimal Linux kernel image with not many traditional components loaded on to it. We believe that better sharing would be observed when running full versions of commodity operating systems. A simple shell script to access files from a folder was run on different domains and sharing among domains was observed but different runs produced indeterministic values for each runs. We believe this indeterminism arises due to the complexity of the memory virtualization.

Some of the challenges that we faced while working on this project are as follows: (i) We encountered problems with the grub configurations while installing Xen. (ii) Xen being a research project, it was lacking in documentation which posed a great challenge for us to even understand the big picture. Through systematically reading and understanding code and discussions with people who have worked on Xen, we believe we have a better understanding of the block I/O mechanism in Xen. We have done our best to convey what we have learnt through pictorial representation in this paper. (iii) The location of the Xen VBD split drivers partly in the domain0 code and partly within Xen made understanding the control flow tough. The inherent asynchrony in operating system code also posed a similar challenge. (iv) The Copy-on-write functionality was not implemented because we could not figure out the correct way to reclaim the extra machine page while initiating a sharing and allocating it back on a copy-on-write operation.

9. Conclusion

Our results indicate sharing of memory pages across domains even when running very simple shell commands on top of minimal guest OS. Hence we believe, content-based sharing mechanism would provide adequate memory savings while implemented on a Xen system running commodity guest operating systems and real world applications. To achieve a flawless implementation of content-based sharing, finer details of memory virtualization in Xen need to be understood. The memory virtualization and the I/O mechanisms of Xen have been studied in detail and discussed in this paper.

10. References

[1] M. Rosenblum, *The Reincarnation of Virtual Machines*, ACM Queue, 2(5), 34-40, Jul/Aug 2004.

[2] A. Singh, *An Introduction to Virtualization*, [Online]. Available: http://www.kernelthread.com/publications/virtualization/

[3] *Microsoft Virtual PC 2004 Evalutation Guide*, [Online]. Available: http://www.microsoft.com/windows/virtualpc/evaluation/evalguide.mspx, Nov 2003.

[4] G. Banga, P. Druschel, and J. C. Mogul. *Resource containers: A new facility for resource management in server systems.* In the Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI 1999), pages 45-58, Feb. 1999.

[5] M. F. Kaashoek, D. R. Engler, G. R. Granger, H. M. Briceòno, R. Hunt, D. Mazi_eres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. *Application performance and _exibility on Exokernel systems.* In Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles, volume 31(5) of ACM Operating Systems Review, pages 52-65, Oct. 1997.

[6] C. A. Waldspurger. *Memory resource management in VMware ESX server.* In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002), ACM Operating Systems Review, Winter 2002 Special Issue, pages 181ñ194, Boston, MA, USA, Dec. 2002.

[7] P. Barham, B. Dragovic, K. Fraser, S. Had, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield. *Xen and the Art of Virtualization.* Proceedings of the nineteenth ACM symposium on Operating systems principles, pp 164-177, Bolton Landing, NY, USA, 2003

[8] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," ACM Transactions on Computer Systems, 15(4), Nov1997.

[9] Documentation for the Xen Virtual Machine Monitor [online] Available: http://www.cl.cam.ac.uk/Research/SRG/netos/xen/documentation.html