Yu Zhang

Performance Improvement of Hypervisors for HPC Workload

Yu Zhang

Performance Improvement of Hypervisors for HPC Workload

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Universitätsverlag Chemnitz
2018

Impressum

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Angaben sind im Internet über http://www.dnb.de abrufbar.

Titelgrafik: Yu Zhang
Satz/Layout: Yu Zhang

Technische Universität Chemnitz/Universitätsbibliothek
Universitätsverlag Chemnitz
09107 Chemnitz
https://www.tu-chemnitz.de/ub/univerlag

readbox unipress
in der readbox publishing GmbH
Am Hawerkamp 31
48155 Münster
http://unipress.readbox.net

ISBN 978-3-96100-069-2

http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa2-318258

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Performance Improvement of Hypervisors for HPC Workload

Dissertation
zur Erlangung des akademischen Grades

Dr.-Ing

Herr M. Eng Yu Zhang
geboren am 7.Juli 1980 in Huhhot, China

Fakultät für Informatik
an der Technischen Universität Chemnitz

Dekan :	Prof. Dr.-rer. nat. Wolfram Hardt
Gutachter :	Prof. Dr.-Ing. habil. Matthias Werner
	Prof. Dr.-Ing. Alejandro Masrur

Tag der Einreichung :	30.01.2018
Tag der Verteidigung :	03.07.2018

Zhang, Yu
Performance Improvement of Hypervisors for HPC Workload
Dissertation, Fakultät für Informatik
Technical University of Chemnitz, Januar 2019

Acknowledgments

With eight years efforts, the PhD. study is slowly drawing to an end. Many people have rendered their help, expressed concern, and encouraged me. All these had strengthened my confidence to overcome the difficulties all along the way and go so far to create this work.

First, I sincerely thank my former supervisor, Prof. Wolfgang Rehm, who had offered me the opportunity to pursue the PhD study in TU Chemnitz, arranged the equipments and environment that are necessary for scientific research. More importantly, under the guidance of Prof. Rehm, I was ushered to the domains of Computer Architecture, HPC and System Virtualization. Efforts on a topic of these aspects lead to this dissertation.

Second, I would like to express my heartful gratitude to my current supervisor, Prof. Matthias Werner, who has sincerely rendered his academic supervision on my research topic and arranged a very comfortable environment for me to complete this research task. Without Prof. Werner's generous support, I cannot image to complete this research. As an earnest scholar and upright man, Prof. Werner really impressed me.

My thanks extend to my colleagues in the two professors research groups. René Oertel, Nico Mittenzwey, Hendrik Nöll and Johannes Hiltscher had rendered me invaluable suggestions on many issues, from equipment, publication to professional technical skills to my stay in Chemnitz.

With his patience and experience, Dr. Peter Tröger helped me to avoid quite a lot of incorrect ideas for research and drafting each chapter of this dissertation. I benefited enormously from his comments, feedback, and the conversation with him.

It is always my best memory to attend the conferences of EMS 2013 in Manchester, BigData-Science'14 in Beijing, Euro-Par VHPC'15 in Vienna, and ISC VHPC'17 in Frankfurt. I really feel gratitude for many unknown reviewers who evaluate my contributions positively to give me the chances to contact with world's excellent researchers and scholars. During these conferences, the chairs, Prof. David Al-Dabass, Dr. Alvin Chin, and Dr. Michael Alexander, had very nice talks with me. Dr. Andrew J. Younge, whom I get to know at ISC VHPC'17, gave me valuable constructive comments for a portion of this manuscript.

Prof. Andreas Goerdt encourages me to keep on even at the tough moments. Other friends, Wang Jian, Chou Chih-Ying, and Bai Qiong supported me all through the past eight years. They are true friends with whom I can share joy and frustration.

Furthermore, I would like to mention the China Scholarship Council (CSC), whose financing makes most of this research possible and the Deutscher Akademischer Austauschdienst (DAAD), who also sponsored me in the framework of a research project in the last year.

This manuscript is finally proofread by Ms. Jennifer Hofmann and Ms. Christine Jakobs. They took great patience to scrutinize the lengthy text, correct the grammatical and typographical errors, punctuation, spelling, and inconsistencies. Their efforts had helped me to produce a significantly clearer and more professional scientific work. A few lines of words could never be enough to express my gratitude.

Finally, I express my gratitude to my parents and my brother, who silently supported me all through the years with their love, concern and understanding from thousands of miles away. I dedicate this book to my father, who loved me so much but left me forever before the submission.

Abstract

The virtualization technology has many excellent features beneficial for today's high-performance computing (HPC). It enables more flexible and effective utilization of the computing resources. However, a major barrier for its wide acceptance in HPC domain lies in the relative large performance loss for workloads. Of the major performance-influencing factors, memory management subsystem for virtual machines is a potential source of performance loss.

Many efforts have been invested in seeking the solutions to reduce the performance overhead in guest memory address translation process. This work contributes two novel solutions - "DPMS" and "STDP". Both of them are presented conceptually and implemented partially for a hypervisor - KVM. The benchmark results for DPMS show that the performance for a number of workloads that are sensitive to paging methods can be more or less improved through the adoption of this solution. STDP illustrates that it is feasible to reduce the performance overhead in the second-dimension paging for those workloads that cannot make good use of the TLB.

Zusammenfassung

Virtualisierungstechnologie verfügt über viele hervorragende Eigenschaften, die für das heutige Hochleistungsrechnen von Vorteil sind. Es ermöglicht eine flexiblere und effektivere Nutzung der Rechenressourcen. Ein Haupthindernis für Akzeptanz in der HPC-Domäne liegt jedoch in dem relativ großen Leistungsverlust für Workloads. Von den wichtigsten leistungsbeeinflussenden Faktoren ist das Speicherverwaltung-Subsystem für virtuelle Maschinen eine potenzielle Quelle der Leistungsverluste.

Es wurden viele Anstrengungen unternommen, um Lösungen zu finden, die den Leistungsaufwand beim Konvertieren von Gastspeicheradressen reduzieren. Diese Arbeit liefert zwei neue Lösungen „DPMS“ und „STDP“. Beide werden konzeptionell vorgestellt und teilweise für einen Hypervisor - KVM - implementiert. Die Benchmark-Ergebnisse für DPMS zeigen, dass die Leistung für eine Reihe von pagingverfahren-spezifischen Workloads durch die Einführung dieser Lösung mehr oder weniger verbessert werden kann. STDP veranschaulicht, dass es möglich ist, den Leistungsaufwand im zweidimensionale Paging für diejenigen Workloads zu reduzieren, die die von dem TLB anbietende Vorteile nicht gut ausnutzen können.

Contents

List of Figures

List of Tables

List of Abbreviations

FLOPS	floating-point operations per second
MIPS	millions of instructions per second
NPT	nested page table
VMM	virtual machine monitor
ABI	application binary interface
AI	artificial intelligence
API	application programming interface
BIOS	basic input/output system
CFD	computational fluid dynamics
CFS	completely fair scheduler
CMP	chip multi-processor
CPAIMD	Car-Parrinello Ab Initio Molecular Dynamics
CPU	central processing unit
CUDA	compute unified device architecture
DBT	dynamic binary translation
DDR	double data rate
DFT	Density Functional Theory
DIMM	dual-in-line memory module
DPMS	dynamic paging method switching
EC2	elastic compute cloud
EPTP	extended page table pointer
EPT	extended page table
FFT	fast fourier transformation
FPGA	field programmable gate array
GPA	guest physical address
GPGPU	general purpose graphic processing unit
GPT	guest page table
GPU	graphic processing unit
GUPS	giga updates per second
GVA	guest virtual address
HPA	host physical address
HPDA	high performance data analysis
HT	hyper transport
I/O	input/output
IC	integrated circuit
IOMMU	input output memory management unit
IPC	instruction per cycle
ISA	instruction set architecture
KVM	kernel virtual machine
LHP	lock holder preemption
LINPACK	LINear equations software PACKage
MMU	memory management unit
MPI	message passing interface
MPP	massively parallel processors
MSR	model-specific registers
NFS	network file system
NIC	network interface card
NLP	node-level parallelism
NUMA	non-uniform memory access
NWP	numerical weather prediction
OpenCL	open computing language
OS	operating system
PCI	peripheral component interconnect
PDE	partial differential equation
PFN	page frame number
PF	page fault
PGD	page global directory
PHD	page higher directory
PLD	page lower directory
PMC	performance monitor counter
PMD	page middle directory
PTE	page table entry
PTPC	page table pointer cache
PUD	page upper directory
PVM	parallel virtual machine
PWC	page walk cache
QOM	QEMU object model
QPI	quick path interconnect
REML	restricted maximum likelihood
RMS	recognition, mining and synthesis
RVI	rapid virtualization indexing

SLL	shared last level
SMP	symmetric multiprocessing
SR-IOV	single root input output virtualization
STDP	simplified two dimensional paging
SVM	secure virtual machine
TDDFT	time-dependent density functional theory
TDP	two dimensional paging
TLBM	translation lookaside block miss
TLB	translate look-aside block
TLP	thread level parallelism
TMR	TLB-miss rate
USB	universal serial bus
VALE	virtual local ethernet
VCPU	virtual central processing unit
VirtIO	virtual input and output
VMCB	virtual machine control block
VMCS	virtual machine control structure
VMD	visual molecular dynamics
VM	virtual machine
VR	virtual reality
VT	virtualization technology

Chapter 1 Introduction

Contents

This chapter provides the background, describes the problem, the contributions and the outline of this thesis. First, high performance computing (HPC) is briefly introduced with the focus on its major concerns. Then, the system virtualization technology is introduced to address such concerns posed in the evolution of HPC. However, the adoption of this virtualization technology in HPC has also problems, of which the performance loss is the major one. This thesis focuses on the performance loss in guest memory address translation and attempts to work out solutions for this problem.

1.1 Background

1.1.1 HPC and its Major Concerns

High performance computing has been around for a few decades. This is a branch of computing that tackles the most complex and challenging problems arisen in science and engineering. By solving fundamental research problems with HPC, human makes scientific discoveries and pushes forward the frontier of knowledge. As a powerful tool for research and production, HPC makes use of the cutting-edge hardware, therefore represents the most high-end computing technology. Breakthroughs achieved in HPC may also benefit other branches of the computing community. In this sense, HPC is a key area and has potential significance for computing as well as a broad range of science and technology.

From a technical perspective, the transformation from monolithic mainframes to cluster-based supercomputers is a major breakthrough in the development of HPC system. While this enables more flexible way of construction, a scalable performance and a lower investment for the facility, the growing model for the HPC systems is increasingly meeting serious challenges. Generally, challenges may be posed by the following concerns about an HPC system:

Performance

Intrinsically, the performance of a computer is generally understood as the speed achievable by the computer in accomplishing tasks. For a workload with fixed amount of task, performance can

be indicated by the execution time. The longer the execution time is, the lower the performance will be, hence the relation [1],

$$performance = 1/execution\ time$$

Performance is not only a major concern in computing, but also the key driving force behind the advance in computer architecture [1] and the primary target for computer design. A great deal of efforts are dedicated to the study of performance in computer architecture.

Depending on the purpose of research, the performance of a computer can be measured for the whole system, such as the LINPACK results for a supercomputer or for an individual component. The result can be either an absolute figure indicating the performance of a given hardware and software or a ratio for comparing the influence due to different systems or different configurations for the same system.

From a system engineering point of view, performance is a non-functional quality but depends on many functional qualities. According to Dongarra et al., "the performance of a computer is a complicated issue, a function of many interrelated quantities" [2]. These factors embrace a broad range of issues in hardware and software, including:

The frequency of processor cores, the size, level of caches, the size, level, and associativity of TLBs, the frequency, size of memory chips, the bandwidth and latency of the network card, the frequency of memory bus and PCI bus, the volume and speed of storages, the size of the program, the programming language and algorithm the program applies, the compiler's capability for code generation and optimization, the static and dynamic libraries, the operating system, the organization and architecture of the computer or computing system and so on.

The overall performance of an application is determined by the interaction between the entire instructions and the underlying hardware.

As a result of the enhancements in circuit and architecture technology, processor, memory and I/O devices saw unprecedented increase in speed and capacity. Figure 1.1 depicts the evolution of processors in the past few decades. Alongside with the decrease in process and in power (energy consumption per time unit) is the exponential increase in clock speed (frequency), number of transistors per die, and speed of instruction execution (performance). Since the flourishing of microprocessor and the integrated circuit (IC) industry, these trends accord with a famous "rule of thumb" - the *Moore's law*, which predicts that the number of components per integrated circuit doubles approximately every two years. According to Intel's former executive, David House, the processor chip's performance would double every eighteen months, as a resultant of both the increased number of transistors, and the enhanced speed of them [3]. The prediction proved fairly accurate during the past decades. However, in the recent decade, the shrinking of transistor size is almost approaching the atom scale, posing physical barrier for further shrinking.

While there are discussions about the effectiveness of the *Moore's law* as the general roadmap for semiconductor industry in the future [5], processor has begun to follow the multi-core approach to continue growing its performance by adopting spatial redundancy of the hardware. Although the clock speed comes to a stagnation, the theoretical performance can still be scaled by the number of cores. This becomes the main reason why the *Moore's law* still holds true in a multi-core era. Such an architectural change has a fundamental impact on the way of processor utilization. To make effective use of the parallel hardware, software have to be developed in a parallel and multi-core approach, instead of the traditional sequential and uni-core approach. The actual performance and effective utilization lie in the scalability of the software across cores.

From a system point of view, not only the processor, but also the other computer components, as well as the architecture innovations have contributed to the performance growth. In addition to the multi-core technology, the following architectural technologies also push forward the overall performance [1]:

[1] In practice, other notations such as the number of floating-point operations per second (FLOPS), or millions of instructions per second (MIPS) are also used.

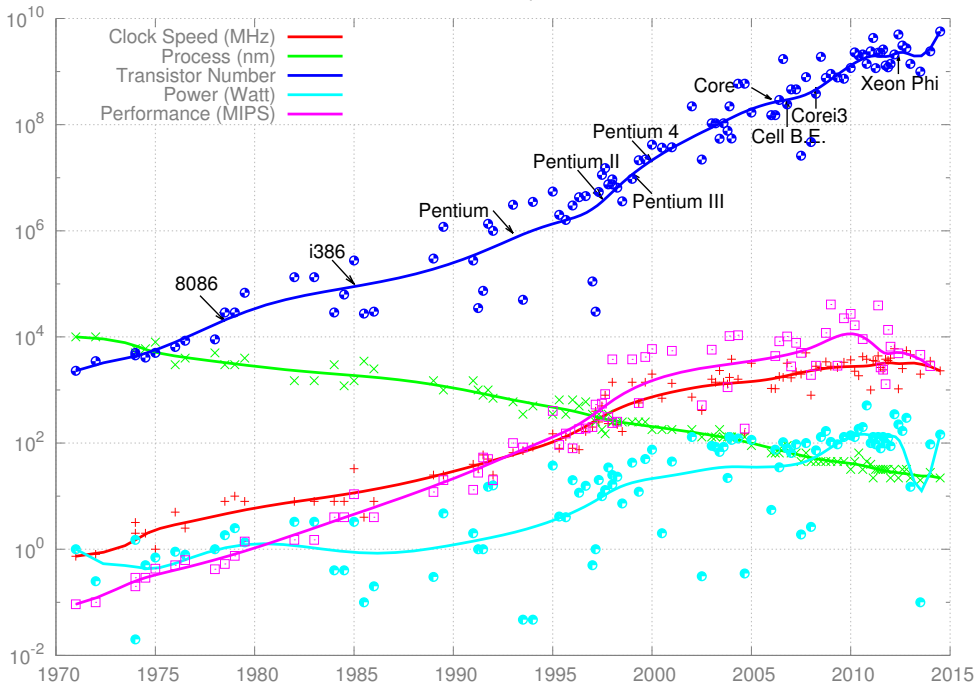


Figure 1.1 Evolution of the processors [4]

- Pipelining and super-pipelining
- Multiple levels of cache memory
- RISC architectures
- Multiple execution units (single data, multiple instruction)
- ISA extension (to take advantage of single instruction, multiple data)

In the recent decade, additional novel technologies continued boosting the overall performance of the computer and computing system, including [1]:

- Multithreading (super-/hyper-threading to exploit the *thread level parallelism*)
- Speculation and prediction mechanism (to take the advantage of idle execution units)
- Hardware accelerating (inside and outside the processor chip)
- Vector and array processing
- Large-scale parallel and distributed computing

Figure 1.2 illustrates the performance growth of the most powerful computing systems in the world, the top 500 supercomputers, since the flourishing of HPC industry. The performance of the top one system has seen a growing from tens of GFLOPS to tens of PFLOPS, and is looking forward to EFLOPS (exascale[2]) era.

Power Consumption and Efficiency

Despite the steady technical advances in processor and architecture, the expansion in dimension still remains as a major source of performance growing. While large-scale parallel and distributed system emerge as the popular computing architecture and the power consumption per processor (core) is decreasing, the power consumed by the whole system is still tremendous. Power consumption and power efficiency become increasingly a concern since a decade ago. Nowadays they are emerging as the critical limiting factors for high performance computing.

[2]exa is a prefix denoting a factor of 10^{18}.

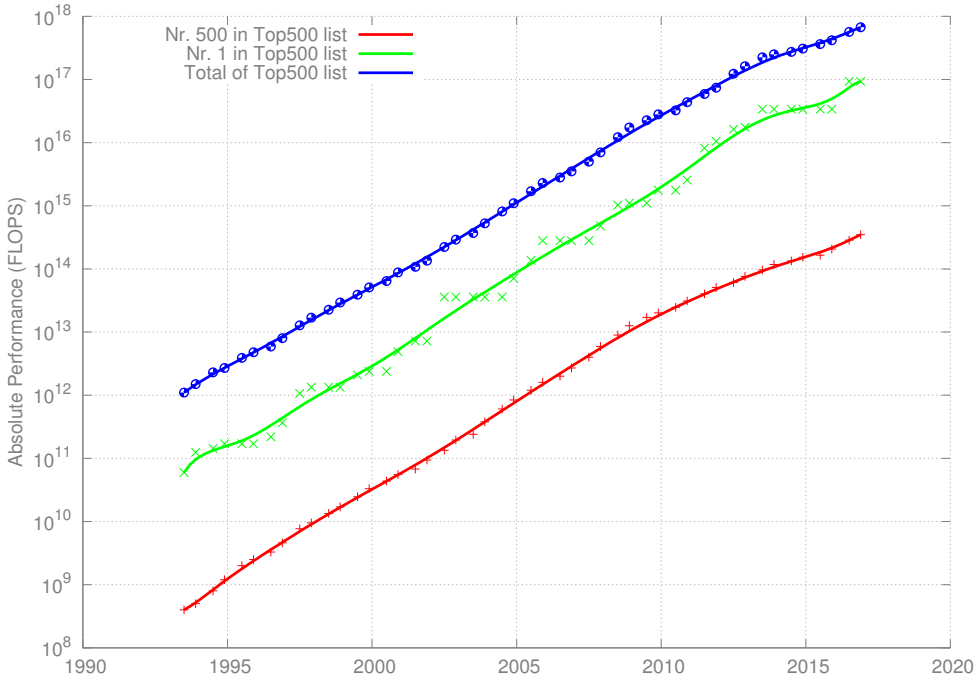


Figure 1.2 Growth of the supercomputer performance [6]

Supercomputers or HPC clusters consume energy not only for computing, but also for cooling to prevent the hazard of excessive heating. Power supply and budget add heavy burden to the computing industry. Such a developing model is unhealthy and unsustainable in the long run. Figure 1.3(a) depicts the evolution of the top 10 fastest supercomputers among the top 500 list since an early age from total performance, total number of cores, and total power aspects. These had undergone rapid growth and currently reached magnitudes unimaginable in daily life. While the total performance is approaching exascale by using tens of millions of cores, the total power is amounting to hundred Megawatts (10^6 Watts), roughly 0.44% of the installed electrical capacity in the world's currently largest hydro-electrical power station[3].

Figure 1.3(b), on the other hand, depicts the efficiency aspects in the same course. Efficiency refers to the ratio of output and input. The power efficiency, core (hardware) efficiency, as well as the "power draw per core" are compared over a relatively long period of time. Power efficiency measures the number of floating-point operations per watt, which has reached at a level of 2.8966 GFLOPS for the top 10 supercomputers. It is about 31.61% of the currently most power-efficient system (NVIDIA DGX-1, 9.4621 GFLOPS [8]). "Power draw per core" measures power consumption per processing unit (core) and indicates the current technical status for energy saving in processor design and manufacture. In the past decade, this has been reduced to a level of 6.81% than at the early age (7.05 W/core in June, 2006, 0.48 W/core in November, 2016). This contributed enormously to curb the drastic growth of power demands for the entire computing industry over the past decade.

[3]Three Gorges Dam, currently the largest hydro-electric power station, and the largest power producing body ever built, has an installed electrical capacity of 22,500 Megawatts [7].

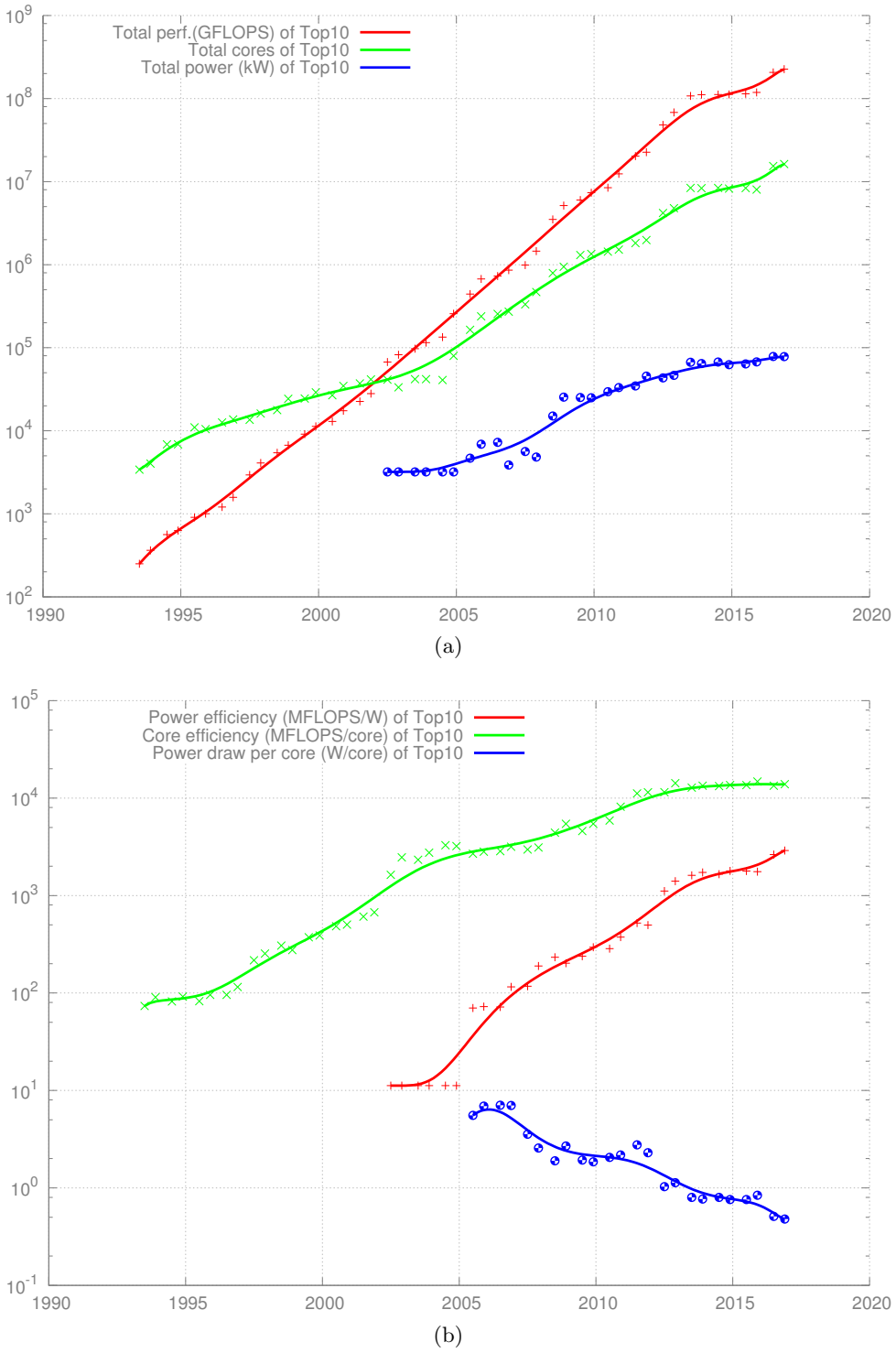


Figure 1.3 Performance, power, scale of supercomputers (a) total amount, (b) efficiency [6]

Hardware Utilization and Efficiency

Figure 1.3(b) also depicts the core (hardware) efficiency, which has increased by a magnitude of two orders since the initial stage of high-performance computing industry. As this is the average of all types of the concerned processing units, it represents the general growing trend of the HPC technology. In the recent years, the core (hardware) efficiency reached a plateau after climbing a slope, implying that the performance is roughly proportional to the number of cores (the scale of computing system). To limit the computing system within a reasonable scale, the strategy is to increase the "performance per unit chip area".

