VIRTUALIZATION OF NON-VOLATILE RAM

A Thesis

by

AYUSH RUIA

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	A. L. N. Reddy
Co-Chair of Committee,	Riccardo Bettati
Committee Member,	P. R. Kumar
Head of Department,	Miroslav M. Begovic

May 2015

Major Subject: Computer Engineering

ABSTRACT

Virtualization technology is powering today's cloud industry. Virtualization inserts a software layer, the hypervisor, below the Operating System, to manage multiple OS environments simultaneously. Offering numerous benefits such as fault isolation, load balancing, faster server provisioning, etc., virtualization occupies a dominant position, especially in IT infrastructure in datacenters. Memory management is one of the core components of a hypervisor. Current implementations assume the underlying memory to be homogenous and volatile. However, with the emergence of NVRAM in the form of Storage Class Memory (SCM), this assumption remains no longer valid. New motherboard architectures will support several different memory classes each with distinct properties and characteristics. The hypervisor has to recognize, manage, and expose them separately to the different virtual machines. This study focuses on building a separate memory management module for Non-Volatile RAM in Xen hypervisor. We show that it can be efficiently implemented with a few code changes and minimal runtime performance overhead.

DEDICATION

To my family

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Prof. Reddy for his guidance and support during my graduate studies. This project was his brainchild, and he gave me this excellent opportunity to work on it and learn, despite my past inexperience in the subject area. My time at Texas A&M Univeristy has been very fruitful and enjoyable, largely due to his endless patience and compassion. He stood as a shield protecting me from various adversities, arising more often than not, from my own immature decisions. Prof. Reddy takes really good care of his students, protecting their academic as well as future professional interests, and I can never thank him enough. It is a matter of great pride and honour for me to complete my Master's thesis under his guidance.

This is also an excellent opportunity to thank our academic advisor, Ms. Tammy Carda, for her countless favors extended to every student. She has extensive knowledge of the system rules and regulations, encompassing various departments such as ISS, OGS, etc. and makes seemingly impossible tasks look easy. On several occasions, she went out of the way to help me and sort out my troubles, when the concerned department had given up. Be it any problem or issue concerning the life of a graduate student, when presented to her, it disappears straigtaway. The most surprising part is that, even after helping so many students, she is always available and only an email away. It is my good fortune to have her as an academic advisor in our department.

I would also like to thank Gim Jongmin, Viacheslav Fedorov and Allen Webb for lending their technical expertise to this project. Jongmin, especially, gave many critical feedbacks and identified several potential problems early on, resulting in a better and mature design.

Finally, I want to thank to my parents and my brother, who have constantly provided me with guidance, love and support, my entire life. I recognize the numerous sacrifices they make everyday for my sake, and hope that I can fill their life with joy and happiness as they have done so for me.

NOMENCLATURE

NV	Non-Volatile
API	Application Program Interface
VA	Virtual Address
PA	Physical Address
HDD	Hard Disk Drive
SDD	Solid State Drive
DRAM	Dynamic Random Access Memory
ACPI	Advanced Configuration and Power Interface
PML4	Page Map Level 4
GDT	Global Descriptor Table
LDT	Local Descriptor Table
NVM	Non-Volatile Memory
ELF	Executable and Linkable Format
SDK	Software Development Kit
IMC	Integrated Memory Controller

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

1.1 Motivation

Proliferation of smartphones and tablets is introducing a divide in the computer industry. While mobile technology is burgeoning in the role of access points, computationally intensive tasks are offloaded to the cloud [6]. As a result, the server industry is growing at a tremendous pace. This has also led to the development of associated technologies, the most prominent being virtualization [23].

Virtualization enables multiple operating system environments to run simultaneously on one hardware platform. It provides added security and isolation in the form of an additional software layer below the OS (Operating System), called the hypervisor [25]. This technology has become an industry standard for large server farms. Fault isolation, centralized control, workload balance, and live migration of machines are few of its many benefits [23].

Most virtualized systems today, are constrained by memory and I/O subsystems [24] [31] [20], with CPU resources to spare. However, these limitations shall no longer stay relevant with the upcoming radical changes in memory technologies. In the past few decades, memory subsystem structure has been fairly consistent, with regular upgrades in its size. However, new developments such as 3D memory stacks [27] [19], Storage Class Memory [7] [14], etc., shall completely revolutionize memory architectures.

Among these new technologies, Non-Volatile RAM (Random Access Memory) in the form of SCM (Storage Class Memory), has the potential to overhaul memory architecture in virtual machines. Introduction of non-volatile memory can reduce the frequency of disk I/O operations, and diminish the memory footprint of disk caches. It is highly likely that persistent memory will play a dominant role in future server farms. Thus, it is a natural and worthwhile initiative to inspect the possibility of sharing such a resource in virtual machines, with significant opportunities for performance gains in filesystems, boot procedures, crash recovery, etc [2] [3]. This thesis is going to focus on some of these aspects. To the best of our knowledge, there has been no prior work in this direction, and this work may be considered a first step in enabling NVRAM (Non-Volatile RAM) sharing in virtualized environments

1.2 Overview

This work focuses on sharing non-volatile memory in virtualized environments. Due to several contrasting properties between volatile and non-volatile memory, a conventional Memory Management Unit (MMU) cannot be used to manage non-volatile memory. Therefore, a novel design of an additional MMU is proposed to facilitate non-volatile memory sharing. This thesis also showcases an implementation of the non-volatile MMU in a popular open source hypervisor – Xen.

In Chapter 1, we offer a brief technical background of virtualization and memory subsystem in a modern computer. Chapter 2 is dedicated to the internal architecture of Xen, detailing its sub-components and various design philosophies. Chapter 3 features the design and implementation details of the proposed non-volatile MMU in Xen. In Chapter 4, we present the specifics of the experimental setup along with the results. Lastly, in Chapter 5, the merits and demerits of the system are discussed with emphasis on future work.

1.3 Virtualization

Modern day computers are a combination of complex software and hardware systems. Such a high level of engineering is made possible through concepts of abstraction, where an upper layer interacts with the layer beneath it, using well defined interfaces, oblivious to the lower layer's inner implementation complexity and details.

A computer system can be viewed as a stack of several independent layers as shown in Fig 1.1. Here the flexibility of each layer is constrained by the interfaces defined both above and below it.

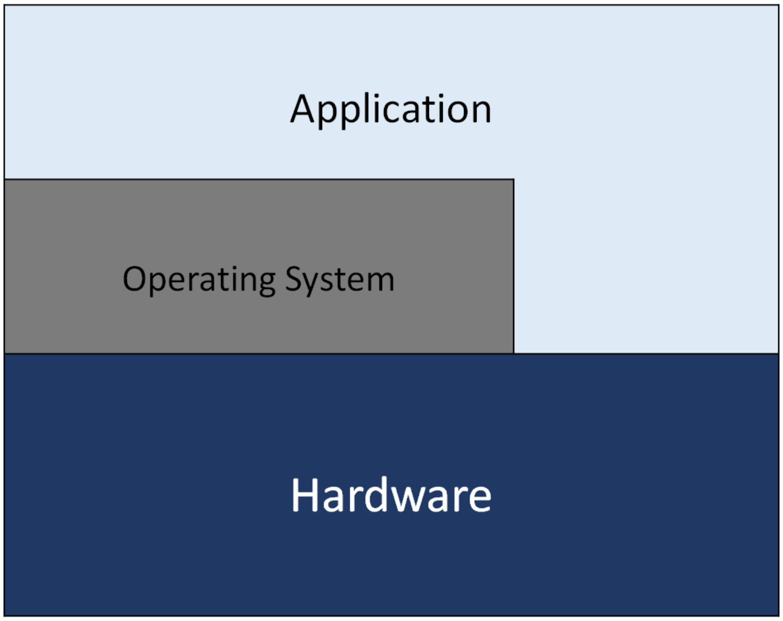


Figure 1.1: Conventional Application Stack

Virtualization provides a way to relax the above constraints, either in the form of the entire system or a subsystem like memory, I/O, etc. It enables the mapping of a virtual system, to real system resources thereby giving an illusion to the process/OS of a custom virtual environment, different from the host machine [25].

Formally, virtualization may be defined as a mapping between a guest state (S_i) and a host state (S_i'), such that a sequence of operations, e_k, modifying the guest state from (S_i) to (S_j), can be represented by some corresponding sequence of operations,

e'_k, which modifies the host state from (S'_i) to (S'_j) respectively [26] (Fig 1.2).

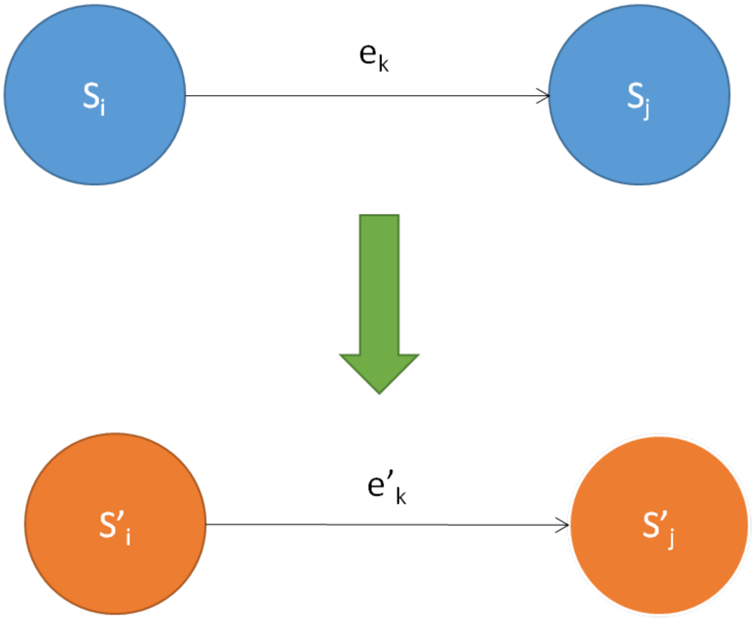


Figure 1.2: Equivalent State Mapping [26]

From an OS's point of view, all hardware can be classified into three broad categories (Fig 1.3).

1. CPU: A CPU is a highly complex piece of hardware abstracted for the OS in the form of an Instruction Set Architecture (ISA). The ISA defines the actual hardware software interface in a machine, converting software code into electrical signals that percolate through the entire system. Every action performed by the software stack (including controlling Memory and I/O), takes place through the available set of ISA instructions.

2. Memory: Memory is a collection of byte-addressable memory elements that can be used to store and retrieve data. Every ISA provides special instructions to interact with memory.

3. I/O: All devices apart from CPU and Memory, such as modem, printer, monitor, etc., come under the class of I/O devices. These devices are essentially composed of several specialized registers which can be programmed to perform device specific instructions. Thus, from the CPUs perspective, there is not much of a difference between I/O and memory, since both are a collection of byte addressable memory elements. Interactions with an I/O device can be performed by either (optional) special I/O instructions or standard memory instructions supported by the ISA. Due to the vast variety of devices available from different manufacturers, individual device drivers have to be developed and added to the OS separately.

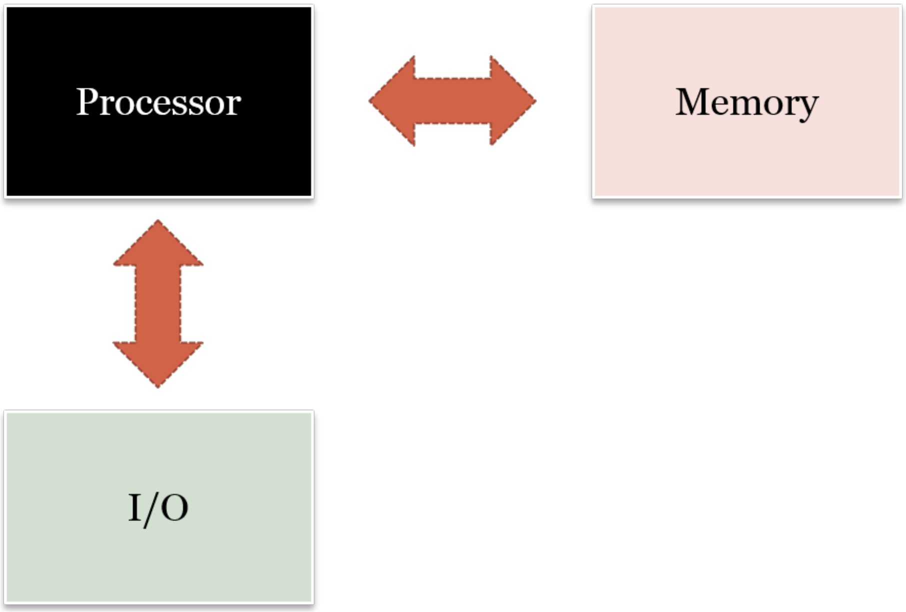


Figure 1.3: Abstract Computer Model

For entire system virtualization, we have to virtualize all the subsystems. This can be performed either in software, or by hardware. A software approach provides higher flexibility at the cost of reduced performance when compared to a hardware solution. The following subsections delve further into each subsystem virtualization specifics.

1.3.1 CPU Virtualization

CPU virtualization aims at providing the hypervisor direct control over the distribution of hardware resources amongst various VMs (Virtual Machines). Similar work is performed at the application layer by the Operating System. To facilitate such protection mechanisms, a typical ISA implements several privilege levels (protection rings), allowing a certain class of instructions to execute in a privileged mode only. The ISA generally presents two levels – user level and supervisor level (Fig 1.4). An attempt to execute a privileged instruction in an unprivileged mode triggers an exception. It transfers control of execution to a specific supervisor level subroutine (generally registered with the OS), which takes appropriate action maintaining the security and protection of the system.

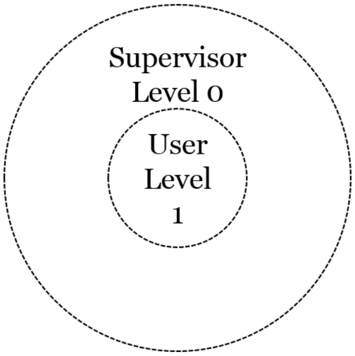


Figure 1.4: ISA Protection Rings

In the case of virtual machines, such a task becomes all the more challenging, as these protection mechanisms have to be implemented at the OS level by the hypervisor. Pioneering work done by Popek and Goldberg [21], defined several constraints on the Instruction Set Architecture of a machine to provide efficient virtualization where majority of the operations run natively on the CPU. The CPU instructions are first classified in the following manner:

1. Privileged Instructions: The group of instructions that can only be run when the CPU is in supervisor mode and will trap outside it.

2. Control Sensitive Instructions: Instructions that change the hardware configuration or resources of the system.

3. Behavior Sensitive Instructions: Any instruction whose output depends on the current state or configuration of the machine.

They proposed that for any architecture to be efficiently virtualizable (in trap and emulate model), all sensitive (behavior and control) instructions must be privileged instructions. Any instruction, which either tries to modify hardware configuration or whose output depends upon it, should transfer control of execution to the hypervisor.