Another concern is the hardware utilization. Since long ago, computer, especially the servers in large data centers are plagued with the underutilization problem, which prevents the effective use of the resources and lowers the profits in production. The server farm in a data center, for example, may suffer huge profit loss due to insufficient utilization of resources. Owners of such infrastructures are in a dilemma. As reliable service providers, they are expected to maintain higher availability of their services, at best 100% of the time without any downtime. On the other hand, the influx of workloads may fluctuate from time to time, thus is unpredictable. The workload may be distributed unevenly among servers, which creates load imbalance from server to server. The consequence is that even the underutilized or idle servers had to stay powered on, with less or no productivity at all. Studies indicate that a typical server is estimated to have 8% to 30% of its uptime for running applications on average, with the rest portion of time simply wasted by running idly (Gartner Group)[9]. Similar thing occurs in companies with production servers. It is estimated that on average 15% of the full-time servers in such companies perform no useful tasks [10], leading to a huge waste of both hardware and energy worldwide.

In high-performance computing, underutilization problem seems not to be acute as that in case of servers of data centers. This is probably due to the fact that HPC has much smaller user groups that generate dedicated tasks in a more homogeneous way. However, since the supercomputer carries a huge permanent investment but has much shorter life-span than normal servers, its utilization tends to be calculated by the unit of "node hour". Idle and underloaded states of the system simply mean lower production, lower economic efficiency, and add the cost per-unit of service the system provides. In this sense, higher utilization of the hardware resources means cost saving and economy promotion for high-performance computing.

1.1.2 System Virtualization

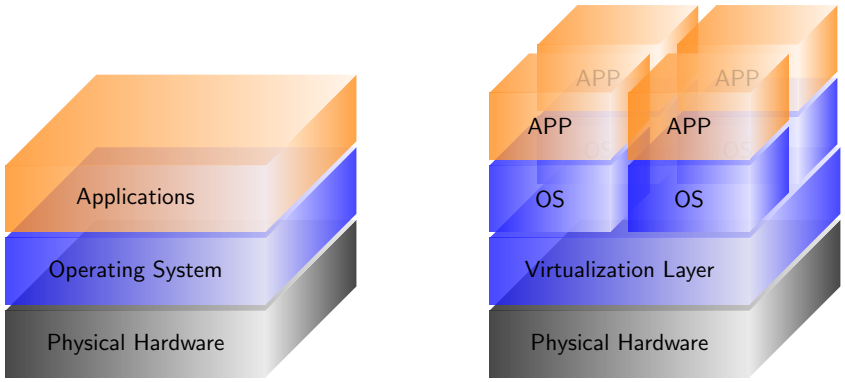
Overview of Virtualization

In computer science, a *virtual object* is a concept used to refer to a kind of logical entity created on the basis of the real physical entity. According to the definition by Popek and Goldberg, "a virtual machine is taken to be an *efficient*, *isolated duplicate* of the real machine." [11] Virtualization refers to the act of creating a virtual (rather than actual) entity, such as virtual computer hardware platforms, storage devices, and computer network resources [12]. Its purpose is to extend or replace an existing interface to mimic the behavior of another system [13].

In practice, virtualization may occur at several different levels in accordance with the abstraction layers in the architecture of a computer, such as hardware level (ISA), operating system level (ABI) and library level (API) [14]. System virtualization is the virtualization at the hardware level. The entity to virtualize is the entire hardware of a computer.

One effect that system virtualization creates for computer architecture is illustrated by Figure 1.4. Virtualization enables the *decoupling* between the OS and the hardware, as well as that between the applications and the OS. These are the bases for almost all other features.

Although the concept of virtualization and especially of virtual machines were in the focus of computer scientists and engineers in the recent two decades, their origins go back to the first generation of computers in the 1960s [15]. Due to the prohibitive-high prices and a centralized, non-interactive way of using the mainframe computers, the expensive computing resource tended to suffer from underutilization problem. A variety of time-sharing, multi-tasking, multi-user systems were created to boost the resource utilization and ease the use. Virtual machine comes as a more advanced variant of that technology. System resource can be partitioned into independent fine grains and assigned to multiple users running multiple tasks in an encapsulated, isolated way. This is an excellent feature, which makes virtual machine quite useful for boosting the computer system resource utilization. Furthermore, more benefits are also brought by virtualization for computing, including:



(a) (b)

Figure 1.4 System virtualization (a) traditional architecture, (b) virtual architecture

Virtualization's Benefits

- **Better customization** The nature of encapsulation and isolation permit customizable execution environment for the user applications.
- **Higher scalability** Physical servers can be set up easily by rapid and dynamic provision. Deploying a new virtual machine is as easy as copying a file. The whole computing system becomes much more scalable by adding or removing virtual machine guests.
- **Higher flexibility** Workload on a physical machine can be migrated, check-pointed, resumed. Even the virtual machines can be reconfigured dynamically. New workloads can be deployed more quicly and conveniently than on physical machines.
- **Higher security** As workloads in one virtual machine are encapsulated and isolated in a single execution environment, malicious attack or accidental mishap in one virtual machine cannot propagated to other virtual machines or bring down the whole physical machine.
- **Less hardware and energy cost** The same amount of task performed by a number of virtual machines results to a reduced amount of physical hardware as well as power consumption.
- **Less deployment effort** Physical servers can be quickly loaded by standardized virtual machine image (VM provisioning).

Virtualization's Limitations

Despite the potential benefits and competitiveness virtualization brings for computing, the adoption of virtualization for computing has also problems. Taking the data center, where the virtualization technology is widely adopted, as an example, the whole system is inevitably complicated due to the presence of a virtualization layer.

A system consisted of both physical and virtual infrastructures also pose great challenge to the hardware and maintainer. According to a number of representative articles [16, 17, 18] by IT professionals, analysts and vendors, the major challenges faced by data centers are the following:

- **VM sprawling** Originally the use of virtual machine were intended to reduce the number of physical servers in data center. However, probably due to the convenience and low cost of deploying virtual servers, benefits are eroded by the overgrown virtual servers in the course of poorly planned VM life cycle and placement. Storages may be depleted quickly by many less-used VM images. Network switches get saturated by the unplanned overwhelming data traffic floods among virtual servers. Network becomes complexed and too often dynamically reconfigured by the virtual network switches. Troubleshooting becomes harder in multiple-layer virtual environments. The physical servers will be overburdened by carelessly launched migration and consolidation. Consistent security policy is more difficult to be enforced in an infrastructure composed of physical and virtual servers.

- **Physical and geographical restrictions** For the VM migration and server consolidation, virtual environment places more restrictions on hardware. For example, server hardware for migration purpose should have some special features such as identical PCI and NIC supports. A data center is expected to situate within a maximum distance to the neighboring one as the source or destination for migration. So sufficient bandwidth and latency can be ensured.
- **Lack of standard and interoperability among vendors of hypervisors** For data centers, hypervisors play the central role. Although VMware platform is still dominating the x86 server virtualization, it is increasingly common to find out data centers based on other hypervisors such as Citrix XenServer [19], Red Hat KVM [20], or Microsoft Hyper-V [21]. Various vendors apply their own management tools for task and resource controlling. But few of them take care of the interoperability with products from other vendors. Instead they normally come up with features which may not be available for hypervisors from other vendors. Maintainers need more efforts to get used to different platforms.
- **Lack of freedom to move between x86 processors** Although the two major x86-based processor vendors - Intel and AMD share the same ISA, the implementations of the ISA have subtle differences, which leads to different behavior for a few instructions. Furthermore, each of them had also come up with their own virtualization extensions. The hypervisor developers need to provide vendor-specific solutions for both Intel and AMD processors. While this poses no problem for VM "cold" migration, it does so for VM life migration.
- **Vulnerability at storage and network switches** A data center is a large array of servers. The SAN (storage area network) and network switches bear the stress of data flow among a large number of physical and virtual servers. This makes them easy to get stuck or congested by data floods. Centralized components like the storage and network are by nature vulnerable to the high I/O traffic throughput.
- **Obscured virtual server security** To maintain the illusion of transparency for end-users, all physical and virtual servers need to have their identifications somewhat merged in a cloud environment. Therefore, the entities and environments become more dynamic, fluid and volatile. The boundary for security in the infrastructure may be blurred. Not only the physical layer, but also the virtual layer needs to be patched when necessary.
- **Software license impact** For the data centers running commercial hypervisors, the license could be an important consideration. Not only for the huge amount of charge, but also for the strong impact on the virtual infrastructure delivered by the license terms and pricing models. Taking the VMware product, vSphere[4], as an example [22]. The vSphere version 4 license charge is based on a combination of physical processor cores, sockets, and physical memory volume installed on the server, with 1 socket, 6 cores, and 256 GB memory as the standard charging level. In contrast, the version 5 standard changed to a pricing model based on virtual rather than physical memory and allows a total of only 24 GB to allocate to all VMs, and up to 8 VCPUs per VM. While the former version encourages "scale-up", running as many VM guests as possible on a system with fewer sockets but more memory, the latter penalizes this way by "scale-out", purchasing more physical servers each with far less physical memory. A single move in license terms has actually led to a three times higher price than previously. This has driven many of the VMware customers to abandon the vSphere and turn to Citrix XenServer or Microsoft Hyper-V.
- **Degradation of the performance** Last but not least, virtualization may suffer more or less native performance. By empirical research in early years [23], a rough estimation of the performance for the major components in a virtual execution environment is: CPU 96-97%, Network 70-90%, and Disk 40-70% of the native performance. Although this estimation is made over a decade ago and may not reflect the current status, large performance loss is still not uncommon for virtualization.

[4] VMware vSphere is a softwares package, including vSphere client, vCenter server and ESXi hypervisor.

1.1.3 Exploit Virtualization in HPC

The concerns mentioned in Section 1.1.1 exist commonly in the computing community, especially where large-scale IT infrastructures are used. High performance computing is not an exception. As the hardware facility is expanding, problems such as the resource underutilization, excessive power draw, and system management are posing increasingly as challenges for the HPC systems. The adoption of virtualization technology enables the HPC to abstract, pool and automate the resources so as to overcome such challenges due to the rigid physical limits in the HPC systems.

This makes sense especially considering that HPC is increasingly demanded by new groups of users other than research area. As a result, HPC systems are growing more commercial and more business-oriented. In such situation, business agility and enhanced security become urgent issues to deal with multiple non-trusted users. Virtualization's benefits for computing lie in:

Customizable Execution Environment

The virtualization technology's competence is basically rooted in its capability of reducing the strong dependency between the system software (OS) and the underlying physical hardware, as well as the strong dependency between the user application software and the system software.

Physical hardware, precisely the processor, is characterized by its ISA, the interface with the software. The software targeted at a given hardware platform must comply with the form of hardware's ISA. Similarly, an OS is characterized by its system calls, the interface with user applications. Applications need to comply with the system calls of the given OS for execution. These dependencies are known as *tight coupling*. Constrained by this, a server can be installed with a single OS only, thus bound to run application software only for the same type of OS and processor architecture. Such a bound leads to the "one server, one application" model.

Adoption of virtualization on the server platform has effectively broken these *tight couplings* and fundamentally changed this model. By inserting the hypervisor – an indirection layer between OS and hardware, a server (physical machine) acquires the capability of running multiple OSes (virtual machines) concurrently. Each of these virtual machines is isolated from each other and encapsulated as an independent execution environment. By deploying virtual HPC clusters in the supercomputer environment, the software stack for execution environment may achieve higher capability to be reconfigured or customized on demand of the users.

System Resource Utilization Enhancing

The "one server, one application" model constrains the service providers to put a single workload on a server for reliability and quality of service (QoS) reasons. Server is equipped with abundant computing resources for running multiple tasks. By multiplexing the computing resources among a reasonable number of virtual machines guests, the physical server gets sufficient tasks to stay busier and more productive. Each VM guest may be dedicated to a specific service as if it were a physical server.

The number of guests is known as *density*. Too small density brings quite limited increase in utilization. While too large density may saturate the physical server and degrade the quality of services in guests. A properly chosen density keeps the server reasonably loaded, meanwhile does not necessarily incur visible damage to the quality of service in each guest. By virtualizing physical server, productivity of the computing resource can be increased by exploiting idle time.

Power and Hardware Efficiency Enhancing

With virtualization of the computing resources, physical nodes in a supercomputer (HPC cluster) may be used by loading a number of VM guests as the virtual computing nodes. More than one applications are allowed to run concurrently in the single host. Due to *decoupling*, a virtual node is not bound to any physical node. An execution environment presents itself as a file stored

in hard disk or in memory, which is fairly easy to move from physical node to physical node via interconnect network, either by copying a static file on a disk (static migration) or a file on a disk together with the image in memory (dynamic migration). Load balance is achieved by migrating virtual nodes from overloaded physical nodes to underloaded physical nodes while power can be saved by migrating those sparsely spreading virtual nodes to fewer physical nodes, and powering off the rest (known as the VM consolidation). Both the hardware and power are utilized more productively, which enhances their efficiencies compared to physical case.

1.2 Problem Description

Traditionally, virtualization technology is not oriented to workloads that are resource-intensive. Hypervisor is optimized mainly for the domains where workloads have moderate and predictable demand for resources.

Among the barriers for adopting virtualization in HPC, performance loss is a major consideration [24, 25, 26, 27]. Among factors that incur performance loss of the HPC workload due to virtualization, memory address translation is a potential bottleneck [28], especially for workloads that are memory-intensive and exhibit unusual run-time behaviors. In dealing with the diverse HPC workloads, the currently adopted two standard solutions, *shadow paging* and *nested paging* for translating the VM guest memory address have the following limitations: 1) guest workloads are treated in the same way, despite their various characteristics and significantly different run-time behaviors in memory-accessing; 2) the default memory address translation way is not optimal when performance comes as the top priority.

Due to these limitations, the guest memory virtulization may not be optimized to yield non-negligible performance loss for a given workload in the guest. Research [29] indicates up to 40% of performance degradation for the memory intensive workloads (PASSMARK MEMORY) in the XenServer 5.5 HVM environment.

The goals of this work are: 1) To investigate the performance loss of typical HPC workload in virtualized execution environment due to guest memory virtualization; 2) To work out solutions to improve the performance of the typical HPC workload in virtualized execution environment.

1.3 Thesis Contribution to the Current Research

This research presents two solutions to work around the limitations for virtualizing the guest memory address translation. Namely, they are **DPMS** (dynamic paging method switching) and **STDP** (simplified two-dimensional paging), as elaborated below:

1. **DPMS** Overcomes the first limitation stated in Section 1.2. With this, the hypervisors can adjust its paging method for translating the guest memory address based on the decision made during the running of the guest applications. DPMS brings a new feature to the hypervisors, which enables hypervisors to react to the ever-changing behaviors exhibited by the running workload with an adaption to the optimal paging method. Workloads with different characteristics can be handled differently based on their run-time behavior in memory accessing. In this approach, the performance of the diverse HPC workloads, especially those memory-intensive workloads will be benefited.
2. **STDP** Overcomes the second limitation stated in Section 1.2. STDP is variant of the traditional TDP. As the bulk of the TDP overhead lies in traversing page tables in the second-dimension, to reduce the overall page-table walking overhead, a fundamental way is to adopt fewer level page tables in the second-dimension. STDP is therefore prompted as an innovation of the current TDP supported by both the hypervisor and the hardware processor. Without considering the TLB, STDP is able to reduce 40% to 60% of the paging overhead in the second dimension, which is a big advantage compared with the current TDP.

In a broad view, the thesis contribution is presented by the last row of Figure 1.5, which depicts the road map for the evolution of the system virtualization technology. Till now, the technology for virtualizing each component of a physical computer has undergone a few generations.

A fact identified is that in each generation, virtualization losts a part of virtualization in its essence. In other words, except for a few core functions which make virtualization "virtual", a significant portion of the functionality has been taken over by the bare-metal hardware. From the software-based pure emulation to the binary translation, to native execution of the instructions with occasional binary translation, from the software-based pure emulation to para-virtualized VirtIO to physical device pass-through, the performance for both the processor and I/O device virtualization gain impressive progress.

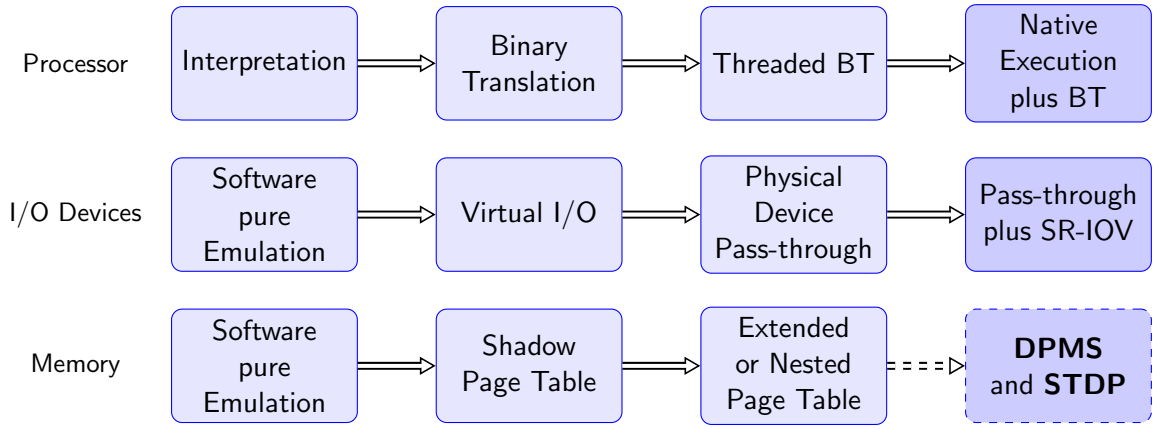


Figure 1.5 Problem domain on the road map of virtualization's evolution

In contrast, the potential breakthrough for memory virtualization occurs with the adoption of *nested paging*. However, despite the advantage over *shadow paging*, *nested paging* has also drawback. In this sense, the current *nested paging* serves as an alternative, but not yet an entire replacement of *shadow paging*. What is the next generation technique for memory virtualization parallel with the breakthroughs for both processor and I/O device virtualization?

For these reasons, the major efforts of this work are dedicated to memory virtualization, with a focus on how to improve the current solutions for such a narrow topic in the big picture. These efforts lead to the contributions mentioned above.

1.4 Outline of the Thesis

The thesis is organized like this: Chapter 1 presents the background and describes the research problem. Chapter 2 reviews the current related work in this area. Chapter 3 is engaged in seeking the performance drawbacks with a series of benchmarks on different testing platforms. Based on these results, Chapter 4 proposes two solutions for improving the hypervisors for HPC workloads. Chapter 5 works out an implementation for the proposed ideas based on a concrete hypervisor, QEMU-KVM. With the purpose of verifying the feasibility, Chapter 6 presents the results for a series of functional testing and performance benchmark. In Chapter 7, parallel with the major contributions in guest memory virtualization, efforts on the performance loss due to the processor and I/O virtualization are also mentioned. And finally, in Chapter 8, benchmark, design and implementation are concluded.

The thesis is outlined by a sequence of concrete questions below:

1. What is the current situation for using virtualization technology in HPC area?
2. What impact does virtualization deliver to the performance of HPC execution environment?
3. Which virtual execution environment may serve as an ideal platform for this research?
4. Which applications can be used to characterize HPC workload?
5. Which component of the virtualized system incurs acute performance issues?
6. What strategies, or measures can be proposed to remedy this?
7. How to implement these strategies in the context of a concrete hypervisor?
8. What is the effect of these proposed solutions?
9. How can these solutions be applied in related area?
10. What limitations do the new solutions have, and what can be done for further progress?

Figure 1.6 shows how the chapters are related with these questions.

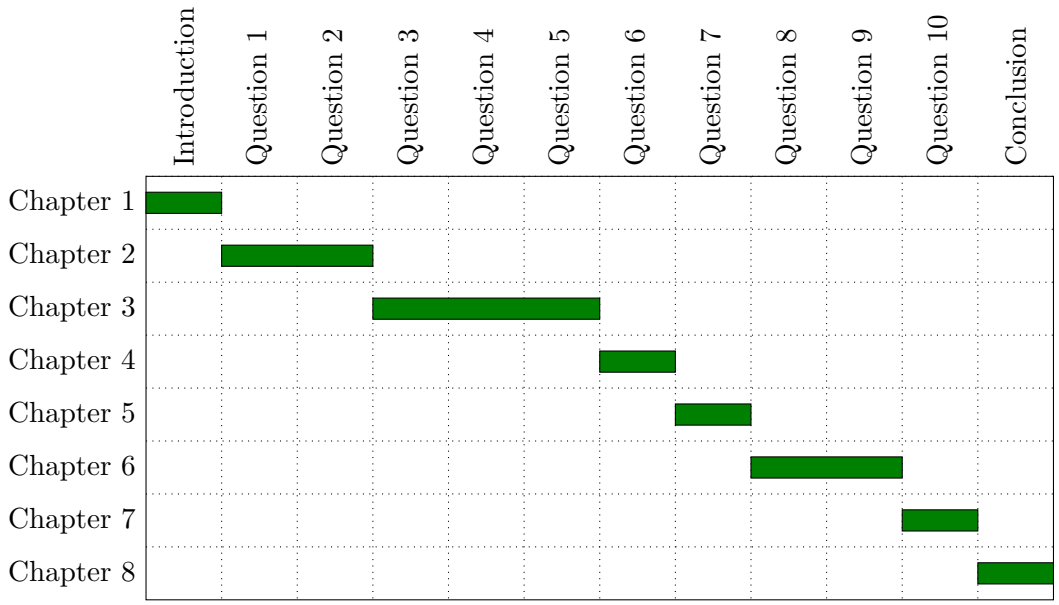


Figure 1.6 Outline of the thesis

Chapter 2 Related Work

Contents

This chapter sketches the topography and frontiers of the research area by exploring the related works. Section 2.1 reviews the current status for system virtualization technology. Section 2.2 examines how virtualization technology is utilized, and to what extent it has influenced the development of HPC. Section 2.3 surveys the efforts that are dedicated to the performance aspect of the memory virtualization in both the traditional and the virtualized execution environments. It particularly focuses on achievements and limitations for the major contributions in improving the virtualization of the guest memory address translation process. Finally, Section 2.4 draws a summary of the reviewed related work, which presents a clear view of the current work, and the context to this thesis.

2.1 System Virtualization

System virtualization [14] provides the capability of creating multiple execution environments (OS plus user applications) simultaneously on a single set of physical hardware. Corresponding to the implementation of virtualization layer, hypervisors[1] roughly fall into two types: type-1 and type-2 [30], as Figure 2.1(a) (c) depicts, respectively. At the time when the trend for virtualization took off, type-2 hypervisors were the most popular form, mainly due to the convenience of turning the underlying physical machine into a virtual machine by installing the type-2 hypervisor on the available OS. However, type-2 hypervisors have a few downsides, primarily in performance, which is the reason for the emerging of type-1 hypervisors. This type of hypervisors runs on the bare-metal hardware as an OS, rather than as an application in the user-space. Therefore it can assume full control over the physical hardware and ensure higher performance, availability and security than the type-2 hypervisors.

Regardless of the hypervisor type, the execution environment presented by the virtualization layer is a virtual machine for the guest OS and the applications running inside. Figure 2.1(b) depicts this effect, a virtual machine that hides details of the real execution environment. The difference is almost unperceivable by the guest.

As the engine of virtual machines, the hypervisor abstracts and multiplexes the physical hardware among guests. The hypervisor is a piece of software to create virtualization by presenting an illusion to the guest OS atop, as if they were running in a bare-metal hardware environment. Controlled by the hypervisor, multiple guest operating systems can coexist and execute simultaneously to exploit the same physical hardware by hosting their own applications. Determined

[1] Hypervisor is also known as the virtual machine monitor (VMM). They are often used interchangeably.

by the nature of each individual component, a computer applies a number of virtualization techniques. The virtualization techniques for the major components are examined below:

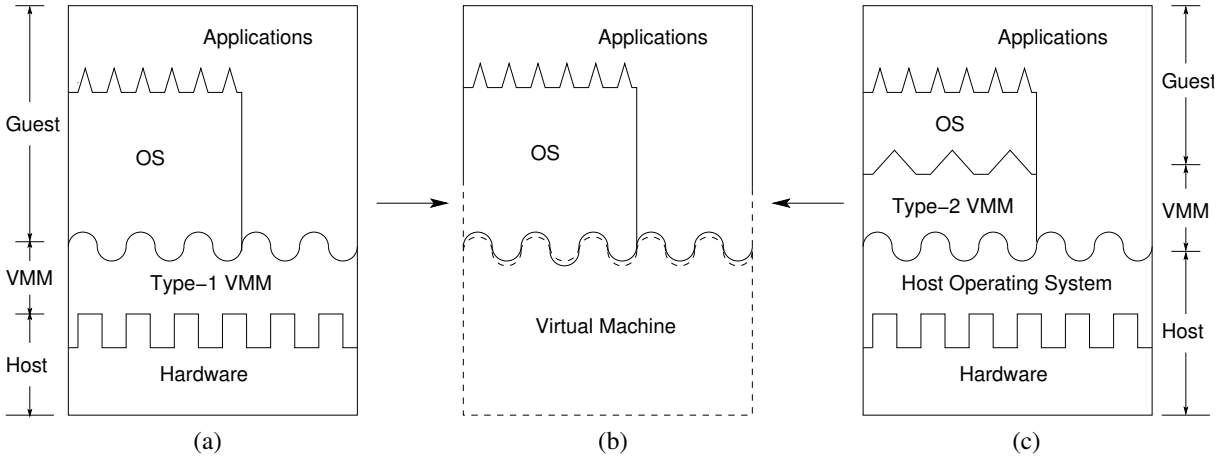


Figure 2.1 Stack layout of the virtual machine [14]

Processor: In the host, the processor is the central part and the primary target to virtualize. By nature, a processor is a device sharable in a temporal way among its tasks. In the traditional physical machine, a processor is utilized by running a specific task within a period of time (time slice). To support virtualization, processor can be virtualized by a number of approaches. Originally virtualization takes the form of emulation [31], which is able to run an unmodified OS and applications targeted to an architecture other than that of the underlying physical processor. The emulation belongs to virtualization in a broad sense, but is significantly different from the current virtualization approaches in many aspects. To run the software for non-native ISAs, the *instruction interpretation* and *binary translation* approaches are adopted. In emulations of these kinds, the physical processor is actually not emulated at the circuit or micro-architecture levels, but at the ISA level. The instructions of the source software are unable to execute on the underlying (target) platform. These approaches enable the semantic to be extracted from the source software and be reconstructed for the target ISA. Thus the software can execute on the target platform. Emulation is the most intuitive form of virtualization. The major problem is the poor performance due to *code expansion*[2][32]. On average, a ratio of 1:10 is expected by using a typical emulator or interpreter without caching [33]. With caching, it has ranged from 1:100 with the software simulators in the 1970s to 1:4 in 1990s [34].

Emulation is mainly used to address the cross-ISA problems between software and hardware. In the narrow sense of virtualization, however, the source and target ISAs are of the same, which permits simpler and more efficient ways for processor virtualization. Instead of the instruction interpretation and binary translation, efforts were made to execute the instructions of the source software directly on the target platform. As one of the primary targets for system virtualization, the "x86-based processor" has a few drawbacks in its ISA. This is because some instructions behave differently in virtual and physical environments. These include privileged instructions, behavior-sensitive instructions, as well as control-sensitive instructions [35].

According to the **classic virtualization condition**[3] proposed by Popek and Goldberg [14], x86-based ISA violates it, thus cannot be virtualized without additional efforts. Breakthrough is first achieved by VMware Workstation (a VMware type-2 hypervisor), which compensates the ISA drawbacks by a complex software functionality, the *dynamic binary translation* (DBT) [35]. By scanning and fixing the afore-mentioned three types of nocuous instructions at run-time, the

[2] code expansion is measured by the ratio of instructions in source ISA and target ISA, respectively.

[3] **Theorem 1** For any conventional third-generation computer, an effective VMM may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

DBT lets the majority of source (guest) instructions directly run on the physical processor. Xen [36], a popular type-1 hypervisor, exploits paravirtualization [37] to overcome the drawbacks of x86-based ISA. The guest OS needs a slight modification to replace the nocuous instructions with the pre-defined secure routines (hypercalls) before execution. The paravirtualization occurs statically, thus yields higher performance and requires to access the source code of the guest OS.

Both DBT and paravirtualization are software-based virtualization approaches to compensate the hardware drawback of the x86-based ISA. A hypervisor gets bloated and complicated after incorporating these functionalities. The modern x86-based processor vendors realized this and finally come up with processors with hardware-based virtualization solutions (hardware-assisted virtualization extensions) [37], such as Intel vt-x and AMD-V for processor virtualization. These extensions are patches to fix the ISA drawbacks of their processors. The extensions differ slightly from vendor to vendor, but are similar in a newly introduced processor mode, the guest mode (root and non-root) and several instructions. In guest mode, the execution of nocuous instruction yields identical results in virtual as in physical machine (classic virtualization). The additional instructions control the operation of virtual machines. Hardware extensions for virtualization permit simpler and more efficient hypervisors and needs no modification to the code of guest OS.

Memory: The memory is a typical device spatially sharable among tasks [38, 39]. It is already virtualized by *virtual memory* in the physical machine. Benefits of *virtual memory* include: 1) The address space used for programming (logical address space) is totally decoupled from the real address space for execution (physical address space), which greatly eased programming; 2) Processes run in isolated memory areas and have no interference with each other; 3) Fine-grain memory block, *pages* can be used as a basic unit for allocation and security-checking, which not only enables memory-swapping for conceptually using larger memory than physically available, but also enhances memory protection [40]. From logical to physical addresses, the mapping is performed through the memory management unit (MMU) by referring to the page tables created by the operating system. With *virtual memory*, a process sees a contiguous and flat memory space exclusively owned by itself. *Virtual memory* has for long been a key function supported by the major microprocessors and operating systems.

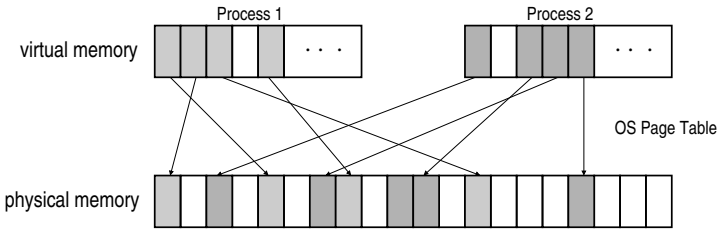


Figure 2.2 Memory address translation in physical machines

On top of the hypervisor, each guest machine is an entity for resource allocation and execution. The main memory of a physical machine is shared by multiple guests. As the operating system does for the processes or tasks, the hypervisor has to provide each guest a view of the contiguous and flat memory space, with zero as the starting address. By this, it mimics the physical machine as true as possible for the unmodified guest OS and applications. The hypervisor introduces a concept called *guest physical address* [41] to fill the gap between guest virtual address and host physical address. Therefore, the guest OS and applications work just as in a physical machine, leaving the mapping from guest physical address to host physical address to the hypervisor or to the host OS. This implies that a guest memory access involves three mapping steps across four address spaces. Figure 2.2 and 2.3 depict the memory address translation in physical and virtual machines, respectively. The current hypervisors adopt mainly two approaches to accomplish this task, which are known as *shadow paging* and *nested paging*[4] [41].

[4]is used for brevity to denote the AMD NPT (nested page table) and Intel EPT (extended page table) solutions.

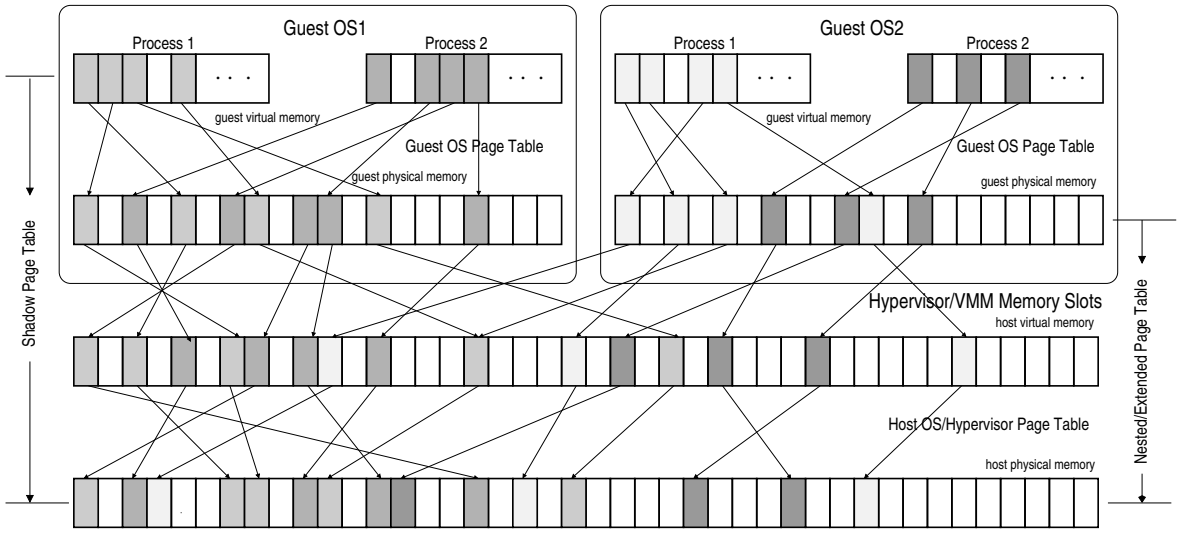


Figure 2.3 Memory address translation in virtual machines

As a software-based solution, *shadow paging* uses shadow page tables (SPT) for mapping the guest virtual address to host physical address. The SPTs are a set of page tables similar to the guest page tables controlled by hypervisor. A key hardware feature *shadow paging* depends on is the *privilege-level protection* mechanism [42] enforced by the processor's MMU. The hypervisor sets the *write-protection* bit in the corresponding control register (`cr0` in x86 / x86-64) [44, 47] for the host physical memory mapped to the guest. If pages containing the guest page table are attempted to be modified (written), a general exception [44, 45] is triggered by the processor, which in turn causes the guest to suspend its execution and give control to the hypervisor or host OS kernel (known as *vmexit*). By this, the hypervisor or host OS kernel can modify the affected page table entry on behalf of the guest OS.

Nested paging, also known as TDP (two-dimensional paging) [41], belongs to hardware-assisted virtualization techniques. Considering that the major overhead under *shadow paging* is incurred by *vmexit* due to the maintenance of SPT, processor vendors shipped facilities for guest page mapping in newer generations of processor. The AMD NPT (nested page table) [46] and Intel EPT (extended page table) [47] are two implementations of *nested paging*. The main purpose is to reduce the hypervisor's intervention to the guest execution by maintaining the page tables for mapping from guest physical to host physical addresses. This is an extension to the mapping from guest virtual to guest physical addresses by using the guest page tables. The mapping from the guest physical to host physical address occurs in each mapping from guest virtual to host physical address, as if they were interleaved or nested as the second dimension of paging.

The key functions performed by the hardware facilities for *nested paging* include 1) enable/disable the *nested paging* mode in the processor; 2) set/clear the corresponding bits in the bitmap for triggering exceptions when `cr3` is loaded, stored or TLB entries are invalidated; 3) trigger the exception when page faults occur in nested page tables. With these functions, the hypervisor's intervention for handling the guest paging faults can be significantly by-passed.

Both *shadow paging* and *nested paging* are indirection layers between guest virtual address and host physical address, with the intention to present the guest an illusion of the physical machine. By that the guest OS can run without any modification. However, if it is possible or necessary to modify the guest OS (probably for higher performance) in the way that host physical address can be directly used by the guest for memory access, such indirection layers are not needed any more. This is what the Xen hypervisor has practiced with its para-virtualization of memory (also known as direct paging). By maintaining the so-called P2M table and M2P table [48, 49, 50] by itself, the Xen para-virtualized MMU can write the guest page table entries with the corresponding host physical addresses [51].

I/O: I/O is the most complex part to virtualize due to various types and natures of I/O devices. Generally, I/O devices can be divided into four catagories by the nature of sharing, namely, the dedicated devices, the spatially partitioned devices, the temporally multiplexed devices, and the spooled devices [14]. A dedicated device can be assigned to and owned by a guest OS in a long period of time without being virtualized. However, since the guest runs at a lower privilege than the hypervisor, interrupts and acknowledgments to the interrupts still need to be routed via the hypervisor. After source and destination are checked and validated, the signals are queued up for injection into the targeted guest in an appropriate moment. Display, keyboard and mouse, which are important for desktop virtualization belong to this catagory.

The spatially partitioned device, such as a hard disk can be virtualized similarly as a memory. The disk may be partitioned and assigned to the guests. Each partition serves as a physical hard disk for the attached guest. A key aspect is the remapping of an I/O request [52] from the guest OS. A request, for example, for writing to a particular region of the virtual disk (identified by cylinder, track and section) must be redirected by the hypervisor to the real region of the physical disk. The result of the operation and the status of the virtual disk are maintained by the hypervisor and reflected in the guests.

A typical temporally multiplexed I/O device is the network interface card (NIC). In physical machines, the NIC queues up data and requests from multiple processes, sends them as instructed (send), saves incoming data from outside and forwards them to the targeted processes (receive). By its nature, the NIC is similar to a processor which can be dedicated to a task in a fine grain of time. In virtual machines, a single NIC can serve multiple guests if the hypervisor presents each guest a virtual duplicate of the NIC. A guest's request to its virtual NIC is intercepted and converted by the hypervisor by replacing the pseudo port number with the real one. Controlled by the hypervisor, a NIC may queue up data and requests from multiple processes of multiple guests, send them and return the results to the attached guests. The NIC receives and sorts the incoming data, converts and routes them to the corresponding guests and processes [14].

Spooled devices tend to be standalone and are shared by more users. A printer, for example, may perform printing tasks from any interconnected users on either physical or virtual machines. However, the requests from a guest must be intercepted and remapped by the hypervisor.

For virtualizing the server's I/O, the major interest lies on the hard disk and network interface. The virtualization of these devices has experienced emulation, para-virtualization (Virt-I/O), and device pass-through, with steady growing performance. The choice for a scenario tends to be a tradeoff among performance, functionality and flexibility of the solution [53].

2.2 Virtualization in HPC

The excellent feature virtualization exhibited in cloud computing has also attracted the attention of high-performance computing. However, substantial differences exist between the facilities for HPC and cloud computing. Traditionally, HPC is engaged in scientific problems and employs supercomputers which take the form of tightly-coupled processors in a single cabinet, known as MPP (massively parallel processors). Otherwise a cluster is used, which is a group of cooperative high-end, homogeneous computers concerted via efficient interconnection in a close range. Although these are large-scale IT facilities comparable with the data centers in dimension and costs, the problems solved by supercomputer tend to be large, complex, diverse, compute- and resources-intensive [59, 60]. The solution of these problems depends on the cooperation of multiple processing units. The pressure for high performance is not only beared by each participating computing nodes, but also by the network. Guided by the "performance first" principle, the use of virtualization is seen as a harm to the performance, thus not advocated in the main-stream HPC. Very few HPC applications run in virtual execution environment [24, 61].

To understand the intrinsic differences between the HPC and cloud workloads, a number of application domains from typical HPC domains are surveyed (Sources: [62, 63, 64, 65]).

Huge Data Processing

In the past few decades, HPC has grwon unprecedentedly and contributed enormously to the social life. A major driving force is the demand for processing a huge amount of data from a wide range of sources. The data explosion in many areas has finally resulted to a new application area, the HPDA (High Performance Data Analysis) to represent the confluence of Big Data Analysis and HPC. Weather forecasting is one of such areas in which supercomputers play an important role, especially as the weather and climate have an immediate impact on the human society and touch deeply upon the daily life of each individual [66, 67]. In recent years the weather has exhibited a global abnormal change reflected by more occurrences of extreme weathers, such as El Niño, La Niña, globally severe drought and extremely unevenly distributed rainfall. This leads to unexpected natural disasters in all spheres of human's life. Accurate predictions of a tornado's movement, heavy rainfall or severe drought may diminish property loss and save millions of lives across vast areas.

However, the increased accuracy is based on the increased amount of data gathered from land, air, ocean and satellites. The pressure for processing such data is extremely huge. This is where massive parallel computing may contribute its strength.

Furthermore, climate modeling, fraud detection, and risk analysis are also application domains that generate huge amounts of data [62].

Complex Process Simulation

HPC's strength also lies in its capability in solving process simulation and optimization problems faster than previously [68].

Being restricted by disarmament treaty and defense budget, or alarmed by the huge impact on environment, nuclear weapon tests are crucial nowadays. The nuclear physicists resort to supercomputer to simulate the explosion numerically. The key aspect for simulation is to trace the reaction chains occurred in molecule or atom scale during the real nuclear explosion within millisecond, which can be simulated by multiple variables. The more variables are traced, the more accurate the simulation is. This results in the amount of input and output data that is beyond the capability of powerful computers [69].

Nuclear reaction is not the only complex process. In physics and astronomy a typical research is to study the behavior of the system composed of a large number of bodies. Such systems range from particles, atoms, molecules to stars and galaxies. A common aspect is their movement and interaction under the influence of physical forces, such as gravity or electromagnetism. The *N-Body Problem* aims to investigate the dynamics and evolution of such systems. Each body moves to a new position, which in turn exerts influence to the acceleration and velocity of the body itself. Considering the number of moving bodies, the number of variables to be traced and calculated for a single time step quickly goes beyond imagination. Even a three-body system like the earth-moon-sun is complex enough to study by normal calculation, not to mention a system containing more bodies. In reality, such problems are so common that nowadays quite a lot of supercomputers are dedicated to perform tasks for molecular dynamic modelling.

Other areas involving complex processes simulation and optimization may also include chemistry, solid, fluid and material dynamics as well as thermal hydraulics. With the huge computing power, simulation is used to explore many complex systems that were impossible to study in the labs [70].

Exhaustive Searching

For certain theoretical researching, a common problem is to find optimal or approximated solution in a possibility space. Since the overall cost is proportional to the number of solutions, even if the cost for a single search is small, the complexity grows rapidly as the size of the problem increases. The typical usage may include cracking the encrypted ciphers in cryptanalysis [71, 72] and analysis of the genome in biology [73].

Taking the cryptanalysis as an example, the possibility space tends to be formidably huge and denies any cracking effort in normal speeds. The massive parallelism delivered by supercomputer makes it feasible to search through parts of or even through the whole space. A supercomputer equipped with ten thousand processors, for instance, can try ten million passwords per second [74]. This significantly reduced the time and increases the chance to crack a password. In general, though, this kind of searching is still too expensive and only feasible for limited problem scales. A better practice is to reduce the searching space to a manageable size by using some heuristics.

In summary, all the enumerated applications are large in problem size, thus may benefit from the processor speed, homogeneity, fast interconnections and parallel file systems. In comparison with the wide deployment of virtual machines in cloud data centers, the interest for virtualization in HPC is mainly focused on providing HPC services by using resources and constructing supercomputers in a cloud environment.

A famous example is the FutureGrid [75] project at Indiana University launched in 2009 till 2013, which is a geographically distributed, high-performance test-bed for developing innovative approaches for grid, cloud, and parallel computing. FutureGrid is composed of a group of HPC resources connected by a high-speed network. It allows to be accessed as a traditional HPC cluster, a computational grid, or a highly configurable cloud infrastructure, respectively.

In 2011, Amazon, one of the market leaders in e-commerce and cloud computing, announced the world's fastest "non-existent" supercomputer [76], which is spun up atop of the Amazon cloud, EC2 (Elastic Compute Cloud), and ranks 41 in the Top500 list at that time.

By December 1, 2014, a computational service platform called Virtual Supercomputer has been released by the European developer of HPC system software, Massive Solution. The Virtual Supercomputer service platform adopts KVM as the hypervisor and InfiniBand exclusively as the protocol for interconnect, management and storage networks [77].

Although many supercomputer vendors and experts claimed that cloud services can't match the requirements of HPC workloads compared with the dedicated HPC clusters, these systems demonstrated the potential to construct virtual HPC clusters in the existing cloud environment. Considering the large investment and long term for building up a dedicated HPC cluster, a virtual cluster in the cloud is an alternative affordable by ordinary HPC users. It also showed the possibility to narrow the performance gap between Amazon cloud and dedicated supercomputers.

Besides the "HPC in cloud", another approach for HPC to exploit the virtualization technology is to launch virtual machine guests on the computing nodes of a traditional supercomputer. For this purpose, the hypervisors and operating systems for HTC tend to incur too much performance loss and are too exhaustive in resource consumption. Thin, lightweight hypervisors and operating systems ensure simpler implementations and less performance loss at the cost of functionality. A research on this was launched by the Sandia National Laboratories in collaboration with New Mexico University and Northwestern University [78]. The two universities had a lightweight hypervisor, Palacios, specially developed for HPC, and the Sandia had a lightweight operating system kernel, Kitten. They were combined to create a virtual execution environment on a supercomputer. Experiments showed that for the benchmark programs, the performance loss was less than 5% on a scale of 50 nodes, and remains at the same level for thousands of processors in a dedicated supercomputer. The project is not only helpful in improving the resource utilization and reducing power consumption, but also in creating virtual execution environments for the development of system software stacks used by the upcoming exascale computing.

According to the authors, the Palacios-Kitten combination has achieved the goals desired by researchers on adopting virtualization in HPC. This is so far the first attempt to use specially crafted lightweight hypervisor and OS kernel for deploying the VM-contained workloads in a large scale on a real supercomputer platform. The benchmark demonstrates the feasibility to adopt virtualization in HPC. The value of their work lies not only on the idea to use lightweight hypervisor and OS kernel for HPC, but also on the idea of *symbiotic virtualization*[5] [79].