Contrary to the norm, an operating system on a VM runs in de-privileged user mode (Fig 1.5). Most of the operations run at native speeds without emulation, with a penalty introduced for sensitive instructions which trap into the hypervisor.

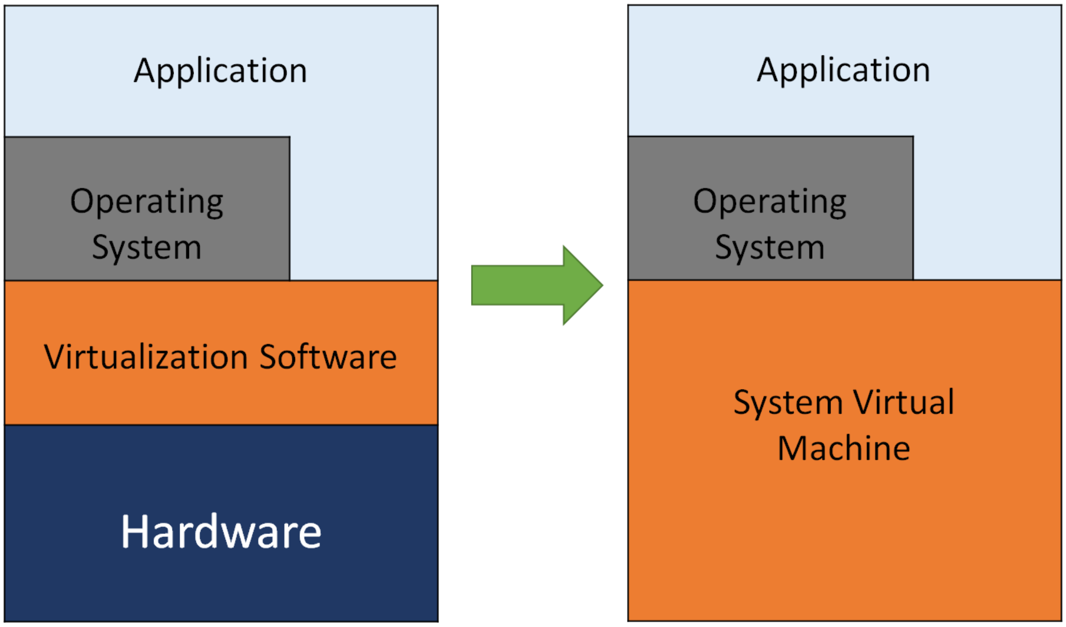


Figure 1.5: System Virtualization Model

According to the above definition, x86 is not a virtualizable architecture. It has a set of 17 sensitive instructions that do not trap to the supervisor mode. Being a predominant architecture in todays computers, significant efforts have been spent producing several solutions.

1. Emulation is the most versatile solution, implemented entirely in software. Here, the dynamic instruction stream is scanned for sensitive instructions, which are then replaced by emulated operations. Emulation can also allow a code, compiled for one ISA, to run on a host machine with a different ISA; though with a severe performance penalty (since every instruction has to be emulated). Binary translation may be viewed as an optimized version of the above, where emulated code segments are cached aggressively, providing significant performance boost for re-entrant code segments.

2. Paravirtualization is another solution, that relies on relaxing some tenets of

virtualization by modifying the source code of operating system to replace sensitive operations with hypercalls to the hypervisor. Unfortunately this trades off flexibility with performance, as different Operating Systems have to be modified individually and the OS source code may not always be available to the hypervisor. Xen is one of the leading open-source hypervisors employing paravirtualization [4], now natively supported by the Linux kernel (from Linux kernel 3.0 onwards).

3. With increasing demand for virtualization technology both Intel and AMD have extended the x86 ISA to include extra features to support a hypervisor in an additional ring at -1 level, as shown in Fig 1.6. As per the original requirements of Popek and Goldberg, the OS executing in the ring 0 is oblivious to the presence of the hypervisor, with the privileged instructions generating a trap to the hypervisor. Additional level of memory virtualization is also introduced with the addition of Extended Page Tables in hardware. Along with the above, several instructions were added to the ISA to support a system call structure for the hypervisor, named hypercalls. It also provides a hardware concept of virtual CPUs with specific instructions to store/restore the state to/from VMCS (Virtual Machine Control Structure). This enables x86 to achieve the status of a virtualizable architecture with the support of these extensions. Detailed explanation is provided in Section 1.5.

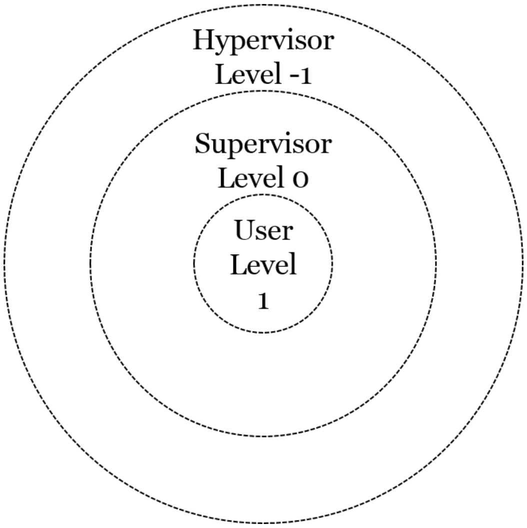


Figure 1.6: Hypervisor Protection Rings

1.3.2 Memory Virtualization

Virtual Memory has been around for a long time, allowing multiple applications to share the physical memory in the system. Each application is given a virtual address space, that is mapped on to the available physical address space via page tables maintained by the Operating System (Fig 1.7). Thus, applications do not have to worry about runtime memory allocation, and can operate on a continuous address space. On the downside, each memory reference now requires an address translation through the page table structure. Many ISAs provide support for a hardware page table walker, which performs the translation in hardware.

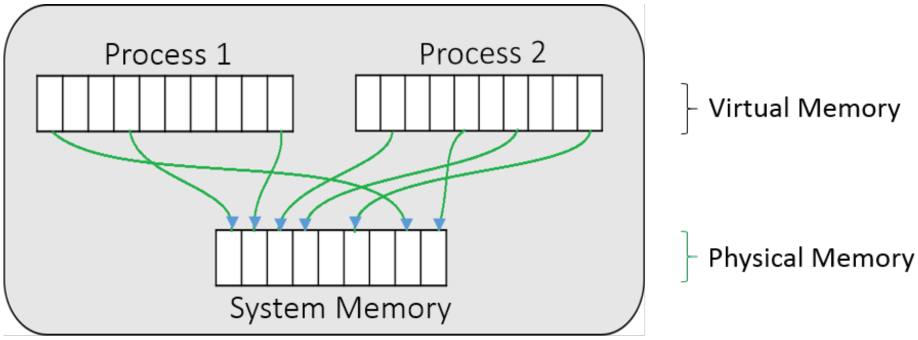


Figure 1.7: Virtual Memory

In a virtualized environment, the system memory is shared among several guest VMs. Thus, it leads to an additional address space.

1. Virtual address space: The address space as visible to applications.
2. Guest Physical address space: Individual VM Level or OS Level address space.
3. Real or Machine address space: The actual system memory address space.

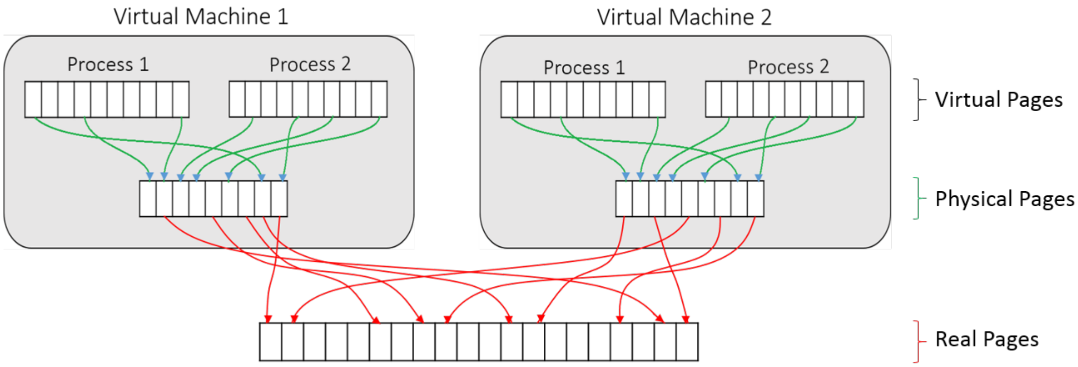


Figure 1.8: Hypervisor Based Memory Layers

Translations from virtual address space to the machine address space require two levels of paging (Fig 1.8):

1. Operating System Page table: Translates from virtual to guest physical addresses.

2. Hypervisor Page table: Translates from guest physical to machine addresses.

Without additional hardware support, a clever software based solution is to maintain an additional shadow page table with the hypervisor, mapping virtual addresses directly to machine addresses. This approach though avoids one level of paging, causes frequent traps to the hypervisor, which are very expensive.

However, recent virtualization extensions added to the x86 architecture now support nested page tables, i.e. two levels of address translations in hardware. It provides far superior performance in comparison with the shadow page table approach. In the event of a page fault due to an invalid entry in the guest OS page tables, the hypervisor need not be involved. However, in the software based approach, the hypervisor first catches the exception and forwards the event to the guest kernel for an appropriate response. It results in two unnecessary context switches, which can be avoided with nested page table support, where the exception is directly delivered to the guest. Moreover, any update to the guest page tables in the former approach have to go through hypervisor, either via hypercalls or exceptions. As evident, the shadow page table based approach generates more context switches. On the downside, in the hardware assisted nested page table approach a TLB (Translation Lookaside Buffer) miss is quite expensive, as the hardware has to traverse a greater number of page tables to resolve an address.

Frequent context switching is another common issue faced in virtualized systems. To reduce its impact, the virtualization extensions provide tagged TLB entries, where TLB entries are associated with a vCPU (virtual CPU) using a tag. Thus, every context switch does not necessitate a complete TLB flush. It may be safely assumed from the above that hardware extensions add a significant boost to most virtualized environments, especially for memory management.

1.3.3 I/O Virtualization

An I/O operation is typically controlled by programming special registers on the device, and using DMA (Direct Memory Access) for data transfers. Since, device speeds are extremely slow (relative to the CPU) it is best not to involve the CPU for data transfers. Thus, DMA requests are moderated entirely by the northbridge chipset (Fig 1.11). The CPU is notified of the completion of the DMA operation, using an interrupt. Device virtualization mainly focuses on the following two aspects.

1. DMA
2. Interrupts

There are several ways to approach this problem. A direct method would be for the hypervisor to manage all the devices directly, and then emulate them for each Virtual Machine. It is a humungous undertaking requiring re-development of device drivers separately for the hypervisor. An alternate and currently popular way is to assign devices to specific privileged domains, which in turn, handle the I/O requests of all the other domains. Herein arises a security issue, where DMA requests from the driver domain can corrupt memory regions of other virtual machines. Traditionally there exists no security checks in hardware to prevent devices from accessing memory regions. As a result, the VM controlling a device, can potentially gain access to the entire machine address space.

Newer extensions such as VT-d (from Intel [17]) can safeguard against such situations. VT-d implements IOMMU (I/O Memory Management Unit) that act as an MMU for devices. The hypervisor can program page tables in the IOMMU mapping the device to the guest physical address space of a particular VM, maintaining memory isolation for the other guests. Moreover, VT-d also includes functionality for remapping interrupts to a particular virtual CPU construct. It helps minimize costly context switches and hypervisor interventions.

Newer ambitious approaches similar to PCI-e SR-IOV technology (Single Root I/O Virtualization) [12], define certain standards for a device to divide itself into smaller units. These smaller device units can then be directly controlled by individual VMs without hypervisor intervention. The hypervisor maintains control of the higher level functionality of the device, such as allocation of units to different VMs. Unfortunately, due to the vast number of devices and manufacturers involved, industry has not come to a common consensus, and the above approach is still in the nascent stages of development.

1.4 Non-Volatile Memory

Traditionally a computer has two forms of data storage.

1. Main memory or RAM.
2. Secondary storage or Disk.

The key property of main memory is byte addressable data, while that of secondary storage is non-volatility – data is preserved in the absence of power. Traditionally main memory has been volatile, while disk storage lacks byte-addressability. Moreover, disk storage is significantly slower than RAM, but has notable greater capacity (Table 1.1). These factors have led RAM to serve as a cache for movement of data to and from disk.

Table 1.1: Comparison between HDD, SSD and DRAM (approx. values)

Property	HDD	SSD	DRAM
Maximum Capacity	6TB	512GB	16GB
Data Retention	>10 yrs	10 yrs	64ms
Write Endurance	Unlimited	10000/block	Unlimited
Read Latency	3ms	20us	50ns
Write Latency	3ms	200us	50ns
Cost ($/GB)	0.05	0.5	10
Write Bandwidth	150MB/s	500MB/s	10GB/s

The volatility of RAM leads to data loss on shutdown. In the case of a planned shutdown, data residing on RAM is backed up on the disk. The system state is restored on boot up by transferring all saved data back to RAM, generating an illusion of persistence of data. However, the heart of the problem lies in the case of unplanned power failures where all data present in the main memory is permanently lost. Several approaches can be taken to safeguard against such a situation:

1. Software solution: – This method cannot eliminate data losses but tries to minimize the effective data loss. Here, the OS maintains complex data structures along with logging and check-pointing procedures to periodically transfer data to disk. Since, disk speeds are significantly slower, heavy performance penalty is observed. The frequency of the above mentioned procedures involves a constant trade-off between performance and data integrity, where one has to be sacrificed for the other. This issue is exacerbated in server farms, where client data safety is of utmost importance.

2. Un-interruptible power source: – This solution comes at steep infrastructure

costs of a backup power source, that lies idle for the most part.

3. Hardware solution: – Non-volatile RAM is a proposed hardware solution to the above mentioned issues. It combines near RAM access speeds with data retentive technologies to provide performance, as well as data integrity. Data movement on the memory bus is several orders of magnitude faster when compared to disk, providing persistence at little or no additional cost.

Out of these three possibilities, the software solution is most commonly applied to current systems, due to the absence of an inexpensive alternative. Whereas, NVRAM technology, though in its nascent stages, is the most promising solution in the future systems. In conclusion, Non-Volatile memory is useful to safeguard against data loss during unexpected power cuts.

1.4.1 Storage Class Memory

Research and innovation into devices have allowed a new class of memory technologies to spring up under the banner of Storage Class Memory that fall in the NVRAM category. Some common examples are PCM (Phase change memory) [22], STT-RAM (Spin Transfer Torque RAM) [13], ReRAM (Resistive RAM) [8] [9], FRAM (Ferroelectric RAM) [5], MRAM (Magnetic RAM) [16] [11], etc. Each of these devices have different properties in terms of power efficiency, speed, density, and cost/bit, but are unified by the following common characteristics.

1. Byte Addressable Memory
2. Non-volatility
3. Significantly faster access times when compared to disks or SSDs.

Introduction of SCM will usher some changes in the motherboard architecture (Fig 1.9). SCM memory can be placed alongside DRAM on the memory bus, and also on a PCIe (Peripheral Component Interconnect Express) bus along with SSDs. With the growth and commercialization of different SCM technologies, SCM is expected to replace/augment both DRAM and Flash memory from their dominant roles in the near future.