[5]Symbiotic virtualization is an approach to system virtualization in which a guest OS targets the native hardware

The limitations are: 1) The result is yielded for the HPCCG benchmark. Each computing node runs a single VM instance. The diverse HPC workloads and environment configurations are not fully investigated; 2) Palacios and Kitten lack functional maturity [80]. They are currently mainly for research.

Actually, the first limitation is mainly due to the difficulty in HPC performance study. Different user applications have to be tested on different hardware with different hypervisors to understand the limitations of existing virtualization technologies [80]. The authors of [80] pointed out that while KVM provides more stable and predictable results, Palacios is better on fine-grained tests, but tends to show abnormal performance degradation on other tests.

While virtualization poses the same challenges for cloud and HPC, the major barrier is the performance loss for HPC due to virtualization. The benchmark results in [81, 82] show that the performance of HPC workloads tend to be more sensitive to virtual than to physical execution environments. Considering that the HPC workload is diverse, the performance is more workload-dependent in virtual execution environments. This makes it hard to evaluate the performance of a given virtual execution environment[6]. Since the benchmark result reflects the performance of the whole system, it is also difficult to evaluate the performance of the subsystems.

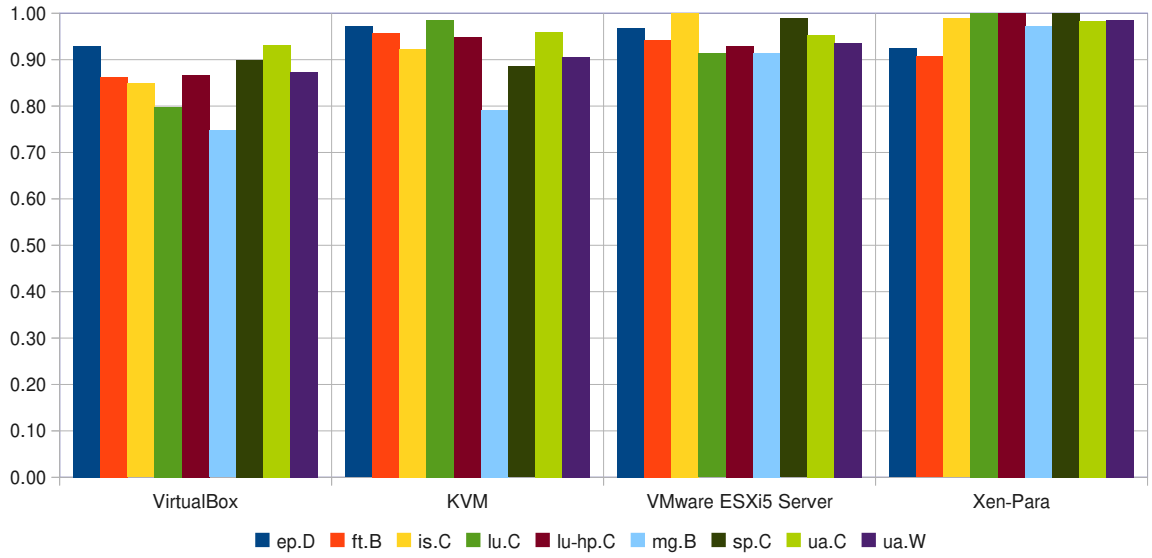


Figure 2.4 Normalized performance of NAS in typical virtual environments

Figure 2.4 reveals the performance of an HPC workload with a few commonly used hypervisors, namely VirtualBox, QEMU-KVM, VMware ESXi5 Server, and Xen (Paravirtualization).

The benchmark applications are from the NAS Parallel Benchmarks (NPB) [83]. What the figure illustrates is not only the performance loss of HPC workloads generally for those virtual environments, but also the nature that the performance loss is workload-dependent. Furthermore, hypervisors yield different performances. Among them, Xen and VMware ESXi Server are more efficient than QEMU-KVM and VirtualBox. This is most likely due to the particular optimization techniques adopted by their own developers, such as the hypercalls for guest-hypervisor interaction for Xen-Para, and some unknown modification to the VMware ESXi Server as its commercial competitiveness. In contrast, QEMU-KVM is still a young open-source hypervisor. VirtualBox is more user-friendly but not server-oriented.

interface as in full system virtualization, but also optionally exposes a software interface that can be used by a hypervisor, if present, to increase performance and functionality.

[6]Performance of a virtual execution environment is evaluated by the performance of workload.

2.3 Performance for Memory Virtualization

So far, the performance for processor virtualization is well studied [84]. Another potential source of performance loss is memory virtualization, or the guest address translation. As the authors of [85] believed, one of the main bottlenecks of virtualization systems is "the reduced performance of the memory system", although it may be critical only for a narrow class of problems.

Fundamentally, performance loss of the memory system is suffered not only in virtual machine, but also in the physical machine. The increasingly widening speed gap between processor and memory is the main reason. In virtualization, such performance loss is further complicated by the presence of an indirection layer – the hypervisor.

Before AMD and Intel's MMU caches were introduced, many efforts were focused on reducing the TLB misses and speeding up the software TLB miss handling. Two entirely software-based solutions were proposed to decrease TLB penalties by [86]. The first solution is *prefetching TLB entries on the IPC path*[7]. This decreases kernel TLB miss penalties by about 50%. The second one uses a large software cache to provide fast path access to entries upon a TLB miss. The hit rates achieved range from 90% to nearly 100%. This software cache can be used by the software page fault handler and manages entries in physical memory to avoid cascading TLB misses that are incurred by reading page table entries in virtual memory space.

PTPC (page table pointer cache) [87] is proposed to replace the TLBs that cache the *pointers to pages of page table entries* rather than to *page table entries*. PTPC traps and handles most TLB misses in hardware usually with only a single memory access. Since the PTPC misses are filled by software, PTPC is easy to implement in hardware. Furthermore, as a PTPC refers to an entire *page of page table entries* rather than *page table entries*, a small PTPC covers large address space and achieve high TLB hit rates. The combined use of TLB and a small PTPC can perform significantly better than TLB alone. Meanwhile, the flexibility by using small and fixed size pages and standard TLBs can be preserved.

Specific to the chip multi-core processor (CMP), it [88, 89] shows that TLB misses have huge impact on the system performance, and 30% to 95% of the total TLB misses are redundant and predictable misses for multi-thread parallel applications. The results are valuable for novel TLB designs in favor of inter-core cooperation through either hierarchically shared TLBs or inter-core TLB prediction mechanisms.

Shared last-level (SLL) TLBs [90] are proposed analogous to shared last-level caches in CMP. It maximizes the caching efficiency of TLBs by sharing limited physical resources among cores. The benchmark results show that SLL TLBs not only eliminates a considerable number of the system-wide TLB misses for parallel workloads, but also does similar or even better for sequential workloads. Due to these strengths, SLL TLBs are promising for CMPs.

Synergistic TLB [91] is proposed as a mechanism to improve system performance by organizing TLBs in a manner that suits page access characteristics prevalent in CMPs. The design is based on the considerations of organizing TLBs with capacity sharing, translation replication and migration. It exhibits the potential to eliminate a large portion of the TLB misses and speedup both the multiprogrammed SPEC 2006 and multi-threaded PARSEC workloads.

In the TLB entry prefetching aspect, a novel pre-fetch mechanism *distance prefetching* [92] is proposed to capture patterns in the reference behavior in a small address space. This is based on a detailed comparison between different prefetching mechanisms, arbitrary stride prefetching, and markov prefetching that were previously proposed for caches and TLB entries. The *distance prefetching* is an architecturally independent pre-fetching technique based on access patterns and inter-core cooperations, and focuses on reducing the cost of handling a TLB miss.

Besides the efforts to diminish the occurrence of TLB misses and to reduce the cost of handling TLB misses, works are also engaged in designing efficient forms of page table. *Hashed page table* [93], *cluster page table* [94], *guarded page table* [95] etc. are all variants of the page tables used by

[7] IPC - instruction prefetching cache

default. The intention is to take advantage of a specific feature, or nature in memory allocation, caching or an extreme case of the virtual address space.

As more hardware-assisted memory virtualization techniques become available, virtualization is increasingly popular. Memory virtualization has gradually been a focus. AMD's PWC (page walk cache) caches most of the upper level page table entries in page tables of both the guests and hypervisor. Based on the PWC, a guest-physical-to-host-physical translation cache – Nested TLB (NTLB) is proposed [96]. A hit in the NTLB allows the two-dimensional page table walk to reuse the previously entries thus skip the corresponding page table walk on the hypervisor's page tables. If each access to the guest's page table hits in the NTLB, the number of accesses to the PWC and the memory hierarchy are the same as in the native execution.

The design space for MMU caches is explored [97] to speed up the virtual-to-physical address translation for processors with radix tree structured page tables. It shows that the most effective MMU caches within this space are *translation caches* that store partial translations for skipping a few page walk steps.

Inspired by the *nested paging*, two paging schemes are proposed and evaluated to reduce the overhead of *nested paging* for virtualized systems [98]. The first scheme is the *nested paging* with flat nested page tables. It reduces the number of memory references at the cost of a slight change on the hardware. The second scheme is the speculative inverted shadow paging, backed by non-speculative flat nested page tables. It enables a direct translation with a single memory reference for common cases, and eliminates the cost for page table synchronization. Both schemes proved effectiveness to improve the performance for the state-of-the-art page table walk hardware.

The cons and pros for both the two standard approaches, *shadow paging* and *nested paging* are found in performance study. As each of them has its strengths and weaknesses when dealing with workloads, neither can be replaced by the other. A better approach is to exploit their strengths according to the nature of the running workload. This is the origin of the basic idea – *dynamic switching of paging methods*.

Basically, the idea of *dynamic switching of paging methods* is not new. The first appearance seems to be in [99]. The authors argued that *nested paging* is better for database-oriented applications since it eliminates a large number of *vmexit* events. *Shadow paging* is better for compute-intensive workloads that incur limited activities in the guest kernel space. The authors pointed out that the type of workload could be automatically detected by QEMU-KVM at runtime. Based on Xen and Palacios, two implementations are worked out [100, 101], respectively. Although different hypervisors, policies and mechanisms are applied in their implementations, each of them have shown that the *dynamic switching of paging methods* is able to yield on-par or better-than-the-best performance yielded by *shadow paging* and *nested paging*.

A recent advance for *dynamic switching of paging methods* contributed by [103] views *shadow paging* and *nested paging* as two extreme cases for guest memory virtualization. Based on the observation that the updating frequency tends to vary considerably at different levels of page tables, the authors proposed *agile paging*. It switches between the two paging methods dynamically for a single address translation. Benchmark results showed that *agile paging* outperforms the best of *shadow paging* and *nested paging*. *Agile paging* requires a slight change not only to the hypervisor, but also to the processor.

The trends for improving the memory virtualization can be summarized as the following:

1. Switching the paging method dynamically based on the performance data sampled from PMCs without modifying the hardware, represented by [100, 101, 102];
2. Switching the paging method dynamically during a single translation with a slight modification to both the hypervisor and the hardware, represented by *agile paging* [103];
3. With modifications to hardware, the standard two-level TDP paging scheme is replaced by other schemes, represented by [104] and [105].

In the third case, the first proposes a hashed page table for the second-dimension page tables. The second proposes a shift from paging to segmentation for the guest memory virtualization, which leads to near-zero translation and better-than-native performance. This is based on the feasibility to map large memory chunks in a process's virtual address space by using segmentation and only the remained small regions by using paging. While both approaches are reasonable, they lack true hardware support, thus are not practicable with the current available hardware.

The above trends reveal that: 1) With modification to the hardware, both dynamic switching and two-dimensional paging can be done in a more radical way, leading to better-than-the-best of the current performance; 2) Such hardware modifications are not supported by any of the current processors; 3) When dynamic switching is done in a conservative way (implies slow pace for switching), the performance is generally comparable with the best of the current performance, but less likely to outperform the best of the current performance; 4) For any better-than-the-best of the current performance, modifications to the hardware are necessary and almost inevitable.

2.4 Summary

This chapter reviews the efforts directly or indirectly related with the topic of this thesis. As the central research object, system virtualization technologies are examined in the evolution of the techniques to virtualize the major components of the computer system. Then, the focus is moved to the application of the system virtualization in HPC and the major problems illustrated by a few well-known projects. Furthermore, the efforts for improving the performance of the memory address translation system in both the physical and the virtual execution environments are enumerated. All these form the foundation on which the efforts in this research are undertaken. With this chapter, Questions 1 and 2 in Section 1.4 are answered.

Chapter 3 A Study of the Performance Loss for Memory Virtualization

Contents

This chapter investigates the performance loss due to the virtualization of the guest memory address translation. The starting point is to examine the deployment pattern of the typical HPC workloads in virtual execution environments. Limitations of the current memory virtualization solutions are revealed by a group of the typical benchmark applications on the selected platforms.

Section 3.1 introduces the two basic patterns for deploying HPC workloads, namely the intra-node and the inter-node, and their respective impact on the overall system performance. The suitability of the selected benchmark applications for the intra-pattern are discussed in Section 3.2. Section 3.3 specifies the benchmark platforms used for investigating the performance loss. These workloads and platforms are also used for the functional testing and performance evaluation of the implementations proposed by this thesis. In Section 3.4, the performance losses for the intra-node pattern are analyzed. Section 3.5 summarizes this chapter.

3.1 HPC Workload Deployment Patterns

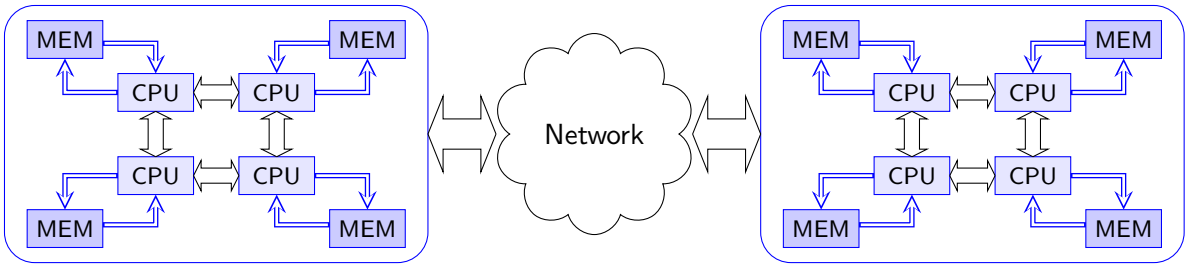
To harness the massive parallelism of a supercomputer, the HPC workload is normally deployed across multiple computing nodes, at the cost of performance loss incurred by the communication over network across nodes. Although the execution in a single node is free from such overhead, workloads in HPC are rarely small enough to fit into the memory of a single node. Therefore, two basic patterns exist for deploying HPC workloads in a cluster-based supercomputer. This is the case for homogeneous computing nodes[1]. For heterogeneous computing nodes, variants may be derived by combining the two basic patterns with various strategies for resource allocation in each single node.

The intra-node pattern is applied not only to run a whole workload if it fits onto a single node, but also to run a part of a large-scale workload across multiple nodes. The processors, memory, storage I/O devices, and buses bear great impact on the performance in this case.

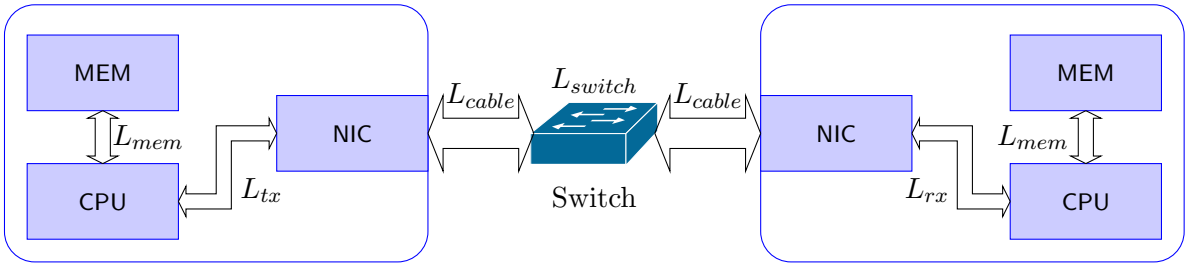
[1]homogeneous means a computing node built up with same type of processors, opposite to heterogeneous, which means computing node comprising different types of processors, such as CPU+GPGPU, CPU+FPGA hybrids.

The inter-node pattern applies to large-scale workloads that run across nodes. For this pattern, the bandwidth and latency of the network are the major influencing factors.

HPC workloads may leverage different levels of parallelism by applying a mix of programming models supported by the programming language extensions, tools and libraries. Pthread (POSIX thread API), OpenMP, CUDA, OpenCL, Intel MKL, AMD ACML, MPI and PVM are such examples. Among them, MPI and PVM are mainly inter-node oriented, thus have the potential to exploit the computing power of multiple nodes. The others are exclusively intra-node oriented, thus are mainly used to exploit the parallelism of multi-cores and multi-threads in a single node.



(a) Overview of the HPC system architecture



(b) The constitution of performance loss

Figure 3.1 Typical HPC system architecture and the constitution of performance loss [106]

Figure 3.1 depicts the intra-node and inter-node patterns as well as the make-up of the overall performance loss in a typical HPC system. The computing node may have chip-multiprocessors (CMP), symmetric multiprocessors (SMP), NUMA, or heterogeneous architecture. Figure 3.1(a) depicts the homogeneous architecture, including CMP and SMP. The intra-node communication occurs via the processor-memory buses and I/O buses. Processor-memory buses are short and generally high-speed. I/O buses connect various I/O devices, thus are lengthy and slower than processor-memory buses. Currently, the mainstream I/O bus technologies mainly include PCI-Express (PCIe), Intel QuickPath Interconnect (QPI), and AMD HyperTransport (HT).

In contrast, the inter-node communication occurs over longer-distance physical network path consisting of switches, routers and cables, thus are generally more expensive.

The bandwidth and latency for the two patterns bear strong impact on the HPC workload's performance. According to a few studies ([106, 107]), the bandwidth and latency for inter-node and intra-node patterns are in the same order of magnitude. The intra-node latency grows when the processor gets busy. The inter-node cost depends on the number of switch hops [106], which can be reduced by *network offloading* [108]. For well-scalable workloads, less time is spent for process synchronization, therefore the performance loss over inter-node is almost negligible. Otherwise, the inter-node latency may increase the overall latency by 10% at worst [106].

Figure 3.1(b) shows the make-up of the overall performance loss. L_{mem}, L_{tx}, L_{rx}, L_{cable}, and L_{switch} stand for the performance loss incurred by memory access (including memory read, write and paging operations), outgoing, incoming data transfer (send and receive), cable and switch, respectively.

Performance analysis of the virtualized HPC workloads focuses on both the inter-node and intra-node patterns, represented by message-passing and memory-sharing parallel approaches.

HPC covers a wide range of professional application areas, thus the HPC workload is extremely diverse in resource demanding and run-time behavior. So far, quite few HPC-oriented benchmark suites are both diverse in workload stress and characteristic. To study the performance for the two deployment patterns, different types of HPC workload are needed.

3.2 Benchmark Selection for Intra-node Pattern

For the intra-node pattern, any workload that stresses the physical hardware is an ideal choice. A common practice is to adopt the multithreaded applications, with which processors, memory and storage may be put under different stress levels of stress by tuning parameters, such as the number of processes, threads, and the size of input data. HPC applications using Pthread, Java thread, OpenMP, or MPI can be applied as tools for benchmarking.

For the inter-node pattern, as the workload needs to exploit the node-level parallelism (NLP) and stress the network between the participating nodes, MPI-based applications are used. The choice of benchmark applications is discussed below in detail.

Table 3.1 lists a number of benchmark suites typically used for performance research. They have different characters such as the type of parallelism, type of workloads, diversity, purpose, and the orientation to HPC. Among them, only SPEC CPU2006, SPEC OMP2001 and SPLASH-2 are diverse and focused on HPC. However, the first two are commercial software, which are not freely available for research. Considering the diversity and orientation to HPC, the choices for HPC benchmark are rather small. SPLASH-2, BioPerf, and PARSEC-3.0 are such candidates.

Table 3.1: A set of typical benchmark suites for performance research

Benchmark Suite Name	Number of Application	Type of Parallelism	Emerging Workload	Diverse	HPC focused	Purpose
SPEC CPU2006	29	C, C++, Fortran	No	Yes	Yes	Commercial
SPEC OMP2001	11	OpenMP	No	Yes	Yes	Commercial
SPLASH-2	14	OpenMP [109]	No	Yes	Yes	Research
ALPBench	5	Pthread, ILP [110]	Yes	No	No	Research
BioBench	7	ILP [111]	No	No	Yes	Research
BioParallel	5		No	No	Yes	Research
BioPerf	10		No	Yes	Yes	Research
MediaBench II	12	ILP [112]	No	No	No	Research
NU-Minebench 2.0	15		No	No	No	Research
PhysicsBench	8		Yes	No	No	Research
PARSEC 2.1	13	Pthread	Yes	Yes	No	Research
PARSEC 3.0	25 + 2	Pthread	Yes	Yes	Yes	Research
NAS PB	11	MPI, OpenMP	No	No	Yes	Research

Except the entries for BioPerf, PARSER 3.0, and NAS PB, other data are from [113]
PARSEC 3.0 merged SPLASH-2 with PAESEC-2.1, thus is HPC-focused at least for its SPLASH-2 part.
Emerging workload means those workloads which are likely to become important in the near future yet not much commonly applied nowadays.

The choice of PARSEC benchmark suite (both v2.1 and v3.0) has a few reasons. First, despite the fact that the applications included in PARSEC-2.1 were not chosen with a special orientation to HPC, they still represent a form of utilization for a HPC cluster (or Supercomputer) [114]. Second, most of the HPC applications (no matter if they are packed into a benchmark suite) are based on MPI and execute mainly on distributed-memory systems. To benchmark a single node, it does not matter too much whether the benchmark programs are specially oriented to HPC. Given the system composed of the hardware and hypervisor, the performance bottlenecks exist independently of any benchmark programs. Non-HPC workload may also be capable of exposing the performance drawbacks of the system for HPC workloads. Last but not least, the PARSEC benchmark suite includes emerging workloads in recognition, mining and synthesis (RMS) and system applications which mimic the large-scale multi-threaded commercial programs.

RMS deals with the processing of huge data by specialized algorithms in several key areas such as artificial intelligence (AI) and virtual reality (VR) [115]. The algorithms involves clustering/ classification, Bayesian network, Markov model, decision trees, neural networks, probabilistic networks, linear/non-linear stochastic, and time series models [116, 117]. Therefore, workloads of this type tend to exhibit not only intensive computation, but also intensive memory and I/O transactions. These exposes more performance drawbacks by loading the computing system with different stress.

Revealed by characterization testing, this benchmark suite covers a wide spectrum of working sets, locality, data sharing, synchronization as well as off-chip traffic [118]. In a word, PARSEC benchmark suite is chosen for its workload diversity and ease of use rather than its orientation to HPC. In the latest PARSEC-3.0 release, the SPLASH-2 benchmark suite gets integrated into the PARSEC package. By this approach, while the workload is increasingly diverse, two originally separate benchmark suites are brought under the same roof for management and execution.

There had been a number of research results [118, 119] on the two collections of programs, dealing with the fundamental properties of each suite, and a quantitative comparison between them [120] in various aspects. According to these, programs in the two suites exhibited fundamentally different characteristics, largely due to the different scopes at the time of their creation.

SPLASH was composed before chip-multiprocessors came as the mainstream, with the focus on HPC and graphic processing where performance has a higher priority than other aspects. The composition of the whole suite reflects the major focus in that condition. In contrast, PARSEC emerges in an era the CMP has been the de-facto standard not only for the HPC and graphic processing, but also broadly for general-purpose computing areas, represented by the emerging workloads. Consequently, the suite is composed with more applications from these areas. While maintaining the workload diversity, new trends and technical innovations are also reflected.