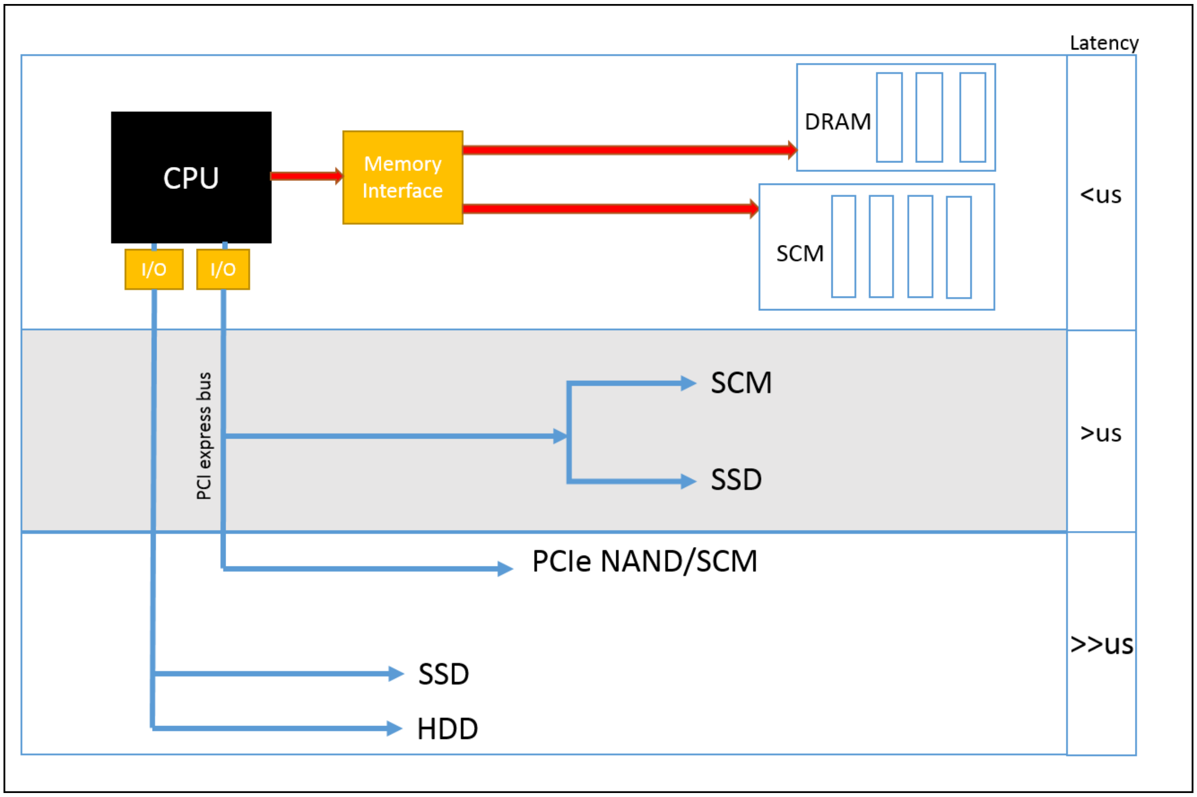


Figure 1.9: SCM Based Motherboard Architectures [30]

1.4.2 NVDIMM

Non-volatile Dual Inline Memory Module (NVDIMM) [15] is another upcoming technology under the banner of non-volatile RAM. As shown in Fig 1.10, it is simply a conventional DRAM backed up by Flash memory. During normal operation, all the write and read requests go to DRAM with the flash device remaining inactive. However, in the event of loss of power (either during normal shutdown or an unexpected power failure), a supercapacitor/battery kicks in to provide alternate power for a short time period. During this time, dedicated hardware logic transfers all the data from DRAM to flash memory, thereby making it non-volatile. The power on procedure restores the state of DRAM from the data backed up in the flash memory. NVDIMMs thus behave exactly like DRAM during runtime, only exposing the flash memory in the reboot sequence.

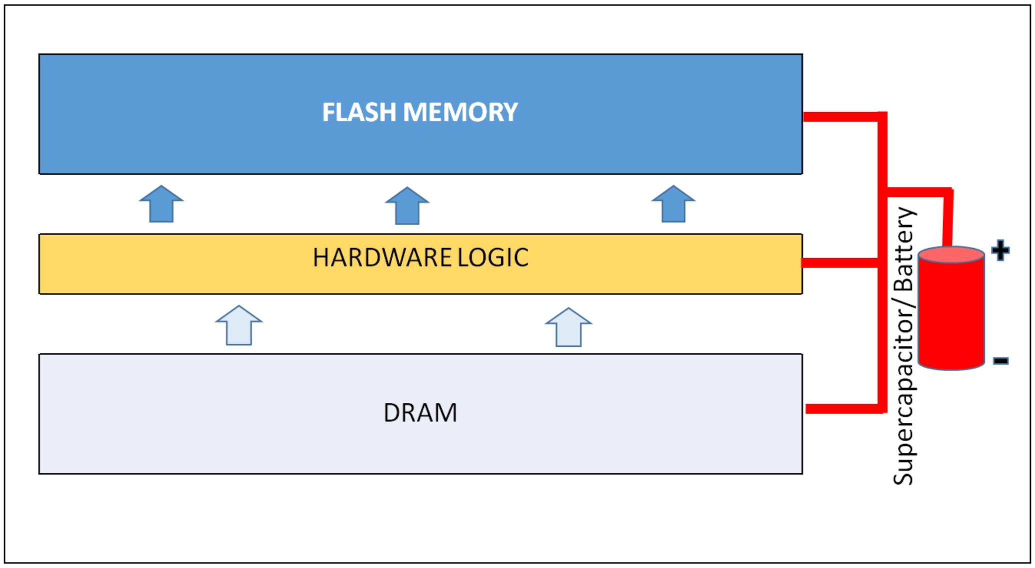


Figure 1.10: NVDIMM Model

One of the key advantages is that NVDIMMs can function at DRAM speeds while providing non-volatility through background operations. Another important benefit is that both DRAM and flash are commercially mature technologies, combined simply by some hardware glue logic.

1.5 E820 Memory Map

Devices external to the microprocessor can be broadly classified into two groups

1. I/O

2. Memory

The microprocessor interacts with these two devices in a similar way. I/O devices contain programmable registers, that act as an interface for the device, whereas memory is just a bank of byte addressable memory elements. Both memory and I/O read/write instructions are issued on the same bus, which are then forwarded appropriately by the Northbridge chipset (Fig 1.11).

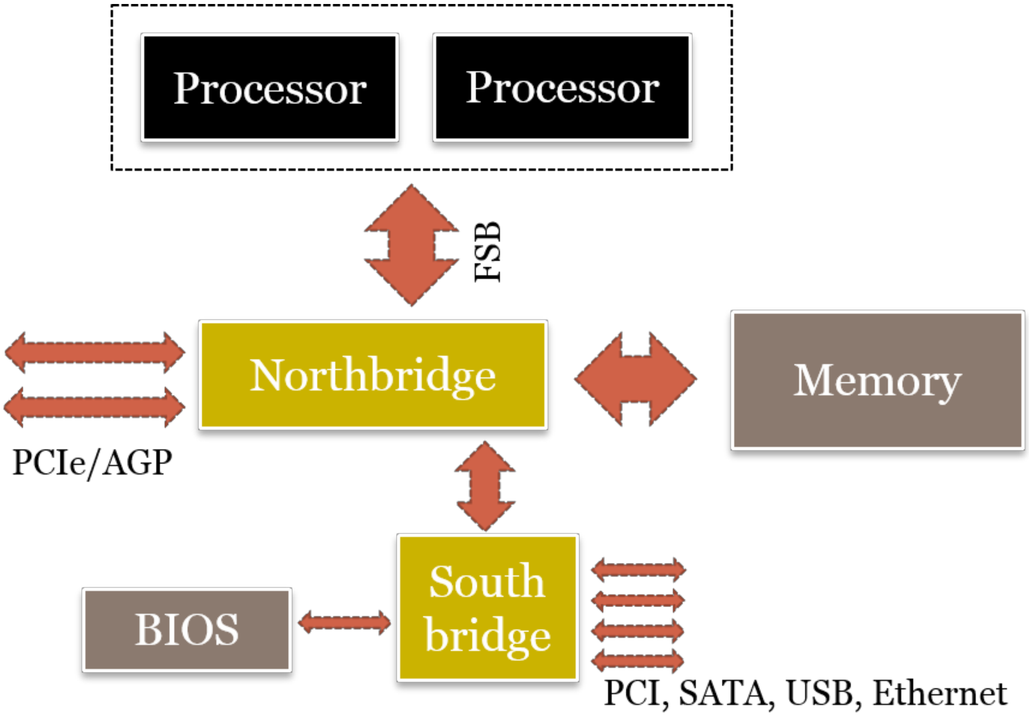


Figure 1.11: Intel Hub Architecture

In the nascent stages of microprocessor development, different companies went with different I/O models. Intel included a separate I/O pin in its microprocessor that separated memory and I/O addresses into two separate address spaces, effectively providing an extra bit. For example, if the ISA supported 16 bit addresses, we get one 16 bit address space for I/O and another 16 bit address for memory, which is the same as a 17 bit unified address space for both memory and I/O. Intel also had to provide a separate set of I/O instructions to manipulate I/O registers (called I/O ports) in its ISA. This address space segregation through an external I/O pin simplified the address decoding logic in Northbridge chipset. Moreover, then, addresses were just 16 bits wide, making the effective extra bit, a precious resource addition. On the downside, the ISA became more bulky with separate I/O instructions performing similar tasks as Memory instructions.

On the other hand, Motorola provided a Memory Mapped I/O (MMIO) model. Here I/O registers and memory are mapped on to the same address space, each occupying distinct addresses.

This eliminated the need for any separate I/O instructions to be included in the ISA. However, as a result, the address decoding logic in the Northbridge chipset became more complex. Where, in the previous case, the chipset just had to check the value of one bit (I/O pin of the microprocessor), here, it had to decode the entire address, to forward the instruction to the correct bus. Moreover, this design had to share the address space between both memory and I/O.

In the long run, many of the disadvantages of MMIO mentioned above, disappeared, making it the dominant model used in today's computers. With scaling of transistor technology, hardware logic became inexpensive, and the Northbridge could easily handle the additional hardware for complete address decode logic. The address space also expanded from 16 bits to 32 bits and now on to 64 bits, which is more than enough for both memory and I/O. This model had a major advantage of preventing duplication of instructions at the ISA level. The vast plethora of memory management instructions are more comprehensive and can be used in the same way for I/O registers. Thus MMIO is more popular, with I/O ports, only supported for legacy reasons.

This begs the question now, that how does the OS know the division of the address space between memory and I/O. The translation occurs at Northbridge and differs from one motherboard to another. These hardware details are hidden in an abstraction layer provided by BIOS (Basic Input/Output System), in the form of E820 Memory Map [28], as shown in Table 1.2. This list is generated by BIOS on raising a software interrupt 0x15, with EBX set to 0xE820.

Table 1.2: Sample E820 Table

Start Address	End Address	Code	Translation
00000000	0009e800	1	(usable)
0009e800	000a0000	2	(reserved)
000e0000	00100000	2	(reserved)
00100000	7270a000	1	(usable)
7270a000	72822000	4	(ACPI NVS)
72822000	72a24000	1	(usable)
73561000	73577000	1	(usable)
73577000	74177000	2	(reserved)
74177000	741f3000	3	(ACPI data)

Physical address ranges are currently divided into 6 regions which are indicated by the type field [28].

A brief description of the different types is as follows:

1. AddressRangeMemory: Available RAM

2. AddressRangeReserved: Reserved by the system, generally contains I/O addresses.

3. AddressRangeACPI: Stores ACPI tables.

4. AddressRangeNVS: Not usable by the OS. This range is required to be saved and restored across an NVS sleep.

5. AddressRangeUnusuable: Memory regions containing errors.

6. AddressRangeDisabled: Memory not enabled.

7. Other: Undefined. Reserved for future use.

An E820 memory map contains a list of valid addresses. It serves the following basic services

1. Indicate the physical address space to the OS.

2. Indicating usable RAM regions.

3. Indicate I/O regions, corrupted memory regions, along with other reserved regions.

The OS discover the system memory and I/O regions using the above table. With the knowledge of the available memory space, the OS can create page tables and other data structures, to use and share the memory. For the reserved I/O regions, there are several self-discovery mechanisms like Plug and Play (PnP) to identify various devices.

1.6 Intel VT Extensions

Popularity and demand for virtualization led Intel to introduce various features for the x86 platform, in VT-x [18] and VT-d [17] extensions. These extensions recognize the following additional modes of operation of the CPU.

1. VMX root: This is the highest level of privilege only accessible to a VMM.

2. VMX non-root: This mode is appropriate for guest VMs, containing all the

typical privilege levels encountered by an OS in a non-virtualized environment.

De-privileging an OS relies on extending the traditional framework between applications and OS, to OS and the hypervisor. Sensitive instructions that used to cause traps to the OS would now trap to the VMX root. These transitions from VMX non-root mode to VMX root mode are termed as VM exits (the reverse is called VM entry). During a VM exit, the state of a guest machine (VM) is saved to, and the state of the host machine (VMM) is restored from, a Virtual Machine Control Structure (VMCS). VMCS is a hardware based memory structure (similar to page table structures) containing various regions defining the CPU state and behavior. It provides a hardware realization to the term vCPU, which was just an abstract concept earlier, realized entirely in software. Since, the CPU state is saved to memory by hardware, it is much faster than prior software implementations of storing registers individually. VMCS structures also provide flexibility of masking VM exits on certain sensitive instructions. Interrupts may be remapped to a vCPU, eliminating the need for the VMM to first catch and then deliver these events to guests.

Typically, applications indirectly gain access to system resources via system calls provided by the OS. These system calls were generally implemented using a software interrupt, that would change the privilege level of the CPU and transfer control of execution to a specific OS subroutine. To eliminate the overhead of a software interrupt and allow faster transitioning, Intel added a SYSENTER instruction that results in a jump to a specified address in the Model-Specific Registers (MSR). A similar functionality has been extended to hypervisors, especially to support paravirtualization. A special VMCALL instruction has been included in the VT-x extensions, that unconditionally causes a VM exit passing the control over to the VMM, in VMX non-root mode. It differs from SYSENTER function in the sense that it stores the guest CPU state to the VMCS structure before moving to the hypervisor, while the SYSENTER instruction does not.

VT-d instructions provide virtualization extensions for devices. It includes facilities such as IOMMU and interrupt remapping which when used in conjunction with VT-x reduce the overhead for I/O operations in virtualized environments. IOMMU has been easier to implement due to the incorporation of northbridge chipset inside the CPU itself. Earlier, memory controllers used to reside on the northbridge chipset external to the CPU. However, due to MOS scaling effects, memory controllers have been moved to the CPU die, and it is a lot easier to support a memory management unit for device memory accesses. VT-d extensions also provide for interrupt remapping capabilities to vCPUs, rather than physical CPUs further minimizing hypervisor intervention.

These extensions are still an ongoing development, with each revision providing newer features and trying to close the gap between virtualized and non-virtualized environments. Hypervisors are also continually evolving to making extensive use of the added capabilities.

2. XEN HYPERVISOR

Xen is an open-source hypervisor, that started as a project at University of Cambridge. It uses paravirtualization in conjunction with hardware assisted virtualization technologies to develop a VMM. Along with widespread adoption, it also boasts of native support from Linux kernel.

2.1 Xen Architecture

Xen hypervisor follows a minimalist design policy with emphasis on security and efficiency. This is extremely important as the hypervisor hosts multiple Virtual Machines and any bugs may compromise the entire system. Virtual machines are hosted on custom environments called domains. Xen exposes very basic functionalities to these domains having similar UNIX counterparts as shown in Table 2.1. Guest OSes should contain relevant modifications to use these features.

Table 2.1: Comparison between Xen and Unix Architecture [10]

UNIX	Xen
System Calls	Hypercalls
Signals	Events
File system	Xenstore
POSIX shared memory	Grant tables

Initially Xen was designed undertaking paravirtualization on x86, i.e. Guest OS kernels were modified to be compatible with Xen. However, with the addition of Intel VT-x and AMD SVM extensions to the x86 architecture, it is possible to support pure virtualization. Guests on newer machines can run in two modes – PV (Paravirtual) guest or HVM (Hardware Virtual Machine) Guest. In HVM mode, Xen can run guests without any source code modifications. On the other hand, running a paravirtualized guest kernel in HVM takes the hybrid approach where the guest may take advantage of hardware features (such as Nested Page Tables), in addition to the performance boosts provided by paravirtualization techniques (such as hypercalls). Guests can use the CPUID instruction to probe whether it is running directly on hardware, or on top of Xen.

The trap and emulate model, for hypervisors, is very expensive in terms of CPU cycles. Thus sensitive operations in a guest OS are typically replaced with hypercalls. This is quite similar to system calls in UNIX. Typically system calls use interrupt 80h (or SYSENTER instruction) to transfer control to the kernel in ring 0 with arguments either placed on the stack or in ISA registers. Hypercalls work in much the same way utilizing interrupt 82h instead in PV mode. However, in HVM mode, most interrupts are generally configured to be fed to the guest kernel instead of Xen. To enter the hypervisor at ring -1, a special instruction, VMCALL, is used instead.

These two separate methods are unified in newer Xen versions via calling an address at a certain offset in a special page mapped to the Guest OS's address space. The offset determines the specific hypercall command. In the above method, the hypercall issue procedure remains the same from the Guest OS kernel's perspective whether in PV mode or in HVM mode. The implementation specifics are hidden in the page mapped to the Guest VM.

In a Xen based system (Fig 2.1), memory and CPU resources are managed directly by the hypervisor, but I/O devices are generally controlled using a privileged domain (generally dom0). This domain runs at a higher privilege level where it is given direct access to many hardware resources. In addition to handling I/O operations, it also hosts the Xen User Interface used for administrative tasks. Thus, its security is of prime importance. Some of its responsibilities, such as hosting a device driver, may be delegated to domU guests (Guest domains) running at a higher privilege level. This feature is very useful in the case of a buggy device driver, as during any device driver related fault, only the specific domU needs to be restarted instead of the entire system. However, in the absence of IOMMU, DMA requests from this domain may potentially affect memory regions allocated to other VMs.

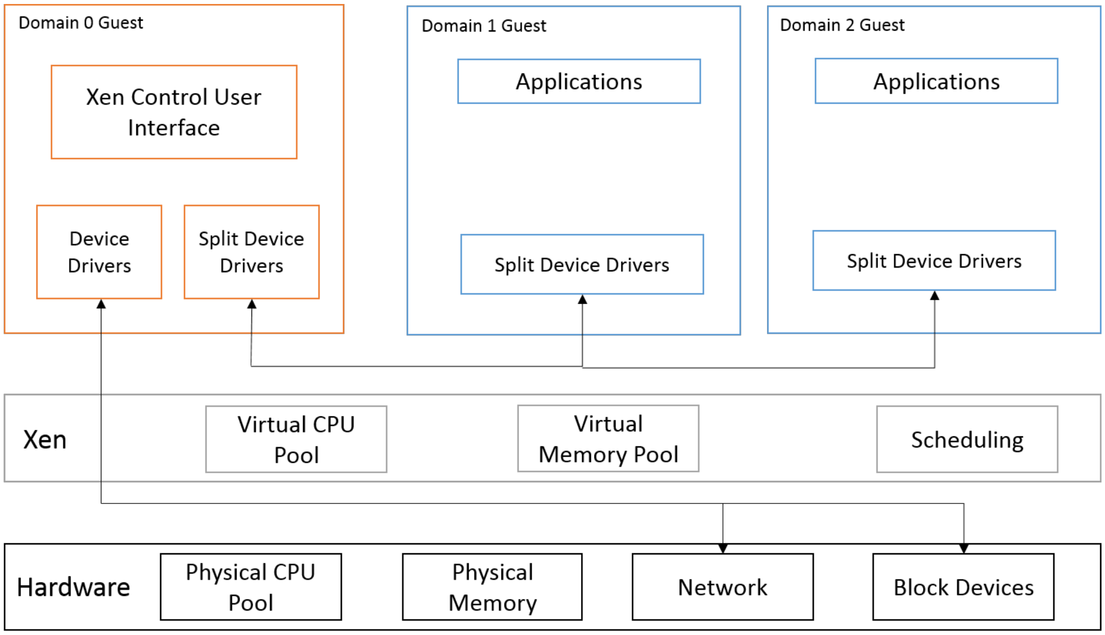


Figure 2.1: Xen Architecture [10]

Memory sharing in Xen is enabled via Grant tables. Memory is shared or transferred among domains at page granularity. This feature is quite useful in many situations such as networking among guests and implementing split device driver model.

One of the features of Xen which makes it very popular is its I/O interface. Xen escapes the need to both emulate devices and develop separate device drivers with a split device driver model. It is made up of the following.

1. Actual device driver.
2. Generic backend driver.
3. Generic frontend driver.
4. Ring buffer.

Typically, dom0 or a driver domain hosts the actual device drivers eliminating a notable amount of redundant developmental work. A generic frontend device driver is implemented in the guest domain at a much higher abstraction level which communicates with the backend device driver using ring buffers. The backend driver deconstructs the I/O requests from the front end driver and forwards it to the actual device driver. The requests are thus kept very simple avoiding device specific details. Ring buffers handle data movement across domains using shared memory utilities provided by Xen (Grant tables). A key advantage of this model is that a single split device driver can cover a whole class of devices.

Time keeping is another important aspect of an Operating System, especially for a scheduler. CPUs are shared amongst the running processes on a time sharing basis, and thus it is of utmost importance that the Operating System has accurate CPU clock information. Additionally, several user space applications also need wall clock time information, which can be calculated from system time. Usually the above is gathered using the CPU clock and network time information. However, in a virtual machine the CPU clock does not reflect the system time, because different Operating Systems share the same CPU. The hypervisor holds the responsibility to provide a virtual machine with system time corresponding to the actual time the VM occupies a CPU. In HVM domains, Xen receives support from the hardware extensions, while in PV domain the above is performed entirely in software.

2.2 Xen Control Interface

The control interface of Xen contains a combination of user space, and kernel space code, through which hypercalls are issued. To host the interface, dom0 must comprise of a compliant kernel (Linux, NetBSD, Solaris, etc.) containing requisite modifications. The process stack, including and above the Xend daemon (Fig 2.2), run as user space applications while the rest run at a higher privilege level.

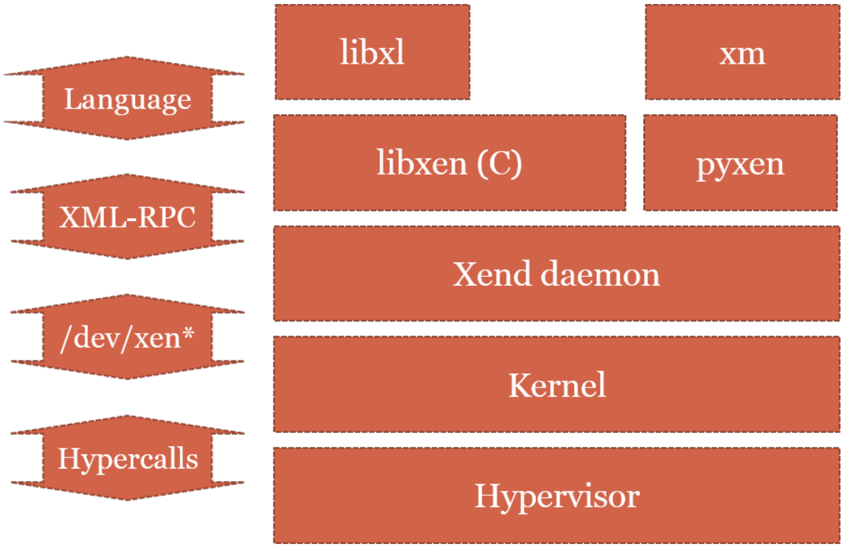


Figure 2.2: Xen API [10]

Xen commands issued through any tool, are converted to XML-RPC messages to communicate with the Xend daemon. Xend daemon forwards the command to the kernel to issue hypercalls. In this chain of flow, the interface of Xend daemon is standardized through the definition of Xen API. This allows development and proliferation of user space management tools independent of changes lower in the stack. The complete Xen Control Interface structure is presented in Fig 2.2. Xen commands can be issued from a variety of user space tools such as xl, xm, libvirt, etc.

Some tools (e.g. xm) written in python, have an additional overhead of a runtime python environment. In contrast, xl is relatively lightweight using libxen library, written in C, to generate XML-RPC messages.

The xend daemon runs in the user space, thus minimizing dependence on a specific kernel. On receiving messages, xend performs certain critical tasks such as access control and issues hypercalls to the hypervisor through the kernel.

Sample commands for xl toolchain are illustrated below:

- Create or start a virtual machine: xl create <config_filename>
- Shutdown a virtual machine: xl shutdown <domain_id>

2.3 Xen Memory Model

Memory management is one of the core components of a hypervisor. With Xen core, the available memory is shared dynamically amongst the different virtual machines. It also allows for thin provisioning, i.e., projecting more memory than the available system RAM using ballooning techniques.

As a part of design philosophy, Xen does not swap pages out of memory itself. Individual guest OSes are the best judges to identify cold pages and thus this job is left over to them. Using the balloon driver, Xen is able to mount or release memory pressure in a VM. When the hypervisor wants to reclaim pages from a VM, it inflates the balloon driver in the virtual machine. The balloon driver requests more memory from the OS, which swaps cold pages out and releases memory to the balloon driver. The latter returns those freed up memory pages to the hypervisor so that it can be allocated to an appropriate VM. During runtime, as the memory requirement reduces, Xen deflates the balloon and releases memory back to the VM. The balloon driver monitors a target memory value, set in Xenstore to dynamically balance the actual memory allocated to the guest. This target memory value (set through domain0) is usually smaller than the guest physical address space and reflects the intended actual memory allocation for that domain.

2.3.1 x86_64 Memory Management

This subsection focuses on the memory management details for x86_64 machines. While extending the x86 ISA to 64 bits, AMD cleaned up memory segmentation controls, leaving a continuous flat address space with page level controls. In x86_64, virtual memory is 64 bits wide, currently allowing only 48 bit sign extended addresses [18]. A typical system looks similar to Fig 2.3, having 128TB + 128TB accessible regions.

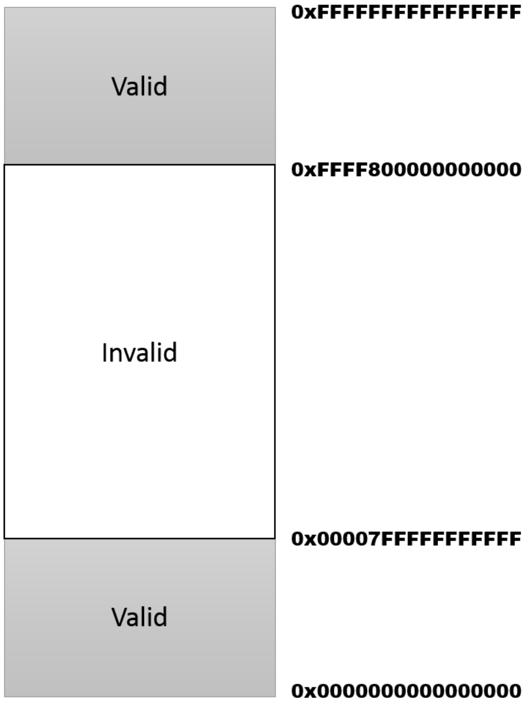


Figure 2.3: x86_64 Virtual Address Space

Typically for most Operating Systems, a virtual address space is generally divided into two parts

1. Kernel Space: This space is shared and common to all the applications. Kernel space also contains a direct mapping of the physical address space (usually with an offset).

2. Application Space: This space is specific to individual applications for application data and code.

For Linux on x86_64 machines, the lower region goes to the application and the upper region is occupied by the kernel. Mapping the kernel into the individual application address spaces avoids the overhead of a context switch during a system call. Moreover, many system calls pass arguments via a pointer to a user space memory region, that is also accessible directly by the kernel, since the kernel is mapped onto the process address space.