Eventually, the significant difference between them and a need to update the SPLASH-2 have prompted an overhaul of the SPLASH-2 and its merging into the PARSEC benchmark suite. The combined suite is benefited from both the diversity reflected by the two suites and the new trends represented by the emerging workloads in the former versions of PARSEC releases.

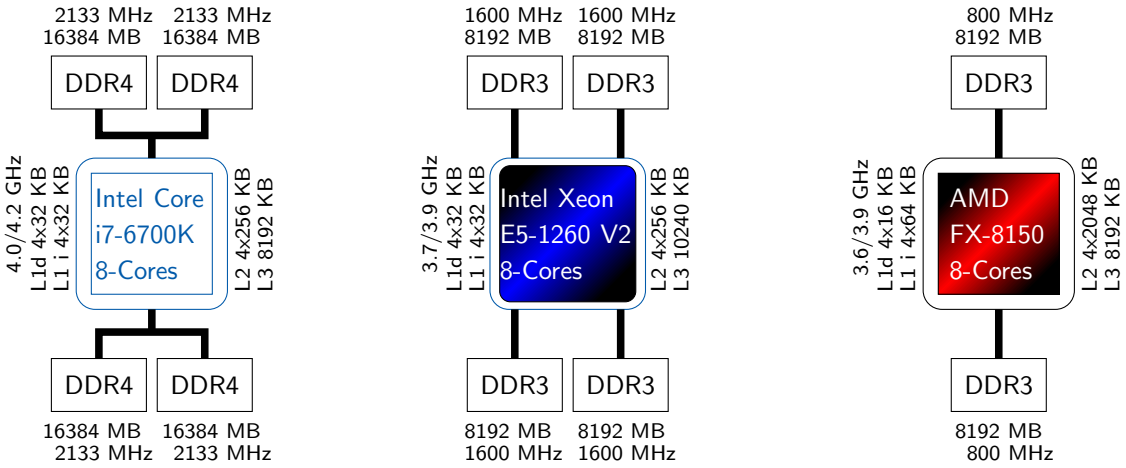
Depending on the purpose, a benchmark suite can be used to measure the impacts of different factors on the performance for a computing system, such as cache size, cache miss, work set size, spatial locality, temporal locality, load balance, and off-chip traffic. In-depth analysis has been presented by [118, 119] when characterizing SPLASH-2 and PARSEC benchmark suites.

For the merits discussed above, PARSEC-3.0 is used as the benchmark to study the performance of HPC workloads for the intra-node pattern.

3.3 Benchmark Platform

Due to the slight difference in the implementations of the instruction set [121], the two major variants of the x86 architecture, Intel and AMD processors, are known to be not completely compatible for a few instructions and operations. Such incompatibility extends to the function of virtualization. Both the two vendors have their extensions for virtualization. A consequence as mentioned in Section 1.1 is that the VM-migration between Intel and AMD platforms can not be done quite freely. When developing an operating system or a hypervisor for x86 ISA, two separate implementations had to be worked out to deal with the minor differences in instructions[2].

It is noticeable that for the same benchmark workload, performance in the environments with nearly identical software configuration may vary considerably from one platform to another. By adopting processors of different families, micro-architectures and vendors, the impact due to different processor is highlighted, and the benchmark results gains a more general sense. For this reason, two types of Intel processors and an AMD processor are used. Figure 3.2 illustrates the main configuration of these platforms.



(a) Platform 1 (b) Platform 2 (c) Platform 3

Figure 3.2 Testing platforms

3.3.1 Intel Platform

Intel Core™ i7-6700K (Skylake)

This platform is equipped with an Intel Core™ i7-6700K processor on Intel Z170-Pro mainboard. The Core i7-6700K chip is a high-end processor for desktop or server, and belongs to the initial batch of Skylake release. Due to the hyper-threading feature in the Core i7 family, the i7-6700 processor is able to present two virtual processor cores for each physical processor core, therefore makes a total of 8 cores for the system. It has a default clock speed of 4.0 GHz and a turbo speed of 4.2 GHz. Another key aspect is the memory hierarchy it utilizes. As Figure 3.2(a) illustrated, the i7-6700 supports up to two memory channels, hence is capable to accommodate two pairs of memory modules. In terms of computer architecture, each pair of the memory modules occupies a *matching bank* of the memory channel. With a capacity of 16 GB PCB (printed circuit board) stick installed in each slot on the main board, the processor is capable of accessing to as large as 64 GB DIMM (dual-in-line memory module) DDR4 in total.

The normal caches include: 4x32 KB 8-way set associative level 1 instruction caches, 4x32 KB 8-way set associative level 1 data caches, 4x256 KB 4-way set associative level 2 caches, and 8 MB 16-way set associative shared level 3 caches; The TLB caches include: 4 entries of 4-way

[2]Actually, the difference is incurred by concrete circuits, rather than by the instruction set architecture.

set associative level 1 data TLB for 1-GB pages, 64 entries of 4-way set associative level 1 data TLB for 4-KB pages, 64 entries of 8-way set associative level 1 instruction TLB for 4-KB pages, 64-byte line size of 4-way set associative level 2 TLB for 1-MB pages, 1536 entries of 6-way associative level 2 shared TLB for 4-KB / 2-MB pages, and 16 entries of 4-way associative level 2 TLB for 1-GB pages [122].

For virtualizaton, i7-6700K has the following features: VT-x / Virtualization technology, VT-x EPT / Virtualization for extended page tables, and VT-d / Virtualization for directed I/O.

Intel Xeon E5-1620 (Ivy Bridge-E)

This platform has an Intel Xeon E5-1620 processor on Supermicro X9SRA/X9SRA-3 mainboard. The processor belongs to one of the Intel Xeon series product line based on the Ivy Bridge micro-architecture, and targeted at the server, workstation segment. The hyper-threading feature has enabled the quad-cores to present a doubled number of virtual cores. For a single physical core, the default and turbo clock speeds are 3.70 GHz and 3.90 GHz, respectively. Since this processor has four memory channels, four pairs of memory modules are supported. However, only half of the total slots are installed. Each PCB stick (DIMM DDR3) has a capacity of 8 GB. The total volume amounts to 32 GB.

The normal caches include: 4x32 KB 8-way set associative level 1 instruction caches, 4x32 KB 8-way set associative level 1 data caches, 4x256 KB 8-way set associative level 2 caches, and 10 MB 20-way set associative shared level 3 caches [123]. TLB caches include: 32 entries of 4-way set associative level 0 TLB for 2-MB / 4-MB pages, 64 entries of 4-way set associative level 1 data TLB for 4-KB pages, 64 entries of 4-way set associative level instruction TLB for 4-KB pages, 64-byte line size of 4-way set associative level 2 TLB for 1-MB pages, and 512 entries of 4-way set associative shared level 2 TLB for 4-KB pages [124].

The extension for virtualization support includes the following features: VT-x / Virtualization technology, VT-x EPT / Virtualization for extended page tables, and VT-d / Virtualization for directed I/O.

3.3.2 AMD Platform

AMD FX^{tm}-8150 (Bulldozer)

The AMD platform is equipped with an FX^{tm}-8150 processor on the MSI 970A-G46 Sockel AM3+ mainboard. The FX-8150 belongs to the AMD FX-series family. which was designed as a rival for the Intel Core-series counterparts (mainly the Sandy Bridge and Ivy Bridge micro-architectures) in the high-end desktop PC segment. The micro-architecture was Bulldozer upon initial release and the successor - Piledriver later. Unlike many Intel 8-core processors, which are presented based on the quad cores by using hyper-threading, one of the amazing aspects of FX-8150 is that it is a true 8 physical core processor, without taking advantage of the hyper-threading.

The FX-8150 processor has a single memory controller and double memory channels, and each channels accommodates a pair of DIMMs. On this platform, half the number of memory slots are installed with DIMMs, which leads to a 16 GB memory addressing space for the processor. On the cache side, there are 4x64 KB 2-way set associative shared level 1 instruction cache, 8x16 KB 4-way set associative level 1 data cache, 4x2 MB 16-way set associative shared exclusive level 2 cache, and 8 MB 64-way set associative shared level 3 cache [125]. TLB caches are organized as two levels. The level 1 iTLB contains 72 entries shared by a variety of page-sizes - 2x24 entries for 4-KB pages, and 24 for 2MB or 1GB pages. The level 2 iTLB contains 512 entries of 4-way associative cache-lines for 4KB-pages [126].

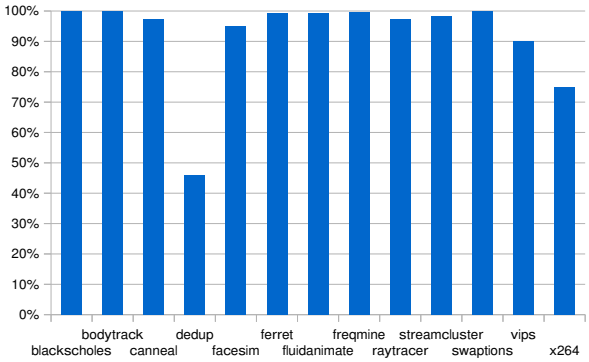
The extension for virtualization is AMD-V, featured with SVM (secure virtual machine) technology, RVI (rapid virtualization indexing, formerly called nested page table) for guest memory virtualization, and AMD-Vi (formerly called IOMMU) for I/O virtualization.

3.3.3 System Software

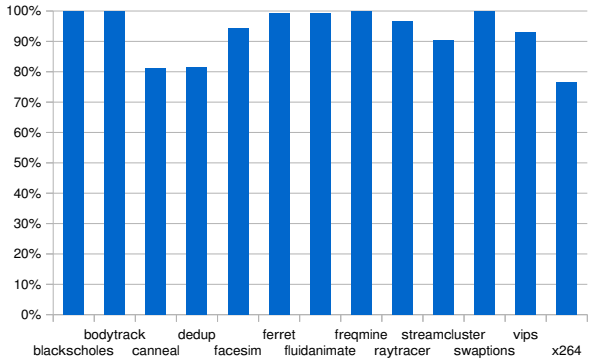
All the above platforms have identical system software, with Ubuntu 14.04.4 LTS (Trusty Tahr) as the OS, and the customized (due to the modification to KVM) Linux 4.4.13 as the OS kernel. The C-compiler is GCC version 4.8.4. To make the guest nearly the same as the host, the same operating system and compiler are installed on the guest and host.

3.4 Performance for Memory Paging

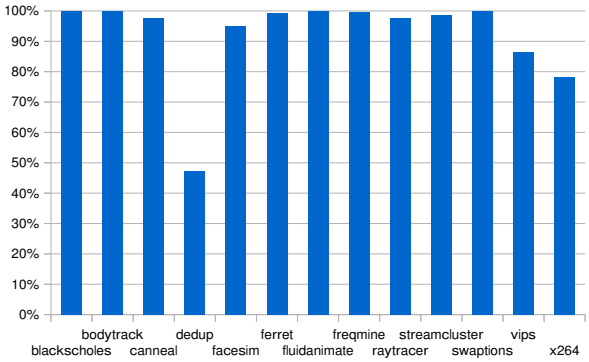
As mentioned in Section 2.1, the memory virtualization or paging in a VM guest is implemented mainly in two approaches, the *shadow paging* and the *nested paging*. According to the benchmark in intra-node pattern, memory virtualization poses as a potential bottleneck in a few cases. Figures 3.3 (a) and (b) compare the normalized performances between the *shadow paging* and the *nested paging* with a page size of 4 KB. Figures 3.3 (c) and (d) compare the performances between the *shadow paging* and the *nested paging* with a page size of 2 MB. Both of them have been executed on platform 3 (see Section 3.3).



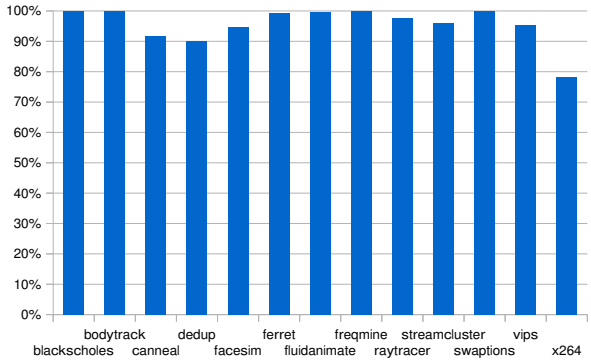
(a) *shadow paging* with page size of 4 KB



(b) *nested paging* with page size of 4 KB



(c) *shadow paging* with page size of 2 MB



(d) *nested paging* with page size of 2 MB

Figure 3.3 The normalized performances of *shadow paging* and *nested paging* (The vertical axis stands for the performance ratio between *shadow paging* and *nested paging*)

The results can be interpreted as the following: 1) For most of the selected benchmark workloads, the *shadow paging* and the *nested paging* suffer less than 10% of the native performance; 2) Both the *shadow paging* and the *nested paging* may suffer large percent of the native performance for certain workloads. But generally, the *shadow paging* suffers more than the *nested paging*; 3) Larger page table (2 MB) benefits the performance for most of the workloads; 4) Both the *shadow paging* and the *nested paging* have their strength and weakness. Neither is guaranteed to serve a workload ideally. Similar case occurs even for larger page tables.

Table 3.2: Performance comparison between the *shadow paging* and the *nested paging*

Workloads	SPT(4KB)	NPT(4KB)	SPT(2MB)	NPT(2MB)
canneal	0.974	0.813	0.976	0.917
dedup	0.461	0.815	0.474	0.900
facesim	0.949	0.942	0.952	0.948
streamcluster	0.984	0.902	0.988	0.961
vips	0.901	0.931	0.863	0.954
x264	0.749	0.767	0.783	0.782

For an easier comparison, Table 3.2 illustrates the most interesting part of the above results. Workloads such as `dedup` and `vips` benefit from adopting the *nested paging*, with the performance gain ranging from 3% to 43%. Conversely, `canneal` and `streamcluster` suffers more in the *nested paging* ranging from 2% to 16%.

These results are obtained by benchmarking only on a single platform and for a single thread. To gain more general insight, more platforms and more threads are adopted. The benchmark suite, PARSEC 3.0 (with 12 more applications) is executed on the above mentioned platforms for 1, 2 and 4 threads, respectively. Figures 3.4a, 3.4b and 3.5a present the performance comparison between the *nested paging* and the native case. In contrast, Figure 3.5b, 3.6a and 3.6b present the performance comparison between the *shadow paging* and the *nested paging*. The following results are identified:

- A small percent of the total workloads suffer large percent of the native performance by using *nested paging*; (e.g. P1: 4%; P2: 16%; P3: 12%;)
- Workloads suffering heavily may vary from one platform to another (See C1 in Table 3.3);
- The performance of a specific application may vary with a different number of threads (See C2 in Table 3.3);

These figures provide the view of direct comparisons of the performances yielded by the *shadow paging* and the *nested paging*. The identified facts are listed below:

- Generally, the *shadow paging* yields on-par performance with the *nested paging*. P1: 88% of the total workloads run nearly at the same speeds in both the *shadow paging* and the *nested paging*, with less than 1% difference; P2: 72% of the total workloads yield above 97% of the *nested paging* performance under *shadow paging*; P3: 84% of the total workloads yield above 97% of the *nested paging* performance under the *shadow paging*);
- The *shadow paging* is significantly slower than *nested paging* for a certain workloads (See C3 in Table 3.3);
- The *shadow paging* outperforms the *nested paging* for a few workloads (See C4 in Table 3.3);
- The *shadow paging*'s superiority over the *nested paging* occurs more commonly with multi-threading (See C5 in Table 3.3);
- The superiority of a paging method over the other is platform-dependent. For example, `fft` exhibits opposite behaviors on P1 and P2, but quite little bias on P3 (See C6 in Table 3.3).

With the increased number of benchmark applications, testing platforms, and threads, quite similar observations are made. Based on these observations, the 25 applications can be classified as the following:

Workloads preferring to Nested Paging (TDP-inlined)

This group includes `canneal` (P3), `bodytrack` (P2,P3), `dedup` (P1,P2,P3), `vips` (P1,P2,P3), `water_nsquared` (P2), `fft` (P1), `x264` (P2). A common aspect of these workloads is that *nested paging* yields better performances for them. They prefer *nested paging*.

Table 3.3: Summary of the results observed in benchmark

workload	Platform 1			Platform 2			Platform 3			
	nt=1	nt=2	nt=4	nt=1	nt=2	nt=4	nt=1	nt=2	nt=4	
barnes				19.76%						C1
canneal						14.49%				
dedup	22.65%					10.73%				
fft				10.03%						
lu_ncb				9.61%		10.97%				
radix				10.65%						
dedup	77.35%	90.11%	70.95%	98.19%	90.09%	78.29%	89.27%	80.68%	68.03%	C2
ferret							99.87%	99.57%	79.64%	
lu_cb				99.55%	99.35%	76.69%				
spl_raytrace	100.00%	96.67%	81.97%							
swaption				99.37%	98.99%	76.52%				
canneal							81.50%			C3
dedup	74.57%			63.06%			53.30%			
fft	90.13%									
vips	87.54%			89.13%						
water_nsquared				93.56%						
barnes	103.83%									C4
fft				107.77%						
fmm							101.73%			
lu_ncb				102.17%						
radix				107.28%			106.58%			
bodytrack			105.56%							C5
canneal						103.92%				
fluidanimate			106.53%							
lu_cb							102.09%	102.09%		
radix					108.66%			102.97%	103.47%	
spl_raytrace			116.62%							
streamcluster			110.07%							
vips			114.99%							
water_spatial						113.71%				
x264						117.66%				
fft	90.13%			107.77%			98.67%			C6
x264				95.79%			101.71%			

1. nt stands for the number of threads;
2. C1 entries are the normalized performance loss in percentage;
3. C2 entries are the normalized performances in percentage;
4. C3, C4, C5, C6 entries are the ratio of performances between shadow and nested paging;

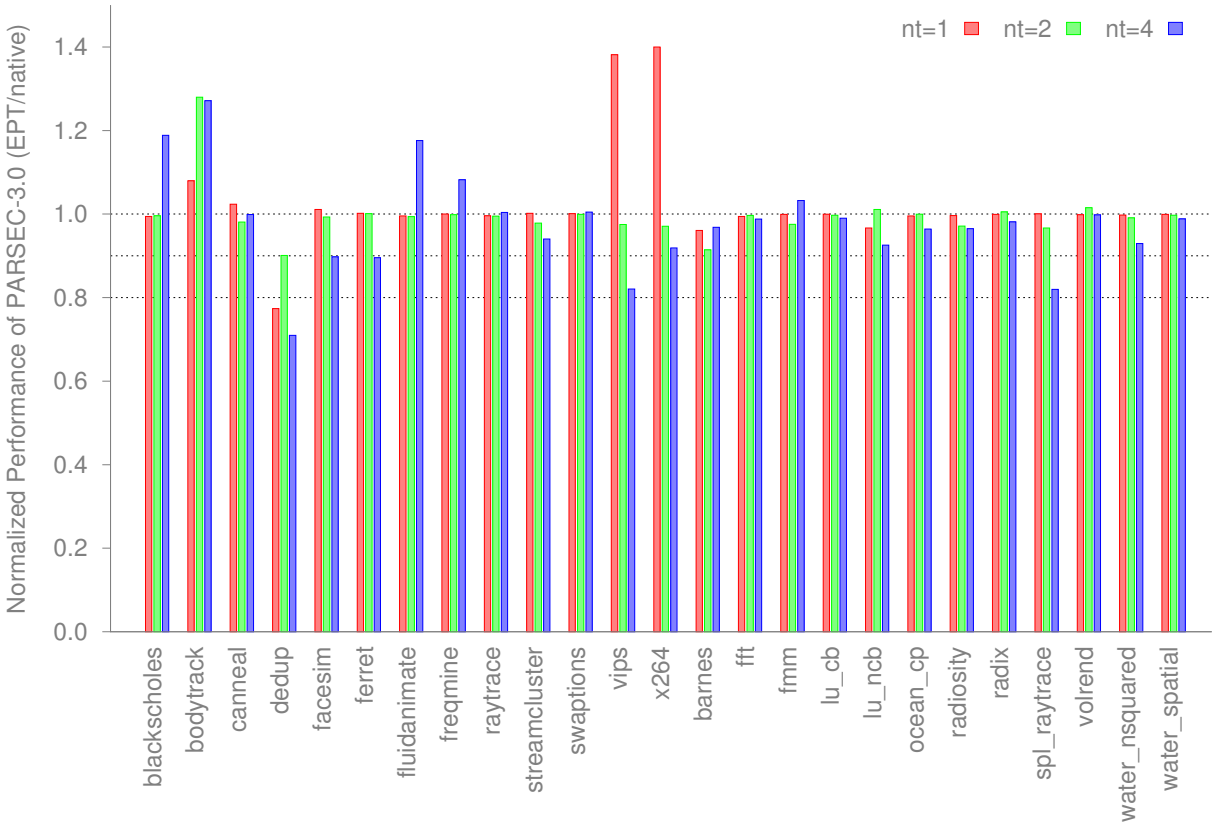
Workloads preferring to Shadow Paging (SPT-inclined)

This group embraces `barnes` (P1,P3), `fft` (P2,P3), `lu_ncb` (P2), `radix` (P2), `fmm` (P3), `radix` (P2,P3), `x264` (P1,P3). They exhibit better performance for *shadow paging* than *nested paging*. Their favor *shadow paging*. In that case, performance gain ranging from 1.73% (`fmm` on P2) to 7.77% (`fft` on P2) can be obtained.

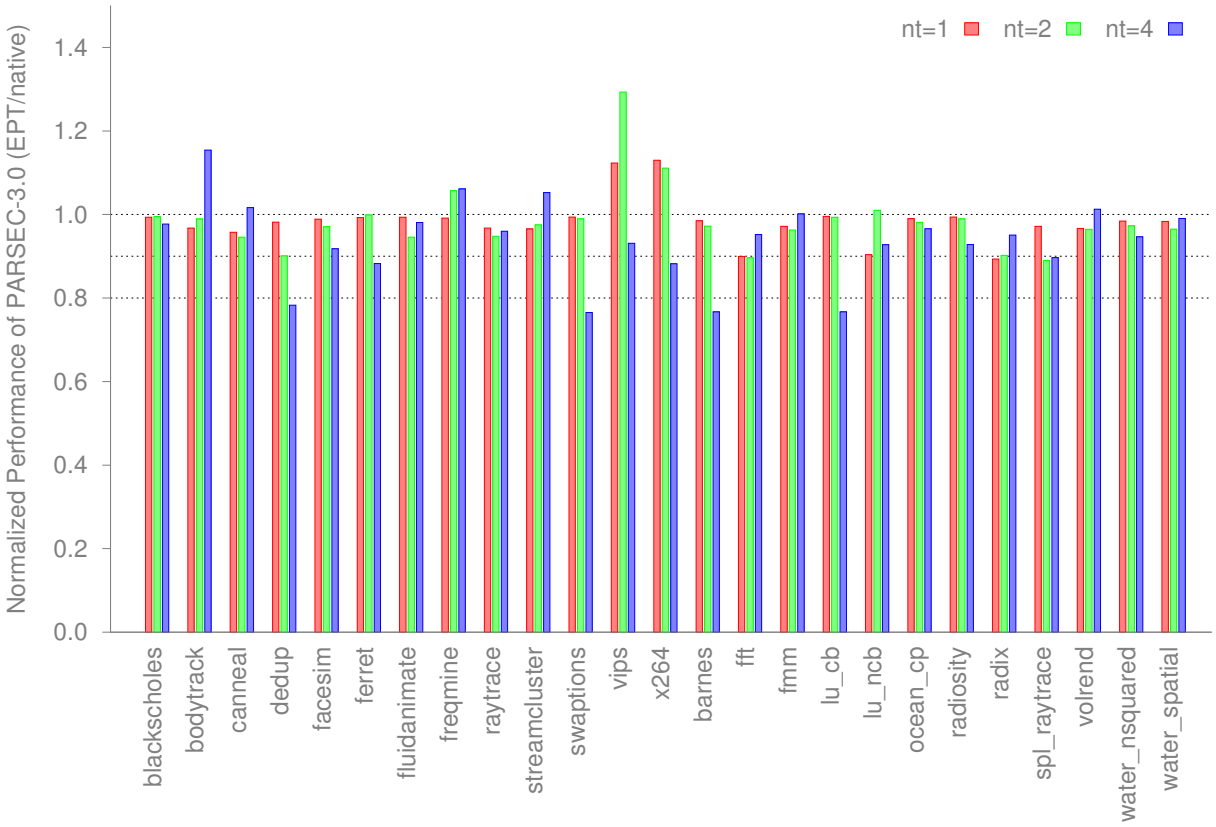
Workloads not too sensitive to Paging Method

This group includes `blackscholes`, `facesim`, `ferret`, `fluidanimate`, `freqmine`, `raytrace`, `swaptions`, `fmm`, `ocean_cp`, `radiosity`, and `volrend`. They are not too sensitive to particular a paging method.

In summary, a conclusion is that both the two paging methods have their strength and weakness, regardless of the testing platform, benchmark application and page size.

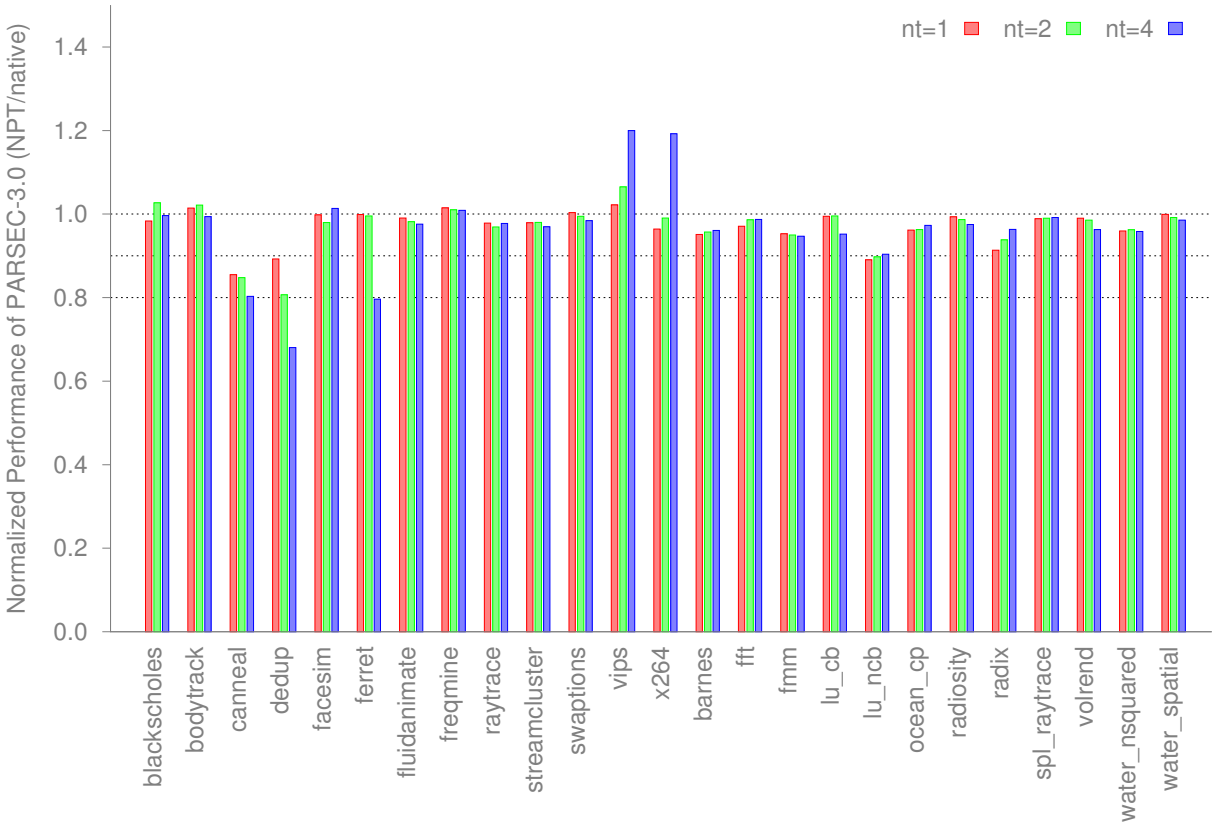


(a)

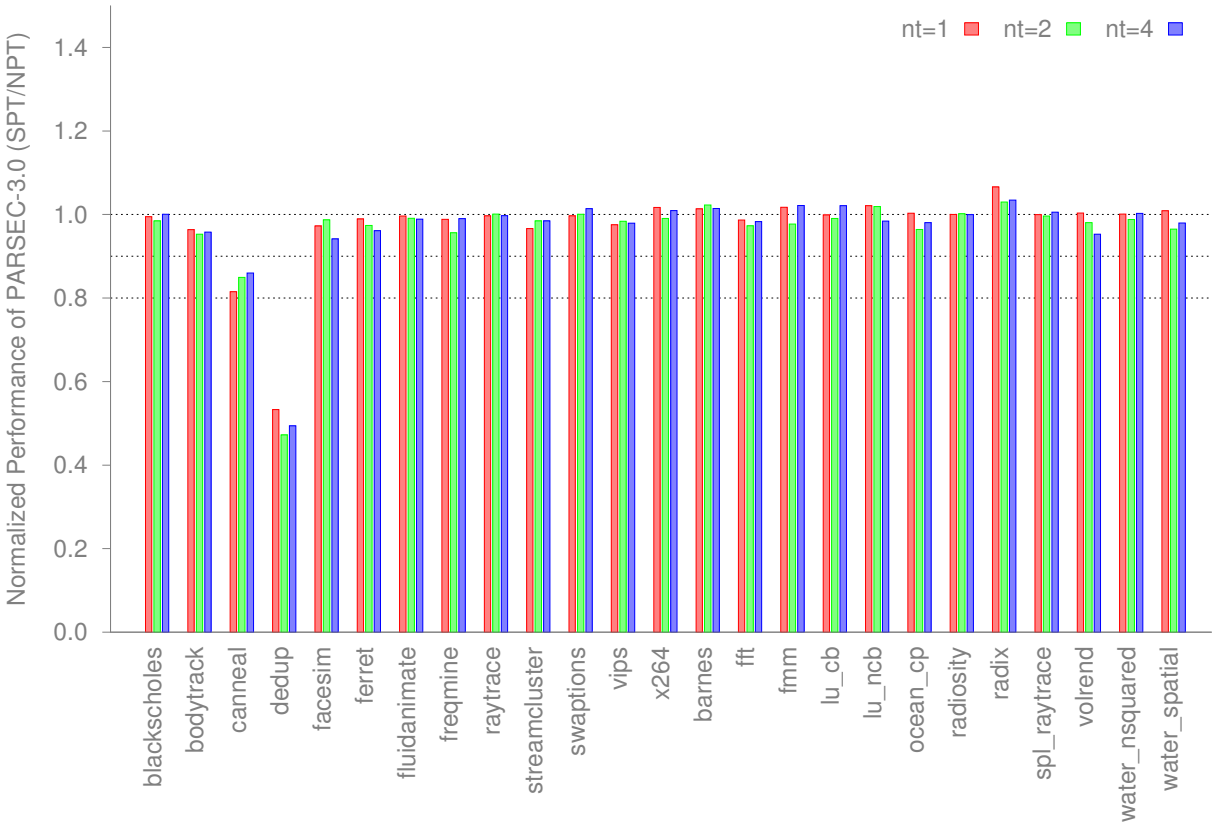


(b)

Figure 3.4 Normalized performance of PARSEC 3.0 in KVM with 1, 2, and 4 threads on (a) Platform 1 (Intel Core i7-6700K), (b) Platform 2 (Intel Xeon e5-1620-v2)

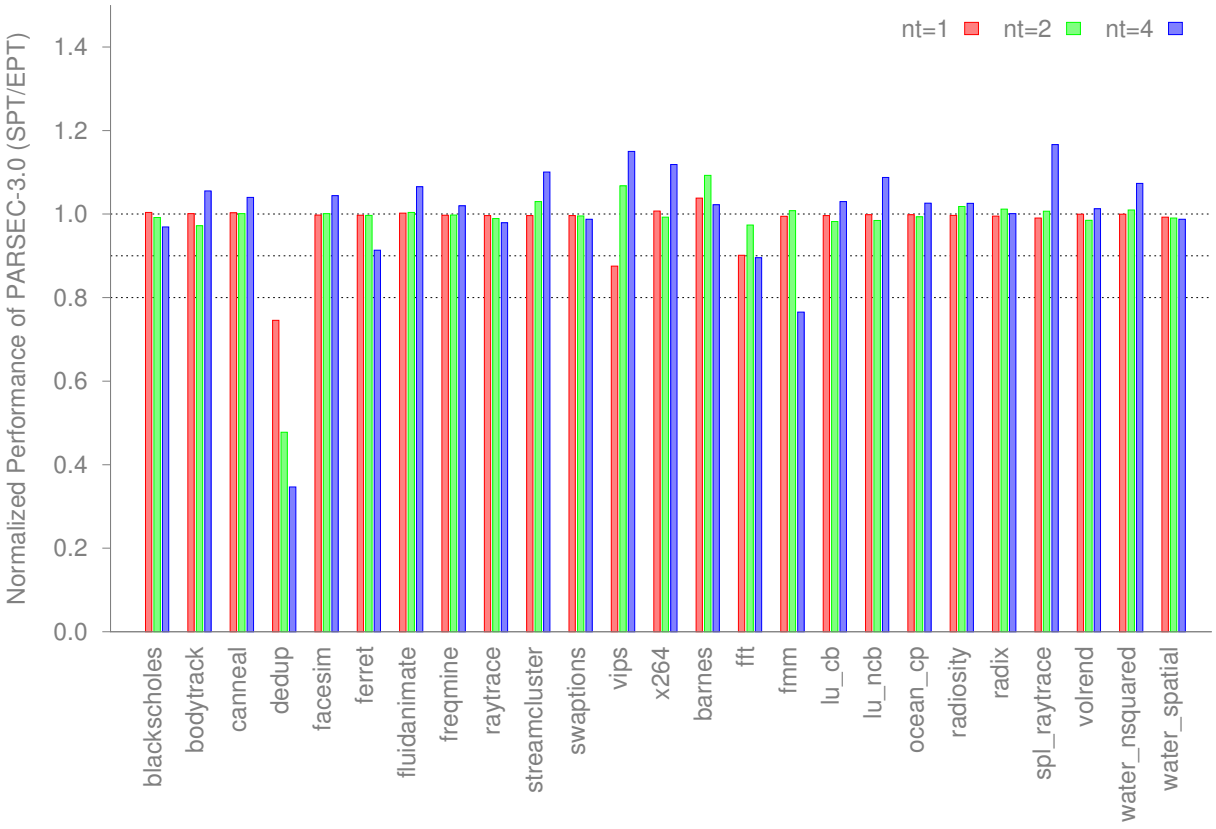


(a)

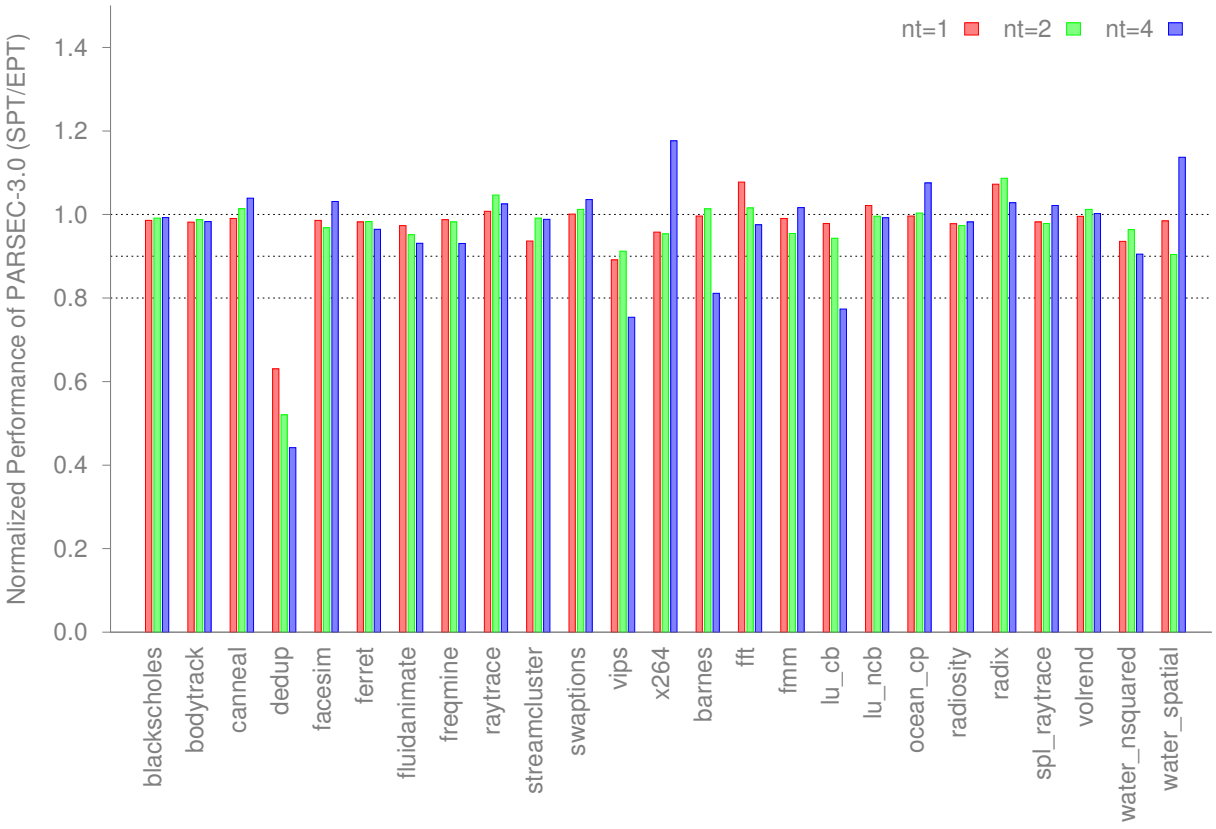


(b)

Figure 3.5 Normalized performance and comparison on Platform 3 (AMD FX-8150) (a) Normalized performance for NPT, (b) Performance comparison between SPT and NPT

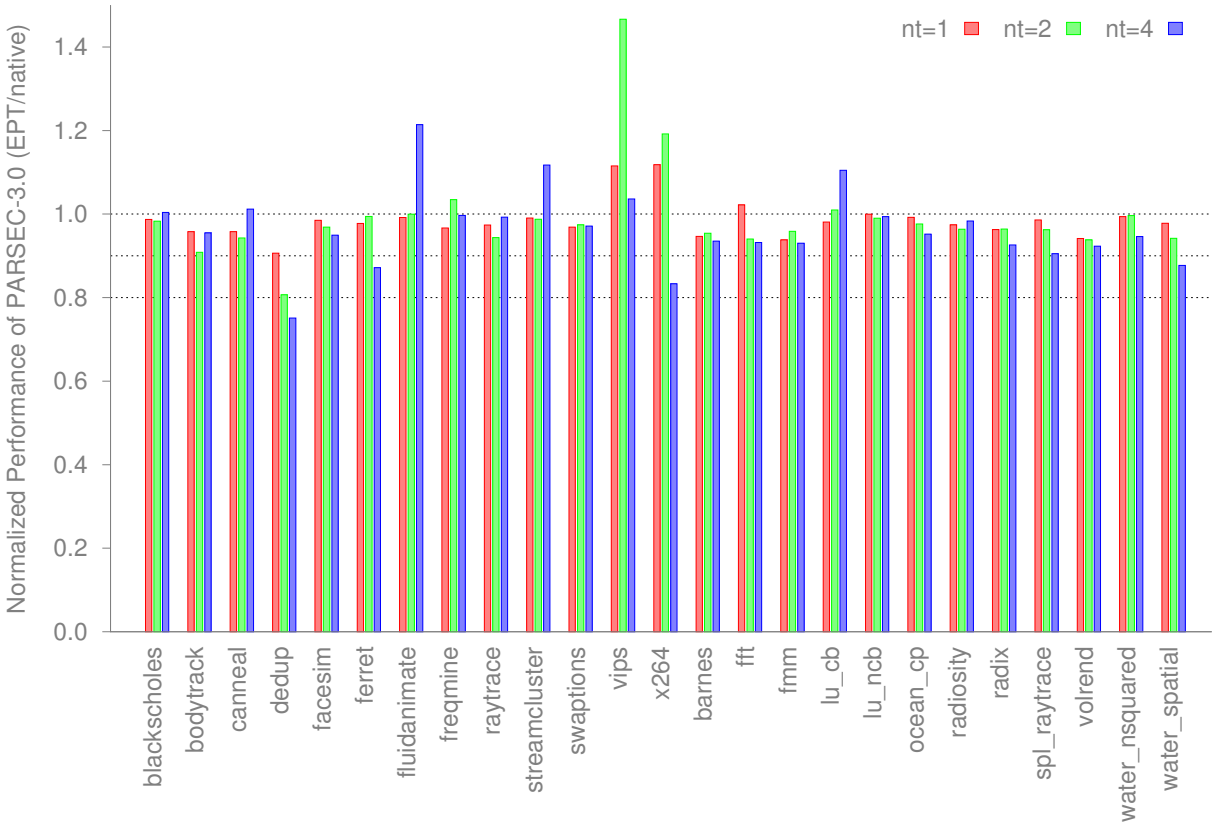


(a)

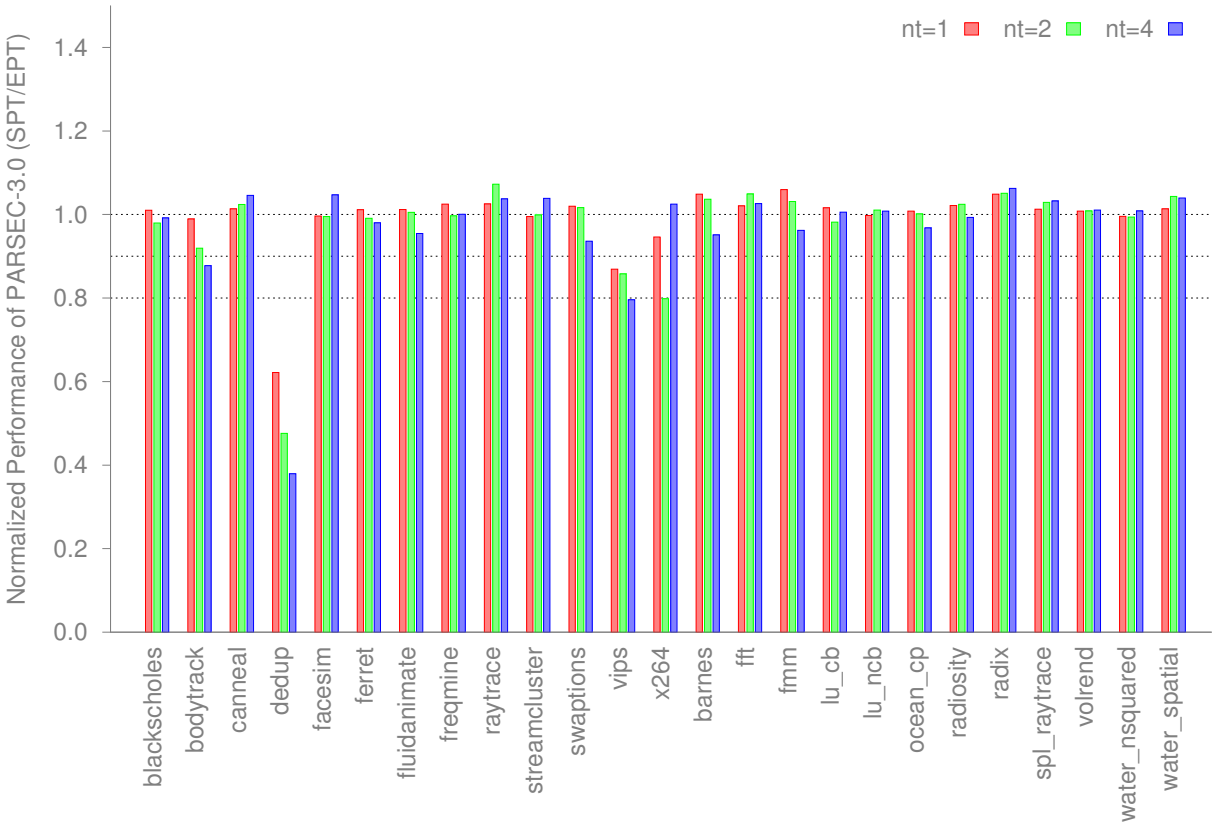


(b)

Figure 3.6 Performance comparison between SPT and EPT
(a) Platform 1 (Intel Core i7-6700K), (b) Platform 2 (Intel Xeon e5-1620-v2)



(a)



(b)

Figure 3.7 Performance comparison with the page table size of 1 GB on Platform 2 (Intel Xeon e5-1620-v2) (a) between EPT and native (b) between SPT and EPT

3.5 Summary

This chapter answers Questions 3, 4 and 5 stated in Section 1.4. By the nature and the demand for resources, HPC workloads are classified into multi-threaded type running in the intra-node pattern and multi-process type running in the inter-node pattern. The benchmark applications are discussed and selected for the intra-node pattern. Due to its diverse workloads and ease of use, PARSEC-3.0 is selected. To yield more generic benchmark results, three testing platforms are specified. Their features are similar but not the same. The results revealed the limitations encountered by the current solutions for virtualizing the guest memory address translation.

Chapter 4 DPMS - Dynamic Paging Method Switching and STDP - Simplified Two Dimensional Paging

Contents

In Chapter 3, the performance loss for a bunch of typical real-world workloads are measured and evaluated for the intra-node patterns in a virtual cluster. The benchmark results indicate that both techniques for memory virtualization have their strength and weakness when dealing with the diverse workloads. Neither guarantees to handle all workloads equally well.

This chapter presents two generic ideas as remedies to this performance drawback in memory virtualization and the paging translation. One is DPMS (dynamic paging method switching), which aims to make the best use of the two paging methods based on a run-time analyzing of the performance data of the running workload. The other is STDP (simplified two-dimensional paging), which attempts to reduce the cost of traversing the nested page tables by adopting a new paging scheme.

In Section 4.1 the benchmark results is reflected, these two solutions for memory virtualization in virtual machine guests are reviewed and the ideas are proposedd. Section 4.2 presents DPMS. From the overall design to the functional units, each aspect of the design is elaborated with an emphasis on its independence from the implementation on a concrete hypervisor. Section 4.3 is dedicated to STDP, with a focus on its feasibility to reduce the paging cost by restructuring the paging scheme for both software and hardware. Due to the lack of appropriate hardware support, the discussion is based on the assumption that this functionality may be supported in future by hardware, thus the software side is focused. Section 4.4 is a brief summary of the innovation reflected by both ideas.

4.1 Reflections and Solutions

An observation from the benchmark results is that two approaches are possible to remedy the performance drawback in the currently used paging techniques.

- Make the best use of the *shadow paging* and the *nested paging*;
- Modify the structure used by the more promising current solution.