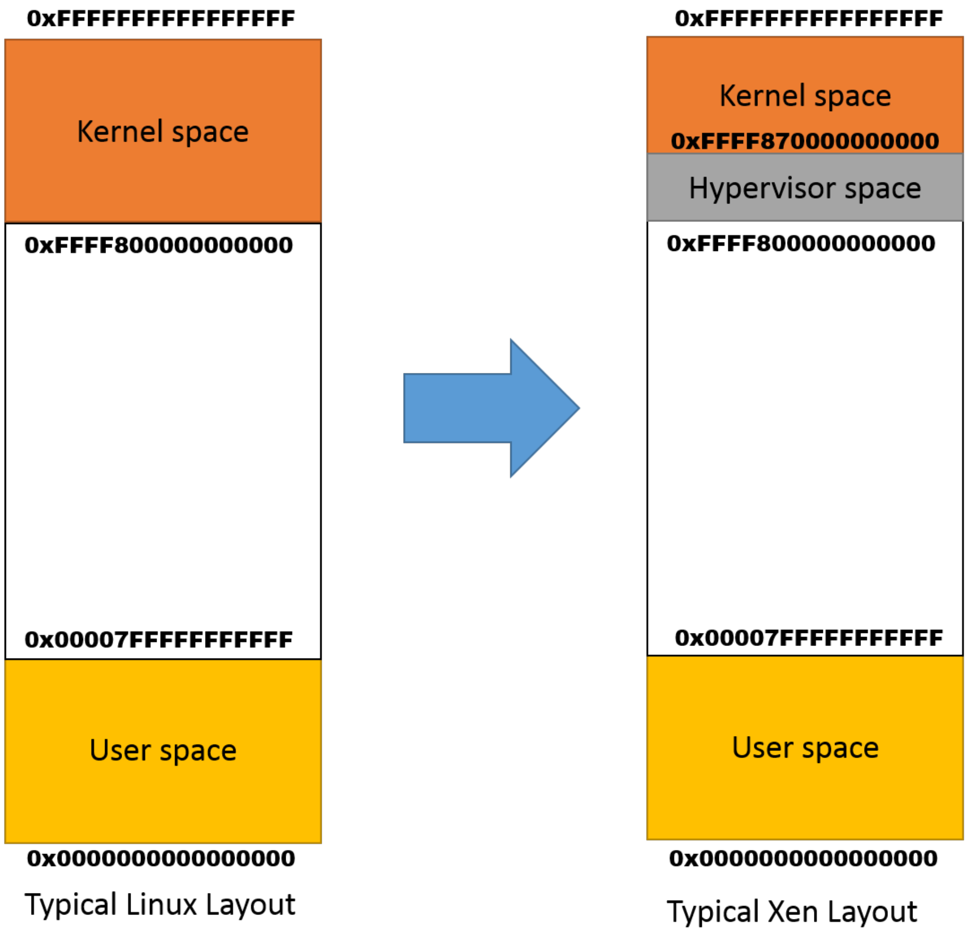


Figure 2.4: Xen Virtual Address Layout Transformation

A similar framework is setup in-between Xen hypervisor and the individual VMs (see Fig 2.4). The hypervisor reserves a portion of the kernel address space for itself. This is done again to avoid context switches during hypercalls from a guest VM. This address space is again subdivided into different regions.

Table 2.2: Virtual Memory Regions [29]

	Start Address	End Address	Description
1	0x0000000000000000	0x00007fffffffffff	[128TB, PML4:0-255] Guest-defined use
2	0x0000800000000000	0xffff7fffffffffff	[16EB] Inaccessible: Only 48-bit sign-extended VAs supported
3	0xffff800000000000	0xffff800000000000	[256GB, PML4:256] Read-only machine-to-phys translation table (GUEST ACCESSIBLE)
4	0xffff804000000000	0xffff807fffffffff	[256GB, PML4:256] Reserved for future shared info with the guest OS (GUEST ACCESSIBLE)
5	0xffff808000000000	0xffff80ffffffffff	[512GB, PML4:257] ioremap for PCI mmconfig space
6	0xffff810000000000	0xffff817fffffffff	[512GB, PML4:258] Guest linear page table
7	0xffff818000000000	0xffff81ffffffffff	[512GB, PML4:259] Shadow linear page table
8	0xffff820000000000	0xffff827fffffffff	[512GB, PML4:260] Per-domain mappings (e.g., GDT, LDT)
9	0xffff828000000000	0xffff82bfffffffff	[256GB, PML4:261] Machine-to-phys translation table
10	0xffff82c000000000	0xffff82c3ffffffff	[16GB, PML4:261] ioremap()/fixmap area

Table 2.2 continued

	Start Address	End Address	Description
11	0xffff82c400000000	0xffff82c43fffffff	[1GB, PML4:261] Compatibility machine-to-phys translation table
12	0xffff82c440000000	0xffff82c47fffffff	[1GB, PML4:261] High read-only compatibility machine-to-phys translation table
13	0xffff82c480000000	0xffff82c4bfffffff	[1GB, PML4:261] Xen text, static data, bss
14	0xffff82c4c0000000	0xffff82f5ffffffff	[197GB, PML4:261] Reserved for future use
15	0xffff82f600000000	0xffff82ffffffffff	[40GB, PML4:261] Page-frame information array
16	0xffff830000000000	0xffff87ffffffffff	[5TB, PML4:262-271] 1:1 direct mapping of all physical memory
17	0xffff880000000000	0xffffffffffffffff	[120TB, PML4:272-511] Guest-defined use.

As seen in Table 2.2, different regions, serve individual purposes, while the complete machine memory is made directly accessible to Xen through region 16.

2.3.2 Hypercalls

Several hypercalls are provided to facilitate common memory management operations in Xen. Although in a pure virtualization environment, it is not necessary, but hypercalls are included for performance gains.

Page table management is one of the most expensive operations in a paravirtualization approach. To prevent direct access to physical hardware, page tables of a domain are generally marked as read only by the hypervisor. Any attempted modifications by the guest VM would result in a trap to the VMM, which then emulates the required operation.

Xen provides hypercalls for the guest to make the above procedure easier. It also allows for multiple page table changes to be clubbed together, via the HYPERVISOR_mmu_update hypercall. However, these operations are not relevant to the case of an HVM guest. In x86 architecture, the VT-x (for Intel chips) and SVM (for AMD chips) extensions allow the guest direct access to its own page tables. This eliminates the scenario for traps altogether.

Thin provisioning is another important feature in a virtual machine. The balloon driver negotiates the actual memory occupied by a domain with the hypervisor. The popular commands used by it are XENMEM_increase_reservation, XENMEM_decrease_reservation and XENMEM_populate_physmap. The first two are used for runtime addition or removal of memory blocks behind the guest physical address space. While the third one is used for the initial mapping of the guest address space during domain creation, for large memory requests. These commands are executed through the HYPERVISOR_memory_op hypercall. Another important command associated with this hypercall is XENMEM_memory_map. It can replace the BIOS call to provide an E820 memory map to a paravirtualized guest. Though not necessary for domU guests, it is one of the available ways to determine the initial layout of guest physical address space.

3. DESIGN AND IMPLEMENTATION

Conception of the proposed system revolves around the design of an additional memory management unit in the hypervisor. Since NVM data has existence beyond the lifetime of a VM, the conventional MMU cannot be used. A new Non-volatile memory management unit is presented as a solution, for a popular hypervisor – Xen.

In this chapter, several features of Xen are described relevant to the x86_64 ISA with VT-x extensions and Intel Hub Architecture, followed by requisite modifications to incorporate the new MMU design.

3.1 Overview

Boot procedure on x86 demands that the CPU should first transition from real mode to protected mode to access the entire address space. It should initialize the necessary page table data structure and segment registers to perform the switch while keeping interrupts masked during the transition process. After entering the protected mode the processor has a lot more flexibility, but typically loses access to BIOS calls. For the above mentioned reasons, bootstrapping on x86 is a quite complex and tricky procedure. Typically, initialization of a MMU is one of the first tasks for any kernel, and it involves a fair amount of bootstrapping code. It makes this undertaking all the more challenging and interesting.

Booting can also designate either the host machine startup or just starting a virtual machine. While the former is associated with the hypervisor taking stock of the hardware resources, the latter involves sharing the same with a VM in a secure manner. The major steps performed in a Xen based system from power on to starting a new domain is summarized in the flowchart presented in Fig 3.1 and Fig 3.2. As evident from this figure this project can be broken down into two fundamental subdivisions.

1. Create an independent Memory Management Unit for the available Non-Volatile RAM.

2. Provide guest domains access to the NVRAM region, in a secure and protected manner.

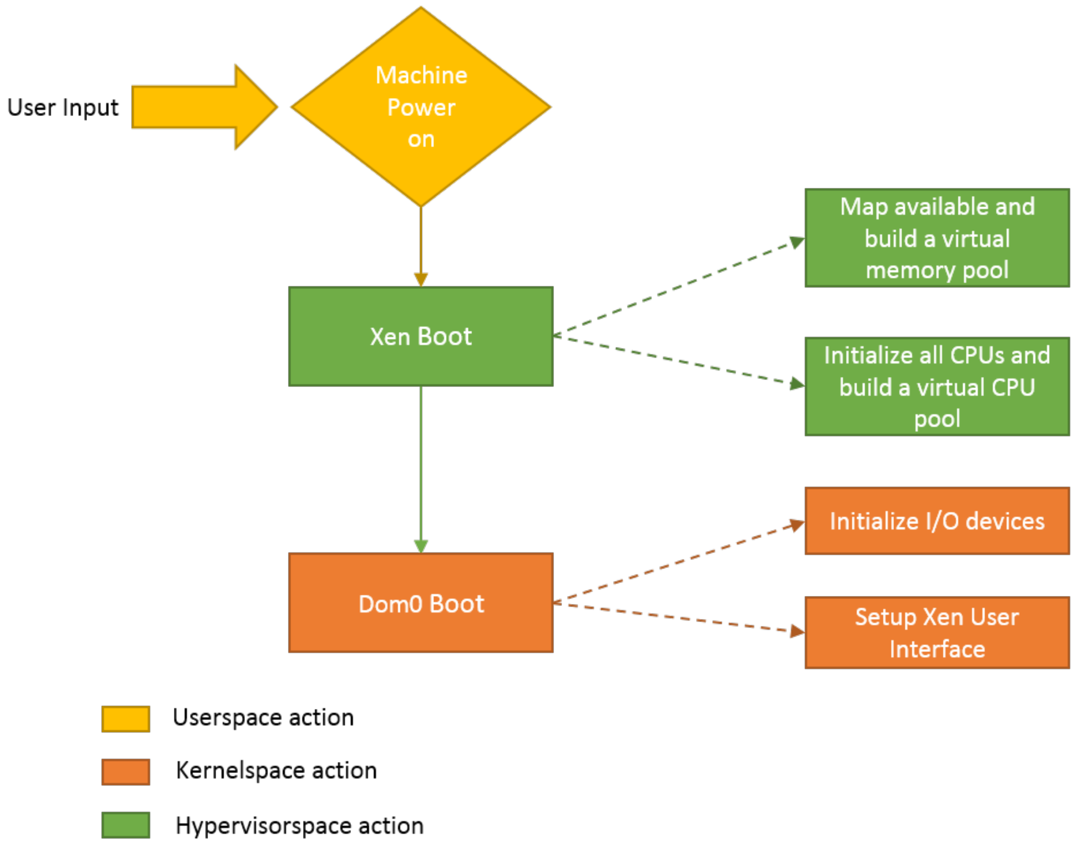


Figure 3.1: Domain Creation Flowchart : System Boot

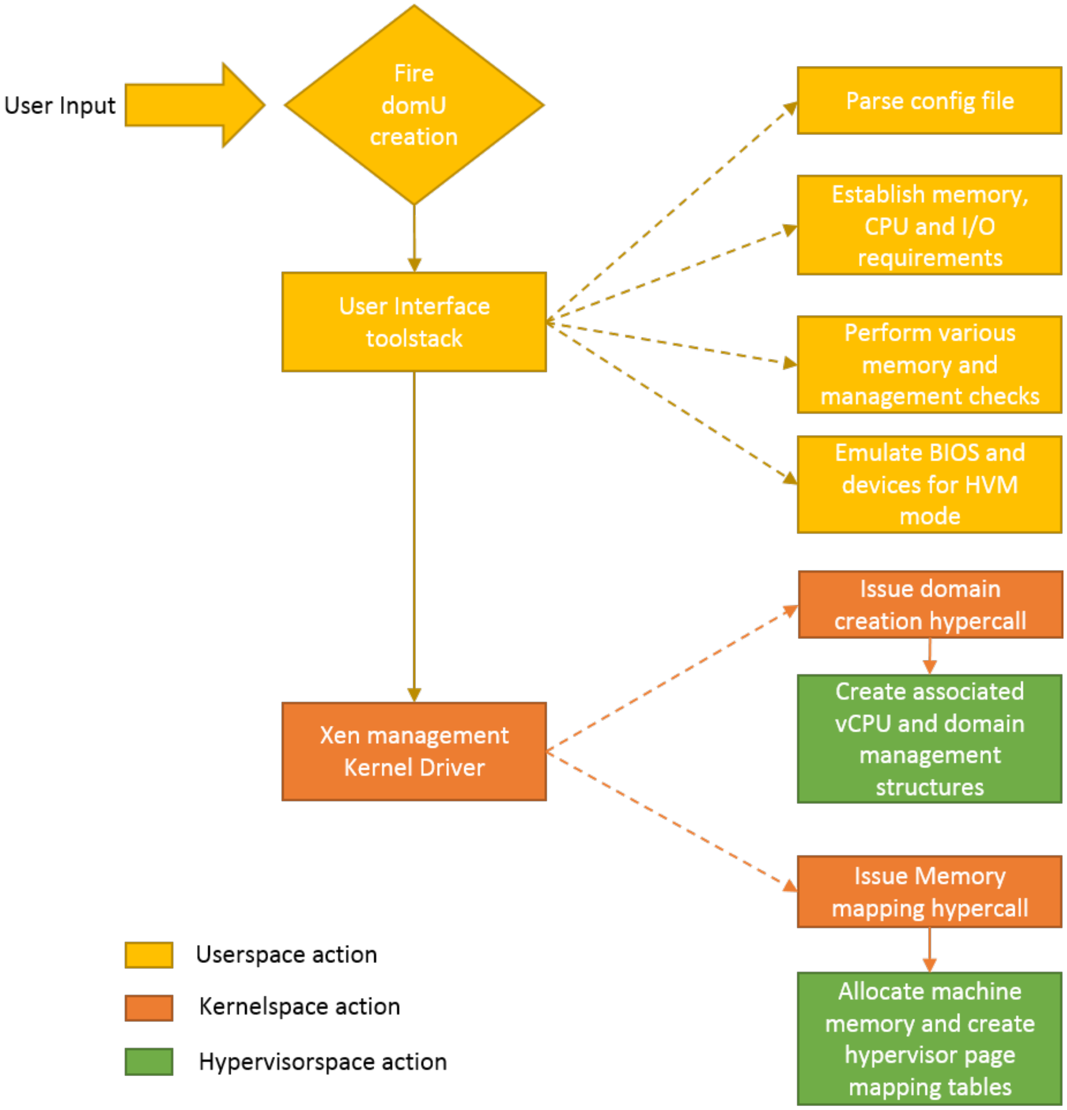

User Input
Fire domU creation
User Interface toolstack
Parse config file
Establish memory, CPU and I/O requirements
Perform various memory and management checks
Emulate BIOS and devices for HVM mode
Xen management Kernel Driver
Issue domain creation hypercall
Create associated vCPU and domain management structures
Issue Memory mapping hypercall
Allocate machine memory and create hypervisor page mapping tables
Userspace action
Kernelspace action
Hypervisorspace action

Figure 3.2: Domain Creation Flowchart : DomU Boot

The first task requires modifications in the Xen kernel, with parts that deal with the boot process. Since the modifications are at the highest privilege level, it is very critical that the implementation be bug free, secure and efficient. Even minor bugs may cause the entire system to crash.

On the other hand, the second job deals more with user level code, specific to a toolchain. A notable part of domain creation focuses on emulation of firmware, performed mainly by various management tools. Only a small portion deals with issuing hypercalls for resource allocation. Therefore, in this section most of the implementation details reside in the Xen control interface with minor modifications to the hypercall structure. Due to standardization of Xen API, many tools have sprung up for VM management. This implementation picks up the xl toolchain, which is the default tool supported by Xen community.

A basic outline of the implementation is presented below, in Fig 3.3, highlighting the modifications in red.