These prompt the core ideas for a contribution to this topic, namely, *dynamic paging method switching*, and *simplified two-dimensional paging with large page tables*. Both ideas are described briefly and explained in further detail.

4.1.1 Dynamic Paging Method Switching

This idea is inspired by the observation that both *shadow paging* and *nested paging* have strength and weakness, which remains even when large page tables are used. The current hypervisors adhere to the static configuration of the paging method when the physical host is booted up. A consequence is that during the execution of the guest, the guest workloads cannot exploit the benefits of both the *shadow paging* and *nested paging*. The diverse nature and run-time behavior of the workload, which may incur large differences in performance has not been used by the hypervisors. In the attempt towards a cleverer hypervisor, the "footprint" of a workload should be used to aid the creation of a more favorable environment for the running workload. With the support of *dynamic paging method switching*, the hypervisor becomes capable of capturing the run-time traits of the workload, and acting accordingly by adjusting the paging method. In this approach, the workload may exploit the benefits of the two paging methods and yield higher performance than in the cases where the paging method is statically configured.

4.1.2 STDP with Large Page Table

Unlike the *dynamic paging method switching*, STDP represents the efforts in an entirely different dimension. The objective is not to exploit two paging methods, but to modify the more promising paging method in a way that the negative side is significantly diminished if not entirely eliminated. The benchmark results indicate that the more promising option is the *nested paging*. Due to the overall advantage towards its predecessor, this solution has the potential to be realized more efficient. The performance overhead of *nested paging* lies in the traversing of the two-dimensional page tables. Thus the performance of a workload is largely determined by the reusability of the TLB entries. If the workload is not able to reuse the TLB entries well, TLB misses may occur frequently and forces the processor to do expensive traversing frequently.

This is the scenario where huge page (2MB) or large page (1GB) can contribute their strengths. A TLB entry that covers 4KB of memory, covers 2MB or 1GB when larger page is used. The enlarged coverage of TLB entries may reduce the TLB misses and do benefits to the performance at certain cost. While the enlarged page size lowers the caching stress for TLB, there are a few downsides. First, the hypervisor still tracks the usage of each memory page. In cases of memory shortage, it frees memory by swapping out the less often used pages. The same occurs also to huge pages or large pages as to 4KB pages, but the cost is significantly higher to swap out 2MB or 1GB than 4KB. If a large page is only accessed in a random and sparse way, the benefit may be offset by the swapping cost.

Fragmentation of the memory is another downside with large page size. After many repeated allocations, contiguous physical memory is fragmented into pieces of various size. Inside a unit of allocation, there are external and internal fragmentations. The former indicates that the total memory size is big enough to satisfy a memory allocation request. However, it is not contiguous, thus difficult to be used by a process. The latter implies that the memory size assigned to a process is more than sufficient, so that some portions cannot be used by any process. Huge page

and large page are mainly bothered by the internal fragmentation problem. The idea of STDP is proposed to overcome these downsides. The following merits may justify an implementation:

- Two steps may be saved for a single mapping from the guest physical to host physical address;
- STDP is free from the heavy swapping cost and fragmentation suffered by using the huge and large page, therefore, the merits of using normal page size (4KB) in the guest are preserved.

4.2 DPMS - Dynamic Paging Method Switching

As described in Section 4.1.1, the current practice is to determine the paging method before the guest execution. The limitation is that it may incur more overhead than the other if the current paging method is not suitable for the workload.

Supposing that the paging method could be changed in response to the ever-changing behavior of the running workload, the overhead may be minimized. DPMS is designed for combining the strength of both *shadow paging* and *nested paging* at run-time. The purpose is to periodically adjust the paging method based on the sampled performance data for the workload. As Figure 4.1 depicted, four basic function blocks are necessary for this design, namely, *Performance Data Sampling*, *Data Processing*, *Decision Making* and *Paging Method Switching*. The subsequent subsections are dedicated to the details for each function.

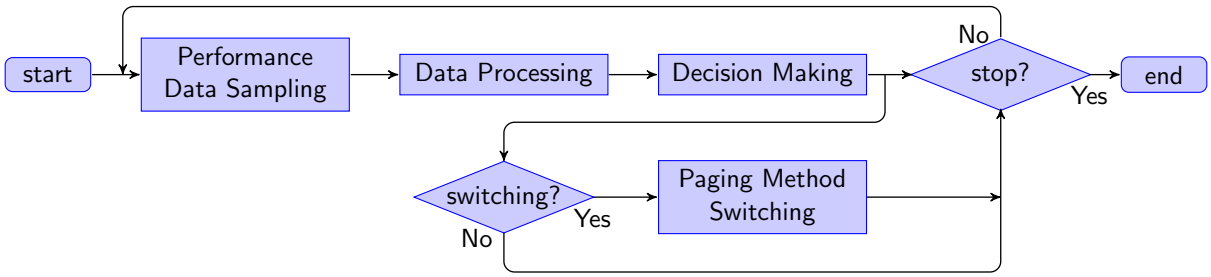


Figure 4.1 High-level design of DPMS

4.2.1 Performance Data Sampling

DPMS, in the view of *cybernetics*, is a self-adaptive system that constantly reacts to the changes occurring to itself by adjusting its behavior, which leads to further changes it reacts to without external interventions. The change is an event triggered by the variation of the performance of the running workload at the run-time. As a reaction, the paging method will be periodically determined if a switching to its alternative can benefit the performance. Depending on the result of a calculation and comparison, the paging method is changed or not changed. In such a closed *signaling loop*, the sampled performance data is used as the *feedback* for further decisions and actions in the system. Performance data is used to denote the raw unprocessed data.

Performance metric refers to the output by processing the performance data. It represents a specific aspect of the performance and reflects how well a program interacts with the underlying execution environment.

Chapter 3 has discussed the limitations of both the *shadow paging* and the *nested paging*. To partially overcome these limitations by DPMS, the first step is to check whether the current paging method is still suitable for the workload.

As for the *shadow paging*, performance loss is directly incurred by the *vmexit* due to the page fault in shadow page table. The occurrence of this event is considered to be the major indicator to the performance. For the *nested paging*, since a complete traversing of the page tables could be five times (24:5) more expensive than in the physical machine, the performance mainly suffers

when traversing the multi-level nested page tables in the case of TLB-miss. TMR (TLB-miss rate) come as the main indicator. Meanwhile, the IPC (instruction per cycle), which indicates the overall speed of program execution, can be used to evaluate the effectiveness of the adopted action. In summary, the performance metrics to be calculated are:

- PFR - ratio of *vmexit* due to the page fault in the *shadow page tables* to the overall *vmexit*
- TMR - ratio of instructions incurring TLB-miss to the overall executed (retired) instructions
- IPC - ratio of instruction number to the clock cycle number

To calculate the above performance metrics, the following data need to be sampled:

n_{pf} – number of the *vmexit*s due to page fault in the *shadow paging* within a sampling period
n_{vmexit} – number of overall *vmexit* due to any reason within a sampling period
n_{tm} – TLB-miss number within a sampling period, including d-TLB and i-TLB
n_{ret} – number of retired instruction within a sampling period
n_{cycle} – number of clock cycles within a sampling period

of which n_{pf} and n_{vmexit} are normally statistics monitored in the context of a hypervisor software, hence obtainable simply by reading from both variables. The remaining, on the other hand, are events related to hardware and therefore gathered by sampling from the performance monitoring counters on the processor or logical processor.

4.2.2 Data Processing

With the raw data sampled from either software or hardware, the following performance metrics can be calculated by the formulas:

$$cur\_tmr = \frac{C \cdot n_{tm}}{n_{ret}}, \quad cur\_pfr = \frac{C \cdot n_{pf}}{n_{vmexit}}, \quad cur\_ipc = \frac{C \cdot n_{ret}}{n_{cycle}} \tag{4.1}$$

where C is a coefficient of the quotients to prevent too much floating-point precision loss during the division of two integers. It scales the numerator to an appropriate order of magnitude before the division. For convenience, C could be a power of ten, depending on the value of data as well as the required quotient precision.

In practice, *Data Processing* serves as the preparation for *Decision Making*, therefore is entirely determined by the requirements of the latter. Without knowing the data needed for *Decision Making*, no further details about the rules or algorithm can be laid down except for one thing. As DPMS is based on a predictive model, which relies on the comparison between the current and previous values of the concerned performance metrics, a number of historical data needs to be stored for further use. These data are expected to be updated incrementally by replacing the oldest value with the current one. The ring buffer comes as an ideal choice to store such data. Figure 4.2 depicts the ring buffer for this.

The current element in the queue is pointed to by a variable and filled with either the current value of the metric, or the result of a comparison. All values in the queue may be evaluated by summing them up with a weight for each value. While the oldest value is overwritten, it must also be subtracted from the total value. Further details of the *Data Processing* will be presented in the implementation of DPMS for a concrete hypervisor.

4.2.3 Decision Making

Decision Making is an important part of DPMS. It reflects the intent of the workload based on calculations and comparisons. This is a critical proccess, as it does not only has a strong impact on the overall performance, but also determines whether the use of DPMS is worthwhile.

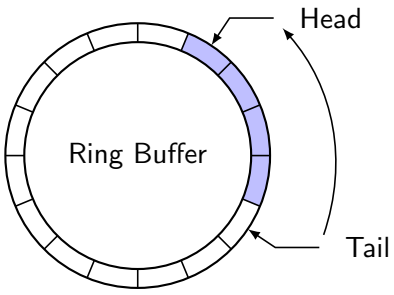
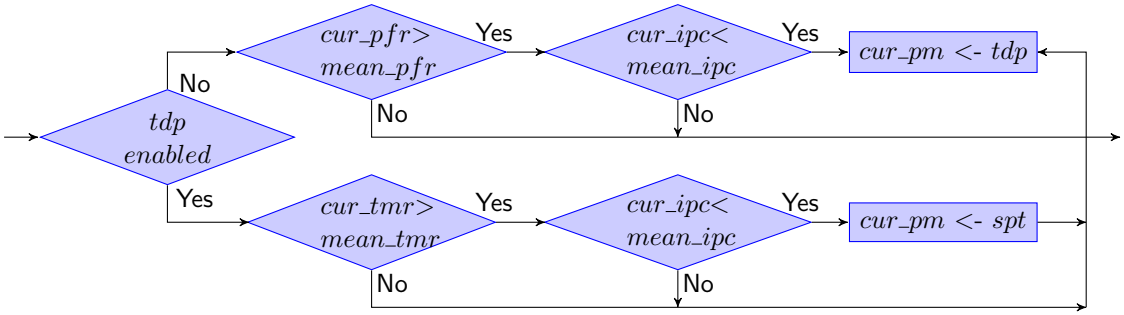


Figure 4.2 Ring buffer update

The purpose of *Decision Making* is to instruct a switching to the more suitable paging method for the current workload based on an analysis of the run-time data. The chosen paging method is expected to make the running workload more efficient by eliminating *vmexit*s or making better reuse of the TLB. These are also the criteria to test the effectiveness of *Decision Making*.



cur_pm: the current paging method

Figure 4.3 Flow chart of a basic algorithm for *Decision Making*

Depending on the choices of various factors for performance, the algorithm for making decision can be complex. But the principle is simple, making adjustments as soon as the paging method does not suit the workload any more. The input data is a key aspect to establish the correlation between the desired paging method and the sampled performance data. As an example, Figure 4.3 depicts the flow chart of a basic algorithm for making decisions based on PFR, TMR and IPC. In this logic, the first thing is to check which paging method is currently used. This is done by checking a global variable in the hypervisor's context. If *nested paging* is disabled, PFR and IPC are the two metrics to observe. The *shadow paging* is preferred when PFR is rising and meanwhile IPC is falling, otherwise no change is needed. If the *nested paging* is enabled, TMR and IPC are the metrics to observed. The *shadow paging* is desired if TMR is rising while IPC is falling, otherwise no change is needed.

The logic is simplified for making decision. In practice, however, this may not be much useful, since the correlation between the desired paging method and the sampled performance data may not be so straightforward. More factors tend to be involved. The flow chart is mainly based on three considerations: 1) Sufficient sensitivity, which means that the logic should be triggered as quickly as possible to react to the performance data; 2) Sufficient stability, to avoid unnecessary switching (jitter or oscillation) incurred by the misprediction; 3) Simplicity, ensuring a quick response to the changing-behavior of the workload and low cost for DPMS itself.

4.2.4 Switching Mechanism

With DPMS, the paging method is expected to adjust dynamically to react to the ever-changing workload. Compared to the conventional way that configures the paging method once and for all before the execution of the guest, the run-time reconfiguration of the paging method is desired by the workload. This adds complexity and probably overhead to the current hypervisor.

However, it is worthwhile due to the possible performance gain. As the central part of DPMS, the paging method switching involved mainly the following aspects:

1. Which part of the hypervisor is suitable for adjusting the paging method?
2. How to notify the hypervisor of a paging method switching desired by the workload?
3. How to ensure a smooth transition between the two paging methods?
4. How to reconfigure for the chosen paging method?
5. How to avoid oscillation or minimize its impact on the performance?

Depending on the concrete implementation of a software, answers to these questions may vary from one hypervisor to another. However, more or less they have something in common. A few general principles are worked out to aid the implementation of a specific hypervisor software for specific hardware.

To signal the intent on the paging method switching, a wise practice is to take advantage of the *vmexit* whenever it occurs. In other words, only do it when really necessary. This ensures that the guest execution will not be interrupted due to the intent of switching the paging method and incur as little overhead as possible. Figure 4.4 depicts the occasion when or where the paging method switching should occur with this consideration. DPMS takes any opportunity during a *vmexit* due to any reason rather than forcing *vmexit* for it own purpose. In any hypervisor for any architecture there is such occasions to perform it.

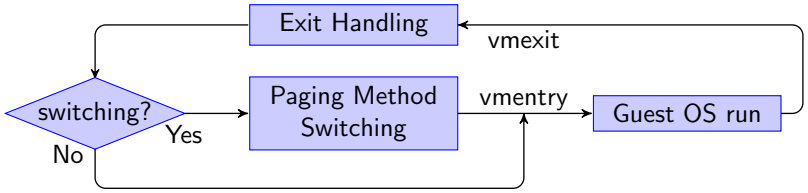
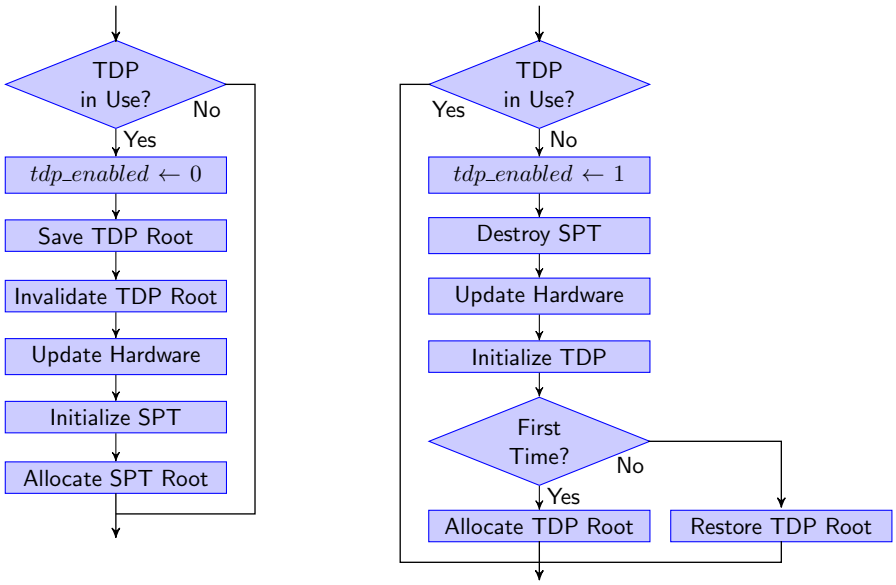


Figure 4.4 Occasion for PM switching in the execution flow

By comparing the current and the expected paging methods, the intent to switch the paging method is checked by the hypervisor. If different is expected, a flag is set to signal this intent, and the corresponding action is taken before the next *vmentry*. Otherwise nothing changes. In this way the hypervisor knowns the intent of the guest workload and takes appropriate action if necessary.

The third and fourth questions are closely related with each other. A transition from one paging method to the other involves reconfiguring certain parts of the hypervisor and the processor. On the software side, the major thing affected is the page tables (the root of page tables as well). On the hardware side, it is the operating mode of the processor regarding to the paging method. A smooth transition is only possible when the processor's MMU (memory management unit) makes use of the selected page tables and calls the page fault handler accordingly.

In principle, different strategies exist for switching between the two types of page tables. For example, to retain or to destroy the current page tables before switching. However, considering that the two types of page tables have different stability and way of maintenance, not all strategies are guaranteed to work. As the shadow page tables are more volatile than the nested page table, they may contain inconsistent entries when returning from its alternative. On the other hand, the nested page tables are not burdened with maintaining the consistency with the guest page tables. So even the entry changes, it occurs in a much slower pace than the shadow page tables. Therefore, a reasonable choice is to retain the nested page tables, rebuild the shadow page tables when switching from the *nested paging* to the *shadow paging*, but to discard the shadow page tables and restore the nested page tables in the reverse process.



(a) TDP to SPT (b) SPT to TDP

Figure 4.5 Basic operations for Paging Method Switching

Naturally, an alternative way is to retain the shadow page tables and keep them synchronized with the guest page tables under *nested paging*. So the efforts for rebuilding the shadow page tables can be saved at each time. However, new effort must be taken in tracking the modifications in the guest page tables and updating the shadow page tables accordingly. This not only incurs overhead, but also adds complexity to the hypervisor. Figure 4.5 depicts the control flow for switching the paging method.

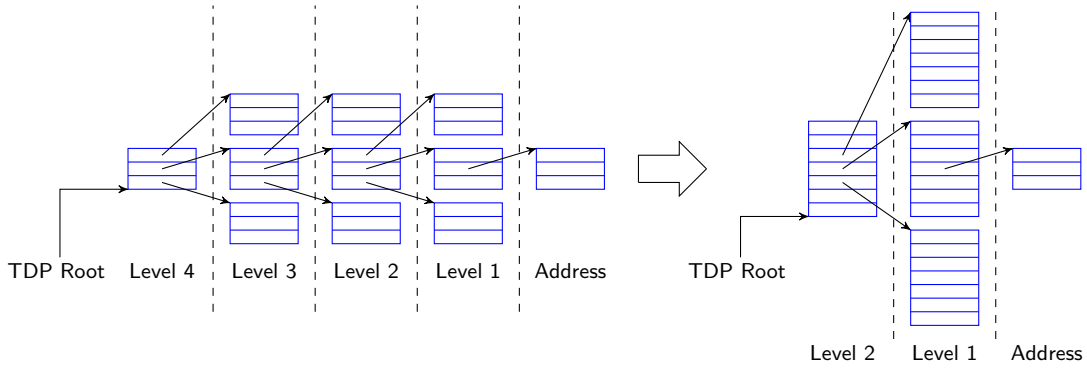
Nowadays, HPC is dominated by the x86-based architecture. The idea of DPMS is focused on this architecture. However, its applicability is not limited to x86-based architectures. The HPC history has witnessed the rising and fall of several dominant processor architectures. On the road to exa-scale computing, the adoption of energy efficient technologies has been a trend in HPC. ARM processor, for example, is expected to contribute its strength in this aspect. MIPS, PowerPC, UltraSPARC also take slight shares in the total installations of HPC system.

The applicability of DPMS on the non-x86-based architectures is a little complex. A few ISAs, such as the recent ARM, and PowerPC, have built in mechanisms similar to *shadow paging* and *nested paging*, which is known as *Stage-2 MMU* in ARM for performing *nested paging*. Shadow page tables are created by referring to the SLB (segment lookaside buffer) and HTAB (Hashed page TaBles) on PowerPC [127]. To the best of our knowledge, for these architectures, hardware-assited memory virtualization mechanism has not been reported yet. Generally, the virtualization technology for such processor architectures is far from mature compared with that for the x86-based architecture, probably due to the less demand in production. Nevertheless, at least for ARM processor, DPMS is applicable.

4.3 STDP - Simplified Two-Dimensional Paging

The Simplified-TDP with Large Page Table (STDP for short) is another approach to diminish the performance loss for memory virtualization. Unlike DPMS, whose operation is a mix of the TDP and SPT, STDP is merely a variant of the standard TDP (nested paging), with the idea to enhance it by simplifying the internal paging structures. Therefore, not only the overhead incurred by the *vmexit* due to the need of synchronizing the shadow page tables can been avoided, but also the cost for traversing the multi-level nested page tables can be considerably reduced.

Such a modification requires a co-design of software and hardware, and to creation the function in true hardware is beyond the hypervisor researcher's scope. A possible solution is to use a full system emulator. At present, the idea merely serves as a prototype to study the feasibility of adopting the new paging scheme by the processor for a better support to memory virtualization.



(a) Traditional TDP (b) STDP

Figure 4.6 High-level Design of STDP

Figure 4.6(b) is an overview of the STDP design, in comparison with the traditional paging scheme adopted by the nested page table walking in Figure 4.6(a). The STDP transforms the four-level page table tree into a two-level "fat" page table tree at the software level.

The STDP consists of two parts: the software part with the restructured nested page tables and the hardware part with the adaptive MMU for *nested paging*. Both are discussed below.

4.3.1 Revisiting the Current Paging Scheme

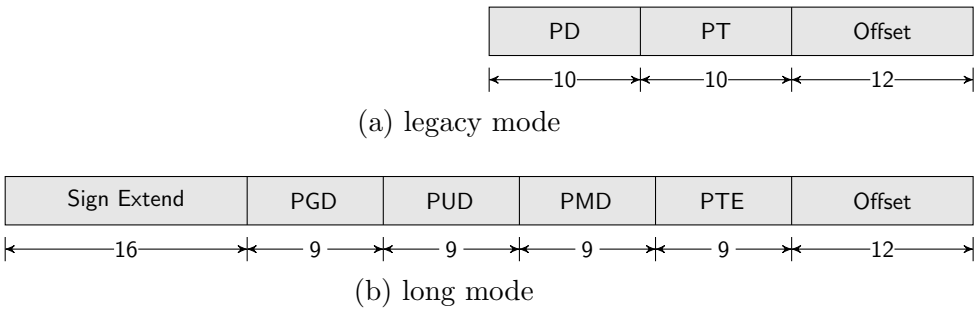
Shadow paging is a software-based solution for memory virtualization. As Figure 2.3 depicts, the major infrastructure is a group of shadow page tables under the control of the hypervisor. To perform the translation from the GVA (guest virtual address) to HPA (host physical address), the *shadow paging* combines the three intermediate steps for each GVA→HPA into a single entry by a series of address mapping in the hypervisor. In this way, the shadow page tables are filled gradually with the ultimate host physical address for the corresponding entries in the guest page tables. If a page fault occurs in the guest for any reason, the shadow page table entries must be freshly filled or updated at the time when control is returned to the hypervisor.

However, since this kind of shadow pages is a software infrastructure of the hypervisor and must be maintained as consistent as possible with the guest page table, the processor had to switch constantly from the guest (non-root) to the host (root) mode to update the shadow table pages. During this period of time, a considerable number of CPU cycles may have been wasted.

As a more recent solution, the *nested paging* allows the GVA→GPA translation be retained in the guest, while assigns the task for translating the GPA→HPA to the processor. The expensive *vmexit* due to the guest page fault is unnecessary. Nevertheless, the weaknesses of *nested paging* are also obvious. For performance reason, the *nested paging* relies on the reusability of TLB entries. Since the TLB contains a number of the most recently accessed GPA→HPA mappings,

it is likely that these entries are still useful in future. In this manner more time can be saved for traversing the page table in subsequent operations. If the TLB miss occurs, multi-level page tables must be travesed to fetch the data from the memory. Due to this nature, the TLB may not be quite helpful in saving more efforts for page table walking when a running workload is not good at taking advantage of the TLB entries. Therefore the performance gain may more or less be offset by the loss suffered in walking the nested page tables.