1. Recognize the presence of NVRAM from E820 memory map provided by BIOS.
2. Create a separate NVRAM pool to manage the resource independently.
3. Create separate interface to specify DomU RAM and NVRAM requirements.
4. Generate virtual E820 memory map reflecting the separate NVRAM pool.
5. Map the virtual NVRAM space in domUs to the physical Non-volatile memory present on the machine.

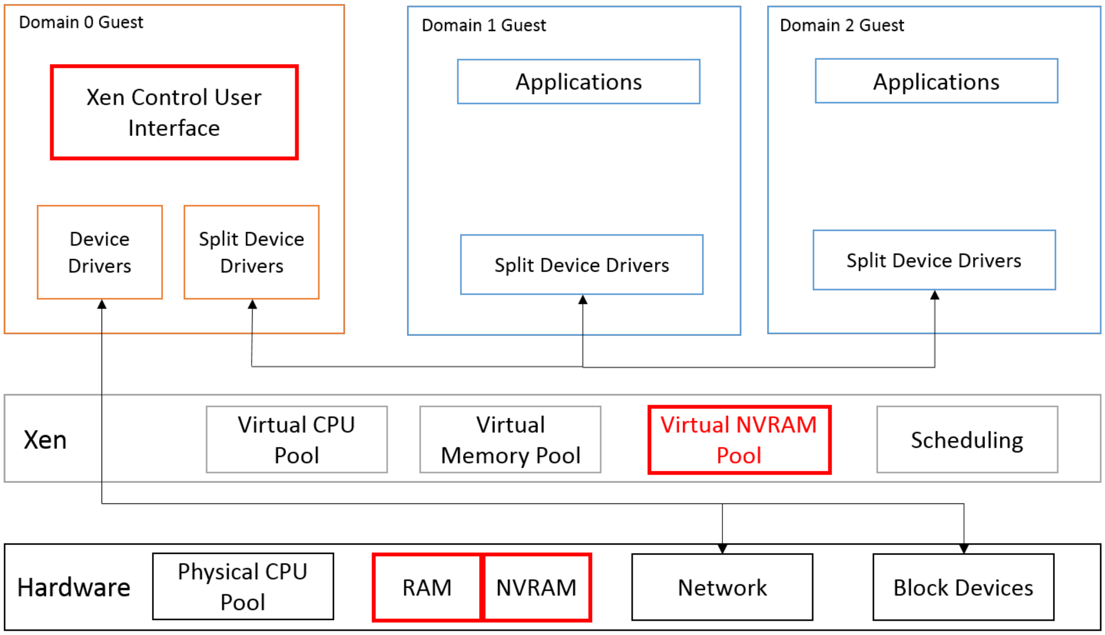


Figure 3.3: Modified Xen Architecture

3.2 Challenges

There are several challenges faced during the design of the proposed system in Xen. The most notable being the absence of a modular design for system memory. Xen has been architected around volatile memory, and the notion of a homogeneous memory is integrated quite deep in the system design. This makes it all the more challenging to introduce another MMU. Moreover, a major portion of the initialization is done during the boot procedure, which is in itself quite a complex process for x86. Along with the above virtualization being a relatively new technology lacks sufficient documentation. Thus, even minor code changes turn into a grueling task. The combination of the above issues makes the implementation an arduous but exciting challenge.

3.3 Xen Boot Procedure

On system start, Xen boots up first to take stock of the hardware present. It queries the BIOS for the E820 Memory map. Identifying the available memory regions, Xen builds preliminary page tables and switches the machine to protected mode. With paging enabled, the hypervisor can access the entire address space. It follows by building free page lists.

Each machine page is identified by a data structure called page_info presented below [29]. This structure contains runtime administrative information about the machine pages such as status, domain identifier, special page status, order, etc.

```

struct page_info
{
    union {
        struct page_list_entry list;
        paddr_t up;
        uint64_t shr_handle;
    };
    /* Reference count and various PGC_xxx flags and fields. */
    unsigned long count_info;
    /* Context-dependent fields follow... */
    union {
        /* Page is in use: ((count_info & PGC_count_mask) != 0). */
        struct {
            /* Type reference count and various PGT_xxx flags and fields. */
            unsigned long type_info;
        } inuse;
        /* Page is in use as a shadow: count_info == 0. */
        struct {
            unsigned long type:5; /* What kind of shadow is this? */
            unsigned long pinned:1; /* Is the shadow pinned? */
            unsigned long head:1; /* Is this the first page of the shadow? */
            unsigned long count:25; /* Reference count */
        } sh;
        /* Page is on a free list: ((count_info & PGC_count_mask) == 0). */
        struct {
            /* Do TLBs need flushing for safety before next page use? */
            bool_t need_tlbflush;
        } free;
    } u;
    union {
        /* Page is in use, but not as a shadow. */
        struct {
            /* Owner of this page (zero if page is anonymous). */
            __pdx_t _domain;
        } inuse;
        /* Page is in use as a shadow. */
        struct {
            /* GMFN of guest page we're a shadow of. */
            __pdx_t back;
        } sh;
        /* Page is on a free list. */
        struct {
            /* Order-size of the free chunk this page is the head of.
  */
            unsigned int order;
        } free;
    } v;

    union {
        u32 tlbflush_timestamp;
        struct {
            u16 nr_validated_ptes;
            s8 partial_pte;
        };
        u32 shadow_flags;
        __pdx_t next_shadow;
    };
};
```

An array of these structures is initialized for all the machine pages, as shown in Fig 3.4. This array of struct page_info occupies the region 15 of Xen virtual address space identified in Table 2.2. Xen modifies this data structure to indicate any changes to the machine page. With predefined virtual address space regions for both the page_info structures and the machine pages, a pointer to this structure can be used to calculate the virtual address of the corresponding page and vice versa.

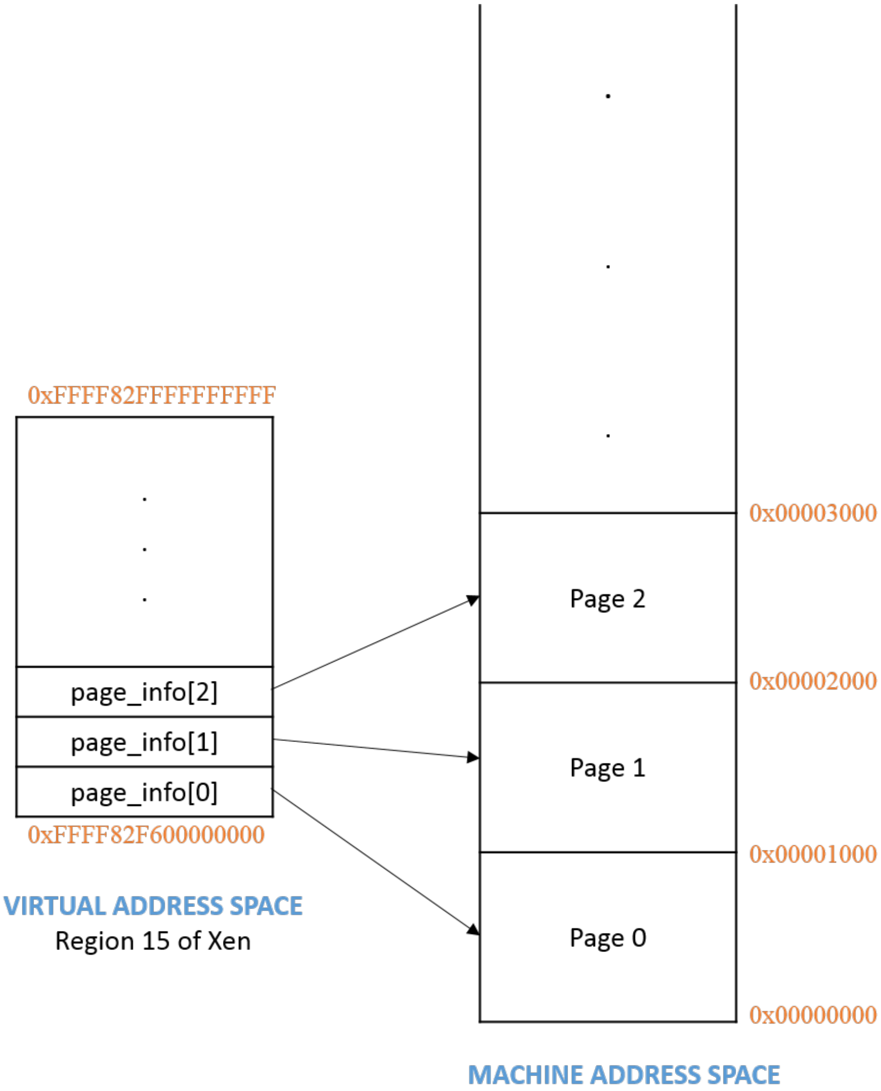


Figure 3.4: Struct Page_info array to Machine Page Mapping [29]

On system boot up, all the machine pages are free, without any domain affiliations. The Memory Management Unit builds a memory pool data structure _heap[MEMZONE][ORDER] to manage and assimilate all the free pages. This _heap data structure (Fig 3.5) represents the free memory pool in Xen, arranging all the pages by MEMZONE and ORDER. Here MEMZONE divides the entire machine address space into different zones based on the position of the first non-zero bit in the address. This is necessary for special DMA memory requests for devices with fewer address bits. In each ZONE the pages are sorted by the order of the contiguous available pages up to a maximum of 1GB.

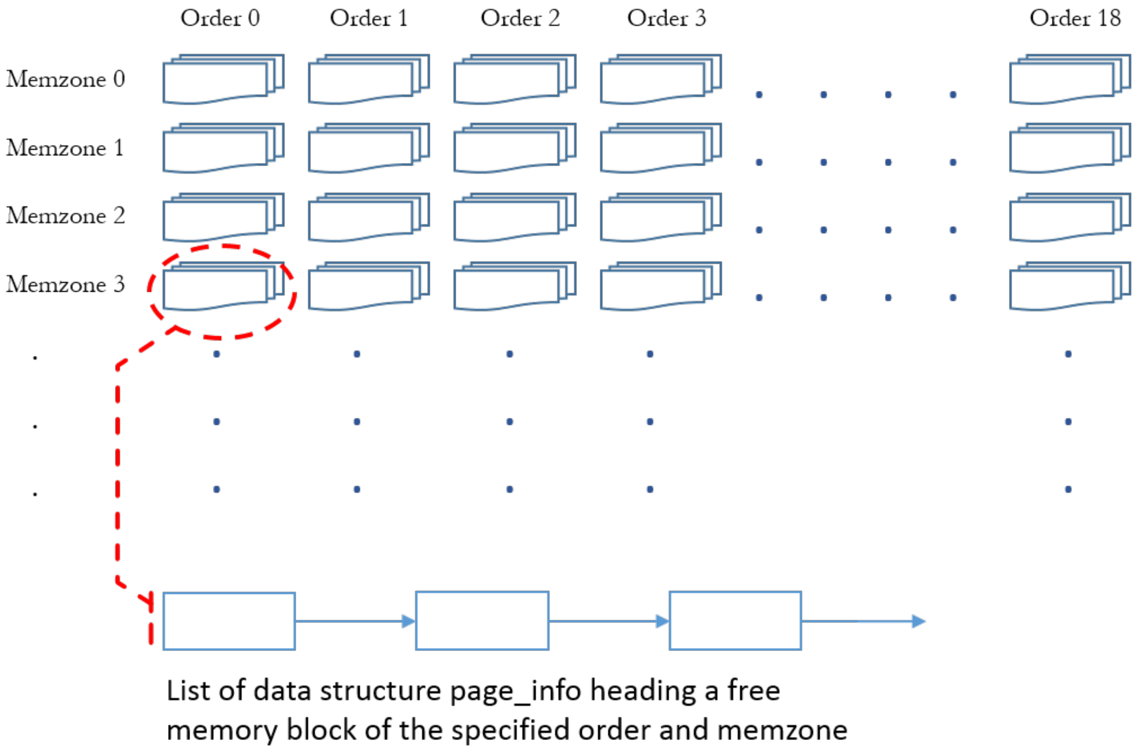

Order 0
Order 1
Order 2
Order 3
Order 18
Memzone 0
Memzone 1
Memzone 2
Memzone 3
List of data structure page_info heading a free memory block of the specified order and memzone

Figure 3.5: Memory Pool – _heap Data Structure

Each location in the 2-D array contains a list of the pages (actually their corresponding page_info data structure) representing contiguous memory regions of the corresponding order and memzone. Memory requests are honored at page granularity by alloc_heap_pages and its wrapper functions, which extracts contiguous pages from the above _heap data structure following a buddy system allocation. Pages returned by any domain are added to the memory pool using the free_heap_pages function call. This function coalesces any adjacent free regions, if available, and adds the memory pages back to the _heap data structure. These pages are first scrubbed clean of all information before adding to the pool, protecting the security across domains. The pool is always aggregated to the highest order during these return requests.

Once the memory pool is ready, Domain 0 is given an appropriate amount of memory and the kernel image is copied. The hypervisor follows to build a CPU pool out of the available cores. It initializes VMCS and other associated data structures and hands control over to Domain0 Linux kernel. It is the responsibility of dom0 now to initialize all I/O devices using their appropriate drivers.

3.3.1 Design Modifications

To enable sharing NVRAM across separate VMs, we have to first recognize it as a memory region. Currently the firmware (BIOS) marks that section with code 90. Due to lack of standardization, this code is subject to change and presently not recognized by Xen. Therefore, the hypervisor treats it as an unrecognized address space. Xen inhibits from either writing to or reading from these addresses.

The first task should be to recognize the memory region, and mark it as Non-Volatile RAM. A separate code word is added to the list of recognized E820 codes, to that effect. After identifying the region, a new NV memory pool is added in the form of an additional data structure _nvm_heap. This structure is quite similar to the _heap data structure explained earlier except that it stores the pages of non-volatile memory. The E820 entries are examined one by one and the appropriate memory pools are populated, i.e. volatile RAM pool (identified by structure _heap) is filled with RAM pages and non-volatile memory pool (identified by structure _nvm_heap) is filled with NVRAM pages.

Separate NVM allocator functions nvm_alloc_heap_pages and nvm_free_heap_pages are defined which operate on the _nvm_heap data structure. Similar to the volatile memory management function counterparts, these functions just manipulate the NV memory pool data structure and do not create any additional page table mappings. Since all pages are accessible via direct mapping and Xen does not perform swapping, page table modifications are not needed. The structure, page_info, is also modified by adding a separate flag to differentiate between volatile and non-volatile pages.

With the above mentioned modifications in place, Xen is able to boot up, identify the available Non-volatile memory and allocate/de-allocate NVRAM pages on demand. A basic non-volatile memory management unit has been setup. It still requires an interface for domains to specify non-volatile memory requirements and request non-volatile memory through the NV MMU. The next section discusses a guest domain creation procedure, where modifications are added to utilize the above infrastructure.

3.4 Guest VM Boot Procedure

Booting on bare metal x86 can be quite different from the virtual environment provided by Xen. An x86 CPU starts in 16 bit real mode, with BIOS providing basic essential functions such as hardware information, address space resolution and I/O device drivers. During the boot up procedure, the OS builds page tables and transitions the CPU to protected mode. At this point generally BIOS interrupt calls are unavailable so the OS device drivers are used for I/O. Most of the above functionalities are unavailable in a VM in Xen because the hypervisor boots up first transitioning the machine into protected mode. This poses a problem as the CPU state is totally different in a VM to what an Operating System is expecting.

A guest in Xen can boot in two available modes PV and HVM. Both of them solve the above issue in separate ways. In PV mode, the OS kernel is modified to allow booting in protected mode. Due to unavailability of BIOS, boot time information is passed to the guest using shared memory pages. There are two types of shared memory pages.

1. Start info page: These pages are mapped to the guests address space by Xen. It provides necessary information such as total available memory (essentially the E820 memory map), number of virtual CPUs, console connection, and data structures regarding Xenstore. For boot purposes, Xen explicitly provides only a console device. Any other device must be mapped by the guest kernel using Xenstore services.

2. Shared info pages: A guest kernel needs to explicitly map these pages to its own address space for accessing dynamic runtime information about the virtual machine such as wall clock time, architectural information and event channels. This data is continually updated to reflect the status of the virtual machine.

Information provided through these channels helps in replacing BIOS functionality in PV guests.

HVM mode, on the other hand, is supposed to run unmodified OSes, thus it has to forgo some of the performance benefits of the PV mode and emulate certain devices. Emulation is not optimal because every abstract action is first converted to device specific commands by the guest OS device driver, and these commands are intercepted by Xen. The device emulator reconverts these device specific commands back to the abstract actions that are forwarded to the actual device driver for completion. The Xen split device driver model in PV guests generally runs at a higher level of abstraction avoiding most of the redundant work observed in emulation.

BIOS is emulated by borrowing code from Bochs emulator. It forms the front end in a split device driver model, with the back end handled by code borrowed from QEMU, which is used to emulate devices in HVM mode. Xen starts a domain from the BIOS start point in the virtual 8086 mode present in x86. This mode is actually intended for running legacy application in real mode, alongside protected mode applications. Since it was designed for userspace code, the boot code of an OS (containing a good amount of sensitive instructions) result in numerous traps which have to be handled individually by Xen.

In a hybrid approach, Paravirtualized guests can take the benefit of paravirtualization techniques as well as hardware accelerators while running in HVM mode. The guest can specify a location in its ELF header where the hypercall page can be loaded. It can also execute the CPUID instruction to determine whether it is running on top of Xen or bare metal hardware. If running on Xen, the guest may choose to switch over to Xen specific device drivers, and also map the hypercall page during runtime. However, all these functions are typically performed after the kernel boots up, thus BIOS and QEMU emulator are generally needed during boot-up.

Starting a domain begins with defining a configuration file that includes the disk storage, memory, vCPU and other parameters. The command xl create ¡filename¿ performs certain administrative tasks in the userspace domain and then fires up the domain. Initially, the xl tool parses the configuration file to identify resource requirements, most important of them being memory. If sufficient memory is not available, then an attempt is made to free up memory by communicating to Xen the additional memory requirement via Xenstore. This would typically result in Xen inflating the balloon drivers. In the event of still not meeting the memory requirement, the xl tool aborts the domain creation process with an error.

On the other hand, if the memory requirements are met, the tool issues a hypercall (via xend) to build necessary data structures (VMCS and internal Xen data structures) for domain management. The management tool also creates a guest physical address space and sends a hypercall for appropriate memory allocation and page-table mappings. The hypervisor, on receiving this hypercall, allocates pages via the associated functions mentioned in the previous section. It also builds up the hypervisor page tables to map the guest physical pages to actual machine pages. If all is successful then a virtual E820 memory map is created, and the new VM is booted with emulated firmware using code borrowed from Bochs and QEMU.

3.4.1 Design Modification

The overall objective of this section is to create an interface for configuration and allocation of Non-volatile memory. Current work focuses on extending this functionality only for HVM guests. Here, implementation details lie in three areas xl management tool, Bochs emulator, and hypercall structure.

First and foremost, a parameter is added to the configuration file facilitating the specification of the non-volatile memory requirement in Megabytes. This parameter is read by the tool chain, which then creates additional guest physical address space for NVRAM. Actual memory allocation is performed at this point using hypercalls to notify the hypervisor of the guest physical page numbers with emphasis on allocation of contiguous memory blocks to minimize TLB entries. The x86_64 architecture provides page mapping in the sizes of 1GB, 2MB and 4KB. The allocator moves in descending order for memory requests to satisfy the requirement. A flag is added to the hypercall argument to differentiate between volatile and non-volatile memory requests. If the flag is set, the hypervisor pulls pages out of the NVRAM pool, else it uses the RAM pool for the purpose.

The above procedure completes one of the most critical portions of domain creation. Allocations of both volatile and non-volatile memory are complete and Xen is left with the task of firmware emulation. The toolchain creates virtual E820 table mappings, with an added region to indicate NVRAM with code 90. This completes the boot procedure and the virtual machine is equipped with non-volatile memory. When the VM is turned off, the hypervisor reclaims both RAM and NVRAM pages and adds them to their respective memory pools.

4. EXPERIMENTAL SETUP AND RESULTS

The experimental setup consists of a server chassis [1], provided by Viking Technologies with the configuration shown in Table 4.1.

Table 4.1: Setup Machine Configuration

Component	Specification	Quantity
CPU	E5-2640	2
Motherboard	X9DRH-iF Ver 1.02	1
RAM	DDR3 RDIMM	1 x 4GB
NVRAM	DDR3 ArxCis NVDIMM	2 x 4GB
SSD	SATADIMM	100GB
OS	Fedora 17 (x86_64)	-
Kernel	3.5.1	-
Hypervisor	Xen 4.1.5	-
NVRAM SDK	ArxSDK 1.3	-

The BIOS present on this machine marks the NVRAM area with a code of 90 and it occupies the address space from 6GB to 14GB. The software development kit ArxSDK 1.3, provided by Viking Technologies, was used for testing the read/write speeds and operation of NVRAM region.

Mainly two components of the ArxSDK was used for profiling purposes.

1. Driver: The driver code programs the Integrated Memory Controller to query for the presence of NVDIMMs. It stores relevant information such as the start

address, size, and number of NVDIMMs, allowing a host of ioctl commands for additional programming of the IMC.

2. Test Code: This code obtains the available NVRAM physical address space from the driver and maps the former to its own address space using the mmap system call. It proceeds to write data sequentially to this region and read it back. It reports the amount of time taken to complete the whole test as a rough measure of throughput.

The driver code was modified to remove most of the above mentioned functionality, since access to the IMC and other privileged commands are not available to a guest domain. Performance measurements were taken using the perf profiling tool, but without any significant advantage. The perf tool collects hardware performance counters such as TLB misses, L1 cache misses, L2 cache misses, etc., but these measurements are not available in a guest domain in Xen.

Each guest domain is equipped with 3.5GB of non-volatile RAM for this test with the cache set to Write Back mode. The results presented in Fig 4.1 compares three cases

1. Test running on 1 guest domain.
2. Tests running on 2 guest domains simultaneously.
3. Test running on the bare Linux (the same version 3.5.1) without the hypervisor.

One run of the test code writes a predetermined sequence to the whole 3.5GB non-volatile memory area and then reads it back. This code is run five times and the CPU time taken in the whole procedure is recorded as a single reading. Each case is tested multiple times (>14 times), and a very low standard deviation is observed in the results as indicated by the error bars.

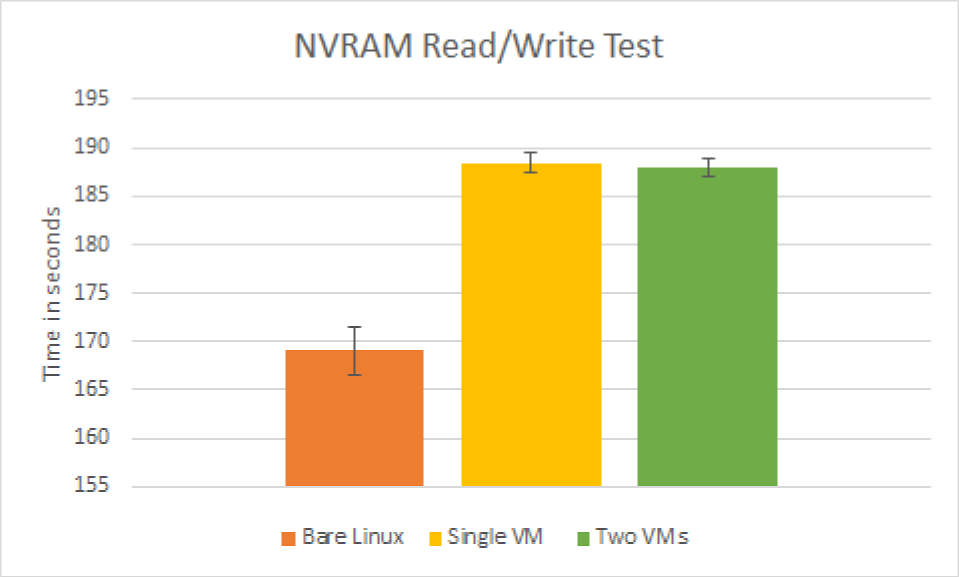
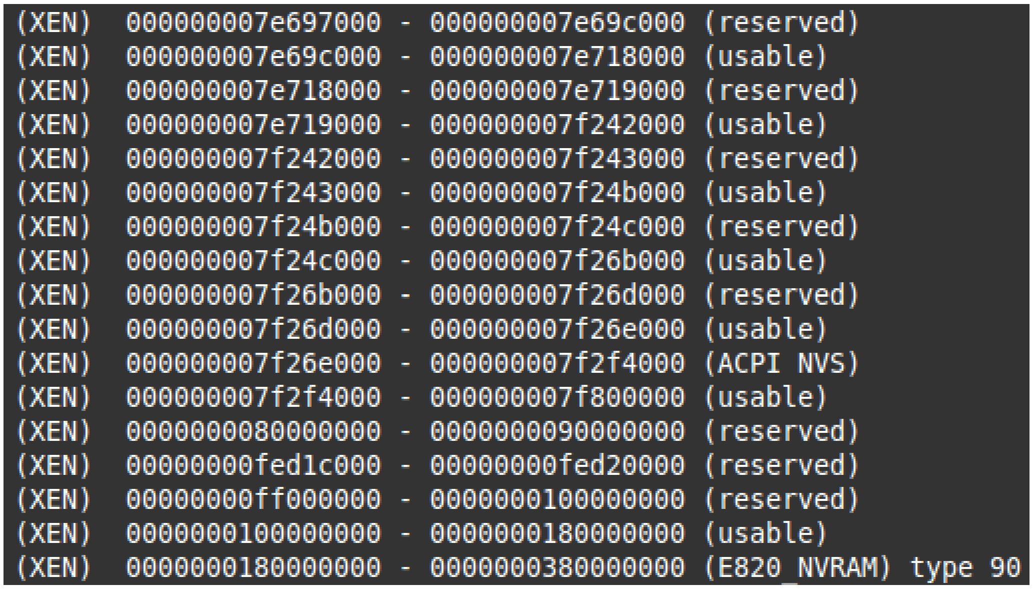


Figure 4.1: Comparison of Read/Write Performance of Bare Linux vs Single VM and vs Two VMs

In the graph presented in Fig 4.1, it can be observed that there is a visible overhead (12% as compared to bare Linux case) for operating in a virtualized environment. However, there is hardly any difference observed between the cases of running a single VM versus that of running two VMs. This can be explained by the fact that system time in a virtual environment is quite different from the wall clock time. For a guest domain, the vCPU time increases only as long as the guest domain occupies that resource. When it is swapped for another virtual machine, the vCPU clock corresponding to the domain is stopped. Thus from the perspective of a virtual machine, the performance should remain similar irrespective of the number of VMs running on the machine. The higher time cost observed in a virtual environment as compared to a bare Linux can be attributed to a combination of hardware and software factors such as higher penalty for TLB misses and overhead for operating on top of Xen.

Several screenshots are attached below which present various stages of the NVM sharing. Fig 4.2 displays part of the original E820 Memory Map generated by the BIOS and received by Xen. The whole list is quite long supporting numerous reserved regions for MMIO devices. In the last line, NVRAM, indicated with code 90, can be seen occupying the address space from 6GB to 14GB.



```
(XEN)  000000007e697000 - 000000007e69c000 (reserved)
(XEN)  000000007e69c000 - 000000007e718000 (usable)
(XEN)  000000007e718000 - 000000007e719000 (reserved)
(XEN)  000000007e719000 - 000000007f242000 (usable)
(XEN)  000000007f242000 - 000000007f243000 (reserved)
(XEN)  000000007f243000 - 000000007f24b000 (usable)
(XEN)  000000007f24b000 - 000000007f24c000 (reserved)
(XEN)  000000007f24c000 - 000000007f26b000 (usable)
(XEN)  000000007f26b000 - 000000007f26d000 (reserved)
(XEN)  000000007f26d000 - 000000007f26e000 (usable)
(XEN)  000000007f26e000 - 000000007f2f4000 (ACPI NVS)
(XEN)  000000007f2f4000 - 000000007f800000 (usable)
(XEN)  0000000080000000 - 0000000090000000 (reserved)
(XEN)  00000000fed1c000 - 00000000fed20000 (reserved)
(XEN)  00000000ff000000 - 0000000100000000 (reserved)
(XEN)  0000000100000000 - 0000000180000000 (usable)
(XEN)  0000000180000000 - 0000000380000000 (E820_NVRAM) type 90
```

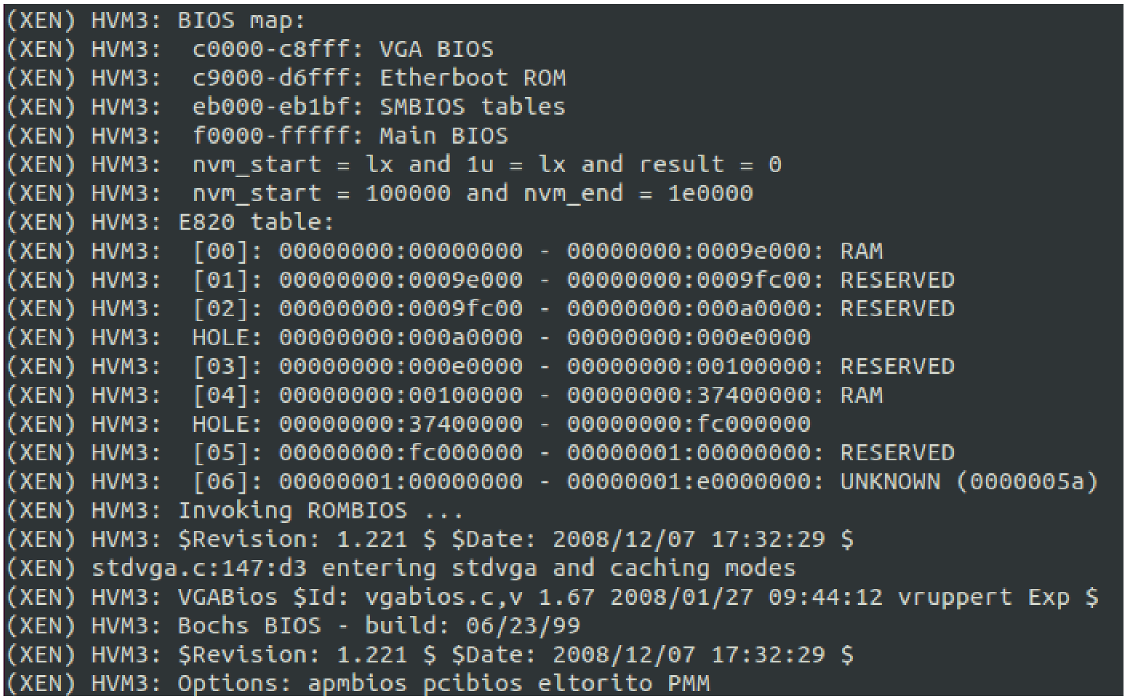
Figure 4.2: Machine E820 Memory Map

Fig 4.3 shows the Xen kernel level logging messages on the creation of a domU with the 892MB of RAM and 3.5GB of NVRAM. The virtual E820 layout generated by Bochs code shows the various regions, with holes introduced for I/O devices. NVRAM is mapped from 4GB to 7.5GB, in consideration with the Arxcis SDK 1.3 which requires the non-volatile memory to be placed above 4GB in aligned blocks of 512MB. This proves beneficial in several ways

1. A separate driver need not be developed, allowing reuse of existing framework.
2. Several Linux kernel versions change the unrecognized code 90 (or 0x5A in

hexadecimal) in the E820 table to code 2, marking it as an I/O space, thereby making NVRAM discovery process extremely difficult. This ordeal can be avoided altogether by placing the non-volatile memory above 4GB, since, chipsets typically place I/O devices just below the 4GB mark. Therefore, any I/O region found above 4GB can be assumed to be non-volatile memory.

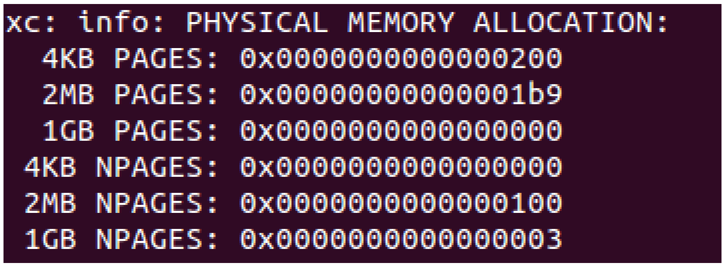
3. Aligning the NVM region on the gigabyte boundary and placing it above 4GB (avoiding the virtual MMIO hole introduced by Bochs), reduces the fragmentation of the NVRAM address space. This allows for superpage allocations of 1GB pages in the hypervisor page tables, reducing the total number of page table entries and potential TLB misses.



```
(XEN) HVM3: BIOS map:
(XEN) HVM3:  c0000-c8fff: VGA BIOS
(XEN) HVM3:  c9000-d6fff: Etherboot ROM
(XEN) HVM3:  eb000-eb1bf: SMBIOS tables
(XEN) HVM3:  f0000-fffff: Main BIOS
(XEN) HVM3:  nvm_start = lx and 1u = lx and result = 0
(XEN) HVM3:  nvm_start = 100000 and nvm_end = 1e0000
(XEN) HVM3: E820 table:
(XEN) HVM3:  [00]: 00000000:00000000 - 00000000:0009e000: RAM
(XEN) HVM3:  [01]: 00000000:0009e000 - 00000000:0009fc00: RESERVED
(XEN) HVM3:  [02]: 00000000:0009fc00 - 00000000:000a0000: RESERVED
(XEN) HVM3:  HOLE: 00000000:000a0000 - 00000000:000e0000
(XEN) HVM3:  [03]: 00000000:000e0000 - 00000000:00100000: RESERVED
(XEN) HVM3:  [04]: 00000000:00100000 - 00000000:37400000: RAM
(XEN) HVM3:  HOLE: 00000000:37400000 - 00000000:fc000000
(XEN) HVM3:  [05]: 00000000:fc000000 - 00000001:00000000: RESERVED
(XEN) HVM3:  [06]: 00000001:00000000 - 00000001:e0000000: UNKNOWN (0000005a)
(XEN) HVM3: Invoking ROMBIOS ...
(XEN) HVM3: $Revision: 1.221 $ $Date: 2008/12/07 17:32:29 $
(XEN) stdvga.c:147:d3 entering stdvga and caching modes
(XEN) HVM3: VGABios $Id: vgabios.c,v 1.67 2008/01/27 09:44:12 vruppert Exp $
(XEN) HVM3: Bochs BIOS - build: 06/23/99
(XEN) HVM3: $Revision: 1.221 $ $Date: 2008/12/07 17:32:29 $
(XEN) HVM3: Options: apmbios pcibios eltorito PMM
```

Figure 4.3: DomU Boot Up Kernel Messages

The actual page allocation is visible in Fig 4.4. Since the RAM address space is fragmented, it contains a combination of 2MB and 4KB pages to satisfy its request. However, NVRAM occupies a contiguous address space comprising of three, 1GB pages and 256, 2MB pages.



```
xc: info: PHYSICAL MEMORY ALLOCATION:
  4KB PAGES: 0x0000000000000200
  2MB PAGES: 0x00000000000001b9
  1GB PAGES: 0x0000000000000000
 4KB NPAGES: 0x0000000000000000
 2MB NPAGES: 0x0000000000000100
 1GB NPAGES: 0x0000000000000003
```

Figure 4.4: DomU Page Allocation

5. CONCLUSION AND FUTURE WORK

Sharing non-volatile memory in a virtual environment opens up several new challenges and design problems. The main issue is that until now there was no notion of different forms of main memory. Memory was treated as a uniform device, and everything else is left as I/O. NVRAM breaks the conformity, by adding a completely different memory device. Thus it requires a new NV memory management suite, alongside the traditional volatile memory management modules. The two, though share similar jobs and have several contrasting features. It calls for a modular design of memory management units similar to that present for I/O devices. In a virtual environment, the need is even more important, as memory and I/O subsystems are considered to be a bottleneck. Introduction of NVRAM would ease the pressure on I/O disk ops, and memory used as disk cache.

This work can be considered as a first step in that direction, which brings several interesting research questions. NVRAM has both the properties of disk storage and volatile memory. What direction should it follow while sharing the resource, disk or memory? The answer is found in a combination of the two. Since non-volatile memory is a precious resource (like RAM), therefore it cannot be permanently allocated to VMs (like disk space), even when they are not operational. The hypervisor needs to implement a dynamic memory pool which grants NVRAM requests on machine boot up and reclaims them back when the machine is no longer operational. It also has to follow the same thin provisioning schemes as main memory, implementing a balloon driver to retrieve memory back.

All these memory like operations brings us to the main difference between volatile and non-volatile memory. In one case, where volatile data ceases to exist as soon as the machines are powered off, non-volatile RAM on the other hand preserves data. This data needs to be preserved between reboot cycles for a VM. The same needs to follow for entire system (including the hypervisor) reboots too. While data retention during entire system reboots is taken care of in hardware, mapping of non-volatile resources should also be maintained. These issues again can be broken down into two separate simpler tasks.

1. Preserve the non-volatile data for VMs even after the VM is powered off.
2. Maintain the hypervisor page table mappings across a system reboot.

When a VM is created, its non-volatile data needs to be restored to its state before power down. Additionally, during a shutdown procedure, NVRAM pages need to be reclaimed, as it is a critical resource. Thus, the data has to be shipped to disk. The above requirement is very similar to the save state and restore state features offered for virtual machines. Here all the data present in main memory is written to a file on disk. The memory state is restored from a file on disk, during the restore procedure. The same procedure can be followed for NVRAM region thereby not wasting memory when the VM is non-operational. The file location can be specified in the VM configuration file. This solution seems to be efficient and straightforward as far as the first task is concerned. However, the second task is quite demanding and tricky.

Since main memory does not expect presence of any valid data during the boot procedure, the available memory regions are usually initialized from scratch. The same procedure is followed for the current implementation of NVRAM memory initialization in the hypervisor. Although we still retain persistent data, the respective mappings of memory pages are lost. It is very important to emphasize again that non-volatile memory proves superior to volatile memory only during unplanned power failures. For all other cases, there is no added benefit of NVRAM over RAM except for faster boot up times. During a planned reboot cycle, volatile RAM regions can also preserve data by saving the state on disk during shutdown and restoring it on start up. Many operating systems provide a similar feature called as hibernate. Therefore, the main task at hand is to restore the system state in NVRAM after start up, if the power is lost at a random point in runtime.

This issue is not completely new, and filesystems have solved a significant part of the problem. A separate NVRAM area can be set aside by the hypervisor for maintaining metadata structures. On system boot up, the NV memory manager would need to read the metadata and recreate the NVRAM memory mappings present before the system was powered off. This state restoration should incorporate page allocations and hypervisor page table mappings of all virtual machines, along with their data, because the system needs to guarantee their data persistence for power failures during random runtime conditions. It can be troublesome because just after power up, a VM does not have any existence except its config file and disk space, while the NVRAM area still remains active.

One possible solution is to divide NVRAM metadata section into several sections, with each running VM occupying one section. The hypervisor is also provided with a separate section for storing its relevant data structures. The VMs store their hypervisor page tables in this metadata section along with the location of a file where the contents are to be paged out if the VM is turned off. The file location can be specified and passed through the config file associated with that domain. Xen, in its metadata section, can store all the data structures pertaining to the non-volatile memory management unit, such as page_struct, _nvm_heap, etc. During the startup process, the hypervisor would read the metadata to determine the state of NVRAM and the amount of free memory available in the NV memory pool. Therefore, instead of rebuilding the data structures, Xen would just need to update the associated memory pointers. In the event of an improper system shutdown, several VMs may not have exited properly and could still be occupying the allocated non-volatile memory. As a part of boot up operations, the hypervisor can page out those NVRAM regions to the associated files on disk to free up redundant space and restore the system state using the method outlined before.

REFERENCES

[1] ARXCIS Development Kit. Retrieved January 11, 2015 from `http://www.vikingtechnology.com/uploads/arxcis_adk.pdf`.

[2] Katelin Bailey, Luis Ceze, Steven D Gribble, and Henry M Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pages 2–2, Napa, CA, USA, 2011. USENIX Association.

[3] Mary Baker, Satoshi Asami, Etienne Deprit, John Ouseterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 10–22, New York, NY, USA, 1992. ACM.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[5] David Bondurant. Ferroelectronic ram memory family for critical data storage. *Ferroelectrics*, 112(1):273–282, 1990.

[6] Greg Boss, Padma Malladi, Dennis Quan, Linda Legregni, and Harold Hall. Cloud computing. *IBM white paper, Version*, 1, 2007.

[7] Geoffrey W Burr, Bülent N Kurdi, J Campbell Scott, Chung Hon Lam, Kailash Gopalakrishnan, and Rohit S Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4.5):449–464, 2008.

[8] Meng-Fan Chang, Pi-Feng Chiu, and Shyh-Shyuan Sheu. Circuit design challenges in embedded memory and resistive ram (rram) for mobile soc and 3d-ic. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ASPDAC '11, pages 197–203, Piscataway, NJ, USA, 2011. IEEE Press.

[9] Meng-Fan Chang, Pi-Feng Chiu, Wei-Cheng Wu, Ching-Hao Chuang, and Shyh-Shyuan Sheu. Challenges and trends in low-power 3d die-stacked ic designs using ram, memristor logic, and resistive memory (reram). In *2011 IEEE 9th International Conference on ASIC (ASICON)*, pages 299–302, Xiamen, China, 2011. IEEE.

[10] David Chisnall. *The Definitive Guide to the Xen Hypervisor.* Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2007.

[11] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Helen Li, and Yiran Chen. Circuit and microarchitecture evaluation of 3d stacking magnetic ram (mram) as a universal memory replacement. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 554–559, Anaheim, CA, USA, 2008. IEEE.

[12] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: Architecture, design and implementation. In *Workshop on I/O Virtualization*, San Diego, CA, USA, 2008.

[13] A Driskill-Smith, D Apalkov, V Nikitin, X Tang, S Watts, D Lottis, K Moon, A Khvalkovskiy, R Kawakami, X Luo, et al. Latest advances and roadmap for in-plane and perpendicular stt-ram. In *3rd IEEE International Memory Workshop (IMW)*, pages 1–3, Monterey, CA, USA, 2011. IEEE.

[14] Richard F Freitas and Winfried W Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4.5):439–447, 2008.

[15] Henry F Huang and Tao Jiang. Design and implementation of flash based nvdimm. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2014 IEEE*, pages 1–6, Chongqing, China, 2014. IEEE.

[16] Koichiro Inomata. Present and future of magnetic ram technology. *IEICE transactions on electronics*, 84(6):740–746, 2001.

[17] Intel Corporation. *Intel® Virtualization Technology for Directed I/O*, October 2014. Revision 2.3.

[18] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, January 2015.

[19] Philip Jacob, Okan Erdogan, Aamir Zia, Paul M Belemjian, Russell P Kraft, and John F McDonald. Predicting the performance of a 3d processor-memory chip stack. *Design & Test of Computers, IEEE*, 22(6):540–547, 2005.

[20] Jiuxing Liu, Wei Huang, Bülent Abali, and Dhabaleswar K Panda. High performance vmm-bypass i/o in virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 29–42, Boston, MA, USA, 2006.

[21] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[22] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[23] Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *2010 the Sec-*

ond International Conference on Computer and Network Technology (ICCNT), pages 222–226, Bangkok, Thailand, 2010. IEEE.

[24] Jeffrey Shafer. I/o virtualization bottlenecks in cloud computing today. In *Proceedings of the 2nd conference on I/O virtualization*, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[25] James E Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[26] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[27] James H Strickler and Watt W Webb. Three-dimensional optical data storage in refractive media by two-photon point excitation. *Optics Letters*, 16(22):1780–1782, 1991.

[28] UEFI Forum. *Advanced Configuration and Power Interface Specification*, November 2013. Revision 5.0 A.

[29] Xen 4.1.5. Retrieved August 15, 2014 from `http://www.xenproject.org/downloads/xen-archives/supported-xen-41-series/xen-415/285-xen-415-1/file.html`.

[30] Jung H Yoon, Hillery C Hunter, and Gary A Tressler. Flash & dram si scaling challenges, emerging non-volatile memory technology enablement - implications to enterprise storage and server compute systems, 2013. Presented at Flash Memory Summit 2013.

[31] Xiantao Zhang and Yaozu Dong. Optimizing xen vmm based on intel® virtualization technology. In *International Conference on Internet Computing in*

Science and Engineering (ICICSE'08), pages 367–374, Habrin, China, 2008. IEEE.