Since neither the *shadow paging* nor the *nested paging* is guaranteed to handle a given workload equally well, one may wonder if any better solution exist. Within the framework of the current technology, this is difficult if not impossible. The performance of each solution depends largely on the ability to reuse the cached results of the previous page table walking. Is *shadow paging* an ideal solution for this? Shadow page tables cannot be maintained without interrupting the guest execution and returning to the hypervisor's context. Actually, to reduce the occurrence of page faults in the guest is the only way for better performance. However, this kind of memory access behavior tends to be workload-specific thus is beyond the control of the hypervisor. The *nested paging*, though also suffers, has the promise to improve the performance if it can be more capable to take the advantage of TLB entries, or mitigate the cost for traversing the nested page tables. While the former suggests the use of a larger TLB, the latter is the focus of this section.



(a) legacy mode

(b) long mode

Figure 4.7 Breakdown of the logical address for x86-64

Assumably for the sake of tradition and simplicity, the same paging scheme has been adopted for x86-64 architecture and the nested page table walking. As depicted in Figure 4.7(b), the 64 bits of a logical address is partitioned into six parts, namely, Sign Extend, PGD, PUD, PMD, PTE and offset. The Sign Extend indicates the *canonical* [128] and is currently not used. Except the 12-bit Offset, the remaining four parts are 9-bit long, and each serves as an index to the entry of a page table at the lower level. To translate an address in this format by the hardware MMU, the page tables on the software side also need to be organized in this 4-level hierarchy. Since each entry for a page table takes 8-bytes, a 4KB-page contains 2^9 entries. An hierarchy formed by these page tables is an N-ary tree structure [129], where N is 1024 in 32-bit (legacy) mode, and 512 in 64-bit (long) or 32-bit PAE modes.

In the legacy mode, as depicted in Figure 4.7(a), two levels of paging are applied, the overhead may not be obvious. However, the overhead grows proportionally to the level. Given that the time for a single memory access is constant, the time for searching in a 4-level tree structure is about twice as that in 2-level tree structure. Even though, it does not matter too much for paging in a physical machine, but dose in a virtual machine. The basic reason is that for a virtual machine, if the *nested paging* is used, the cost difference between 4-level and 2-level page tables can be magnified by a factor of 5 in the worst case due to the same 4-level paging scheme in the second dimension.

So far, the *nested paging* has simply adopted the same schemes as that in long and PAE modes for traversing the two-dimensional page tables, regardless of which paging scheme is applied by the guest. This have been done probably for two reasons. First, it may have been convenient for the MMU to walk the nested page tables in exactly the same paging scheme as in the guest.

The perfect compatibility between the host and guest paging modes helps to avoid further complexity in both the hypervisor (software) and the processor. Second, multi-level page tables is more favored over a single-level page table for reasons of memory saving and higher efficiency of memory utilization. On the other hand, to the nature of a tree structure, the deeper a tree is, the more expensive the traversing will be. From a performance point of view, a multi-level paging scheme is worse than a few-level paging scheme.

Specifically, for the worst case of the 4-level paging scheme, a TLB miss may occur each time. A single GVA→GPA translation costs 24 times of memory accesses in all the page tables, which is prohibitively expensive compared to five times in the physical machine. However, in Figure 3.3(c)(d) and Figure 3.7(a)(b) it can be seen that workloads are incapable of making good use of TLB exist (`dedup`:10.0%, `x264`:21.8% on P3; `dedup`:9.37%, `barnes`:5.35% on P2), even if huge or large pages are applied. Is it possible for the *nested paging* to adopt a paging scheme with fewer levels? If so, how many levels are reasonable for the nested page table walk?

In fact, during a GVA→GPA→HPA translation, the two sets of page tables traversed by the MMU are independent. Just as the long or the PAE mode (scheme) have been chosen by the *nested paging* to support the guest paging, the hypervisor is free to adopt any paging scheme for traversing the page tables in the second dimension without having to maintain this kind of compatibility [104]. The only limiting factor is the processor's capability of switching between different paging schemes when traversing the nested page tables and the guest page tables.

Theoretically, the paging scheme for the *nested paging* has a variety of choices, ranging from the single-level lookup tables to the multi-level hierarchical ones. In practice, it is usually a trade-off between the memory utilization and the performance. Currently, the 4-level scheme keeps a balance for most of the 64-bit operating systems. Therefore, it is simply adopted by the *nested paging*. An advantage is that this does not add complexity to the processor when traversing the guest and nested page tables. However, considering that the 4-level paging scheme yields 24-times memory access in the worst case, this can be a potential source of overhead for a number of workloads common in the real world. It is not an ideal choice for the *nested paging*. The task is to find a new balance between the memory utilization and the performance based on the priority, which drives the paging scheme to move in the direction towards a simpler structure with fewer levels. A new balance means that the paging needs fewer memory access at the cost of higher memory consumption for keeping the page tables. The hypervisor also has an influence in this respect. A few possible candidates are: 1-level, 2-level, and 3-level paging schemes.

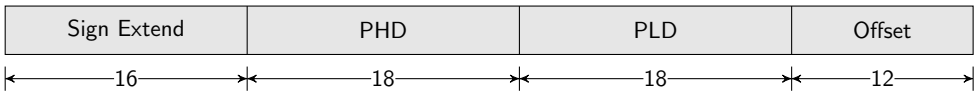
The 1-level scheme uses a huge linear array indexed by the virtual addresses [1] and yields the page frame number for each associated virtual address with a single look-up. While this agility is advantageous to the performance, the extremely huge memory consumption for saving the entries in an 1-to-1 manner is a problem. As the array can be huge, this scheme is extremely wasteful. Even a large chunk of the virtual address range is unused for paging, the corresponding entries must still be reserved in the array, with the pointers to pages being null. For the currently used largest virtual address space, $[0, 2^{48}-1]$, 2^{48-12} page table entries (2^{48-21} page tables) are needed. These may take $2^{48-12+3} = 512$ GB memory space, roughly $1/2^9 \approx 0.2\%$ of the maximal memory space of 256 TB for the current x86-64. Another issue is that such a scheme demands a huge memory space both virtually and physically contiguous.

The currently used 36 bits by indices can also be split evenly into three levels, with each level indexing to $2^{36/3}$=4K entries contained in eight 4KB-pages. The total amount of memory space can also be $(2^0 + 2^{12} + 2^{24}) \cdot 2^{3+12} \approx 512$ GB, almost the same as consumed in the 1-level or 4-level paging scheme. Nevertheless, the mapping is 1-to-many, and most of the entries may not be used or filled at once for a given time. The actual consumption is quite likely much lower than the approximate limit. The unfavorable things are: 1) five memory accesses may be saved in the second- dimensional walking – not quite striking compared to the default 4-level scheme; 2) Much effort is needed to adapt the data structures and functions in a hypervisor.

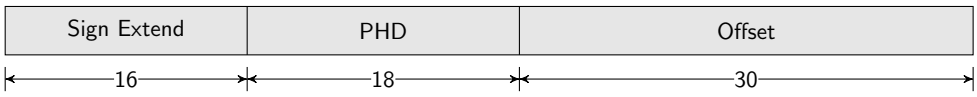
[1] more precisely, the page frame number in the guest physical address space

In one word, the lost-gain ratio does not take a stand in favor of this solution. Therefore, only the 2-level scheme is left. Similar to the other schemes, it consumes maximal about $(2^0 + 2^{18}) \cdot 2^{9+12} \approx$512 GB memory space, but the actual size may lie between the size of 3-level scheme and 1-level scheme. Compared with the 4-level scheme, 10 memory accesses can be eliminated when traversing the second-dimensional page tables, which means a $^{10}/_{24} \approx$40% decrease in performance loss. Furthermore, although the data structures and functions are also affected by such a change, the adaption may be relatively easy due to the "double"-relationship between the indices in the new and old schemes. The infrastructure of the current hypervisor software may be better reused. For these merits, this form of 2-level paging scheme is actually the optimal choice not only among the possible three, but also for the trade-off between the performance and the efficiency of memory utilization. With this scheme, the cost for traversing the second-dimensional page tables will be reduced by about 40% in theory. Even a workload is by nature not cooperative with the TLB, a page table walk does not hurt the performance too much. Figure 4.8(a) depicts the basic form of the 2-level paging scheme, where the lower 48 bits of an address is partitioned into three parts - PHD, PLD and the offset.

In a modified version, as shown in Figure 4.8(b), with the combination of the first level index "PLD" and the offset, a huge page form for the 2-level paging scheme can be yielded, which will further eliminate 5 more times for memory access, and reduce the cost for traversing the second-dimensional page tables by 60%, compared with the basic form. Meanwhile, only $2^{18} \cdot 2^3 = 2$ MB memory space is needed for the page tables of a guest with 4 GB memory.



(a) Basic form



(b) Huge page form

Figure 4.8 The 2-level paging scheme for TDP

As a summary of the above discussion, Figure 4.9 depicts a comparison of the four paging schemes. The performance gain and memory utilization efficiency serve as two criteria for this. The performance gain can be roughly estimated by the times of the traversing a scheme saves compared to the default 4-level scheme. For simplicity, other performance influencing factors, such as the cache size, cache effect are not taken into consideration. The 1-level and 4-level schemes can be viewed as two extreme cases among the practically useful schemes. By trading off the two intrinsically conflicting criteria, the 2-level scheme is considered optimal, thus chosen as the paging scheme for STDP.

The choice of paging schemes is not a new topic, but it has a fundamental impact on the design of hardware, operating systems, compiler, user application, as well as application libraries. Since the advent of the x86-64 architecture, a major breakthrough in the Linux-kernel development is the release and gradual merging of the *four-level page table patches* [130] into the linux-2.6.10 series at the beginning of 2005, which symbolizes fundamental changes of the operating system kernel together with the enhanced processor hardware. "Now x86-64 users can have a virtual address space covering 128 TB of memory, which really should last them for a little while." [130]. After that, the 4-level paging scheme has almost been taken as the standard for software development and computer architecture. Until recently, with the release of the patch for *five-level page tables* [131], another fundamental change is drawing near to the paging scheme. Although the hardware support is not yet shipped by any processor vendor, the implication of "a little while" has been estimated as 12 years between the two events.

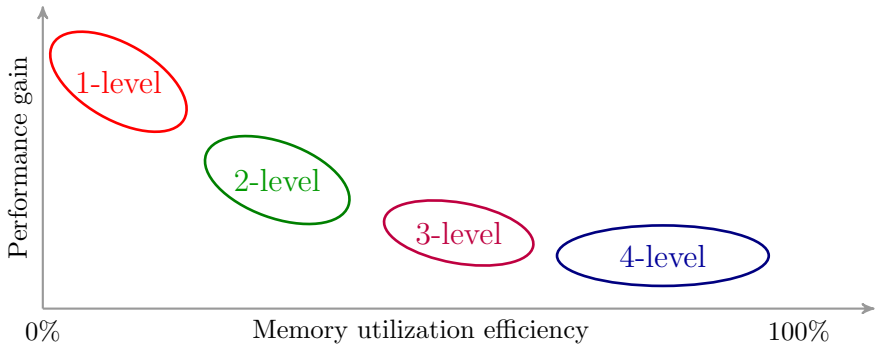


Figure 4.9 Balance between performance and memory utilization efficiency

This example demonstrates that fundamental change occurs rarely, but does occur at a certain moment with the advance of the current technology and the deepening of knowledge. Till now, a fundamental change to the paging scheme in the second-dimensional page tables has not occurred yet. But there is a reason to believe that hardware support to this capability will be available for the future processors.

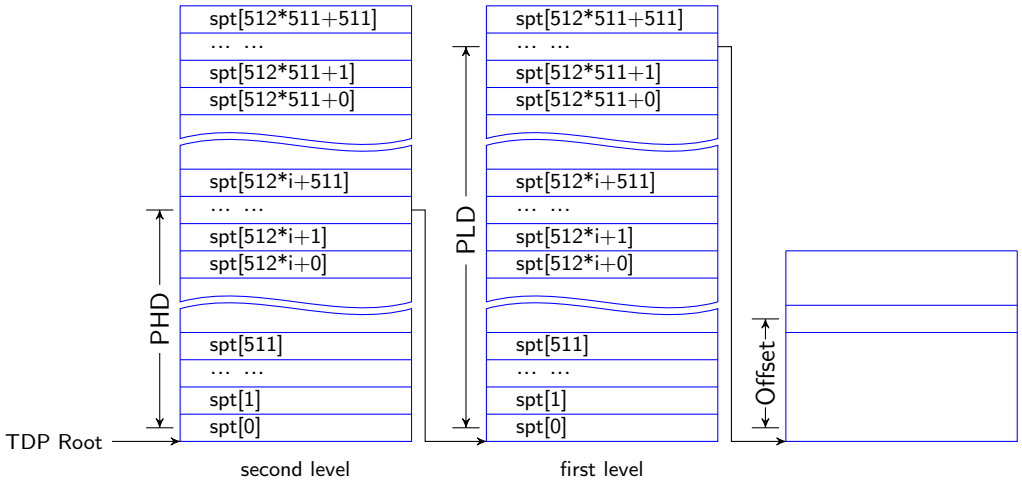


Figure 4.10 Restructured page table

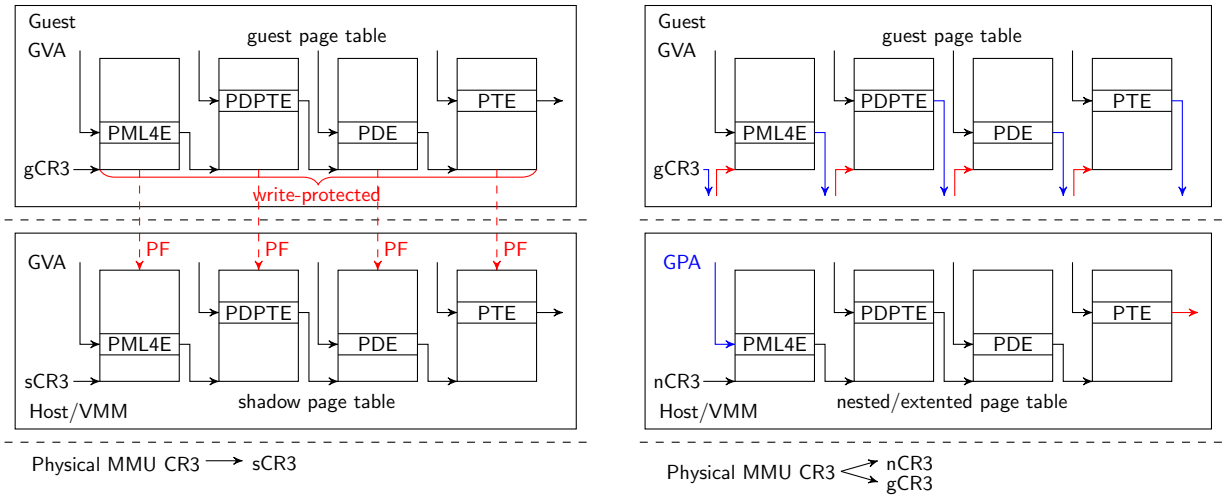
4.3.2 Restructured Page Table

As Figure 4.10 depicts, the term "restructured page table" gives a general impression of the tree formed by new page tables for the 2-level paging scheme, which leads to a tree dwindled in depth but expanded in width. Only two levels of page tables are involved, and the page table at each level contains 2^{18} entries. These are enlarged page tables. Each of them occupies $2^{18} \cdot 2^3$ =2 MB memory space. For a modern hypervisor, it should not be a problem to allocate the memory chunk of this size continuous both virtually and physically in kernel space. Based on the TDP Root, the 18-bit part "PHD" serves as an index of the entry for the page table at the first level. Similarly, the 18-bit part "PLD" is used to locate the page frame needed by the GVP→GPA as the intermediate result or the page frame number for producing the HPA as the ultimate result.

4.3.3 Page Fault Handling in TDP

It is necessary to gain a clear understanding of the page fault handling in the *nested paging* to create the STDP on hardware side. The fundamental distinction between the *nested paging* and *shadow paging* lies in their ways to create, utilize and maintain the page tables.

Figure 4.11 provides an overview of their operations. As the name implies, shadow page tables are a "shadow", or an equivalent of the guest page tables owned by the hypervisor. Therefore, the shadow and guest page tables are equal in serving as lookup tables for guest address translation. However, due to the need of maintaining an illusion, the guest page tables contain no real physical address, thus can not be used for paging directly. This task is performed by the shadow page tables in the hypervisor. By marking the pages occupied by the guest as "write-protected"[2], the hypervisor is informed upon a *vmexit* about the attempts to modify these pages. This gives it a chance to update the shadow page tables correspondingly. In this way, the shadow page tables are gradually filled and traversed by the MMU when a guest virtual address is translated.



(a) SPT Mechanism (b) TDP Mechanism

Figure 4.11 Overview of the SPT and TDP mechanisms

To be precise, the nested page table contains mappings from the guest physical to the host physical page frame numbers. It does not serve as a "shadow" or an equivalent of the guest page table, but as one step in the whole translation path. From guest virtual to host physical address, the translation is done by traversing both the guest and nested page tables in an alternative manner. Without considering the TLB's role, each mapping in the guest page table is interleaved with four mappings in the TDP page table, as depicted by Figure 4.12(a) in more detail. If the TLB is used with the *nested paging*, and the desired entry is cached, a significant number of walks in this dimension can be saved. Otherwise, both the guest and TDP page tables are walked by the physical MMU alternatively. While the guest page tables are maintained by the guest OS, the nested page tables are still by the hypervisor. A page fault in former does not have to trigger a page fault and a *vmexit* in the latter. As a result, the *vmexit*s caused by the faults in the guest page tables are effectively reduced to null.

To deal with page faults in the second dimension, both Intel EPT and AMD NPT specify a few conditions under which a *vmexit* occurs. Taking Intel EPT as an example, accesses using guest physical address may cause a *vmexit* due to the EPT misconfiguration, EPT violation, and page modification log-full events. EPT misconfiguration occurs when translating a guest physical address, if the logical processor encounters an EPT paging-structure entry that contains an unsupported value. EPT violation occurs when no EPT misconfiguration occurs but the EPT paging structure entries deny an access using the guest physical address due to other reasons. A page modification log full event occurs when the logical processor determines a need to create a page-modification log entry but the current log is full [195]. These events follow dedicated exception handlers rather than the one responsible for handling a wide range of events causing *vmexit*. AMD NPT does this similarly. It is this kind of separation that ensures an updated page table in the second dimension, while leaving the guest run as peacefully as possible.

[2]This is done by setting the bit `CR0.WP` in the hypervisor.

On the hardware side, there are detailed descriptions [195, 196] of the translation from the guest virtual to guest physical to host physical addresses. However, hardly no further detail can be found on how physical processor, more precisely, the physical MMU traverses both kinds of page tables. This is understandable, since few users or researchers need to deal with it at this level within the framework of the current hardware. From the fact that the shadow page tables are traversed by the physical MMU when the guest is in *shadow paging*, it can be deduced that when the *nested paging* is used, the physical MMU must switch between the guest and the nested page tables. For each step of the mapping, when a guest physical address is obtained, it is used to produce the mapping in host physical address space by either referring to the TLB or traversing the second-dimensional page tables if TLB misses. The complete procedure and its cost for a GVA→HPA translation by traversing the page tables are summarized in Table 4.1.

Table 4.1: A complete procedure and the cost for a GVA→HPA translation

	In TDP Tables	In Guest Page Tables
GPT Root	4	
PML4E	4	1
PDPTE	4	1
PDE	4	1
PTE	4	1
Ultimate HPA	1	

The translation procedure begins with the translation of the GPT (guest page table) root, and ends with the generation of the ultimate HPA. Except the GPT root, the translation for each of the other levels involves one memory access in the guest page tables and four memory accesses in the TDP tables; therefore, the physical MMU must traverse the guest page tables and TDP tables alternatively, with the `CR3` register, root of MMU, being loaded in the following order: `nCR3`→`nCR3`→`nCR3`→`nCR3`→`gCR3`→`nCR3`→...[3]

4.3.4 Adaptive Hardware MMU

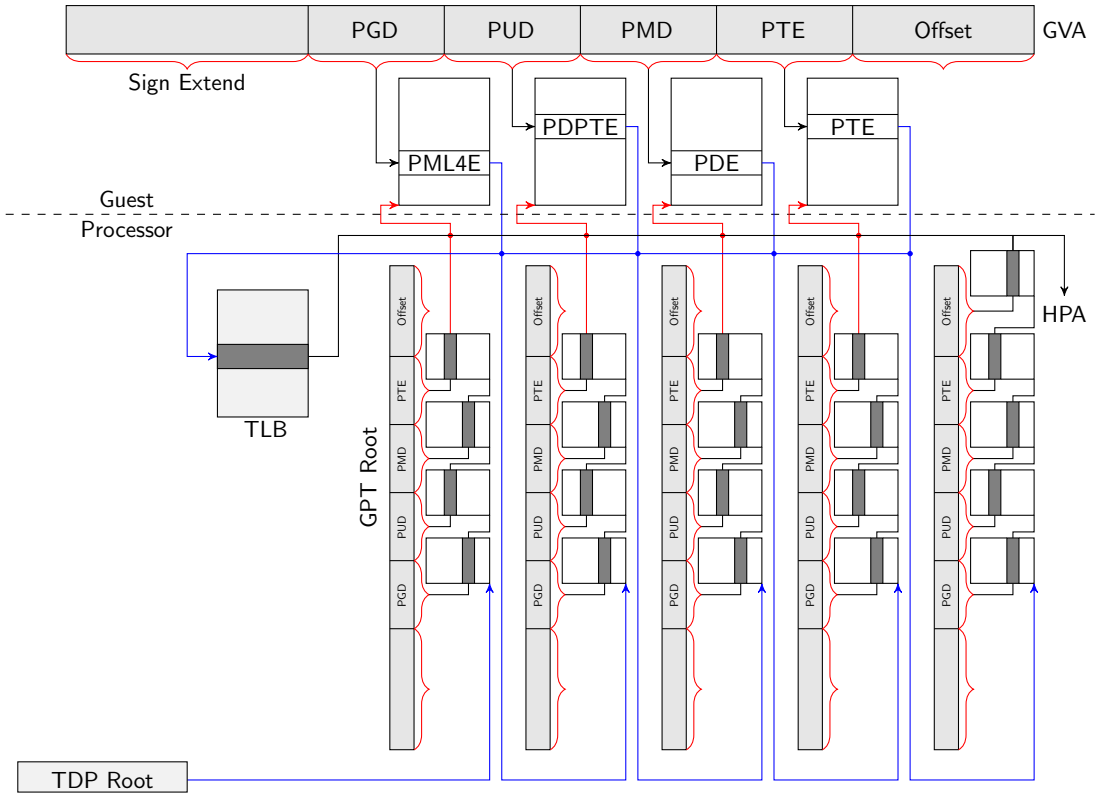
With the conventional scheme, paging keeps consistent in both guest and the second-dimensional page tables. Unfortunately, this holds no longer true in STDP case. The physical MMU is unable to interpret the guest physical address correctly to find the entry of the restructured page table in the second dimension. In other words, the current physical MMU does not support the new paging scheme for TDP. An adaptive MMU with two crucial qualities is needed to do it correctly:

- Interprets a 64-bit GVA as PGD|PUD|PMD|PTE|Offset, but a 64-bit GPA as PHD|PLD|Offset
- Detects the type of traversed page table, and adapts itself to handle it correspondingly

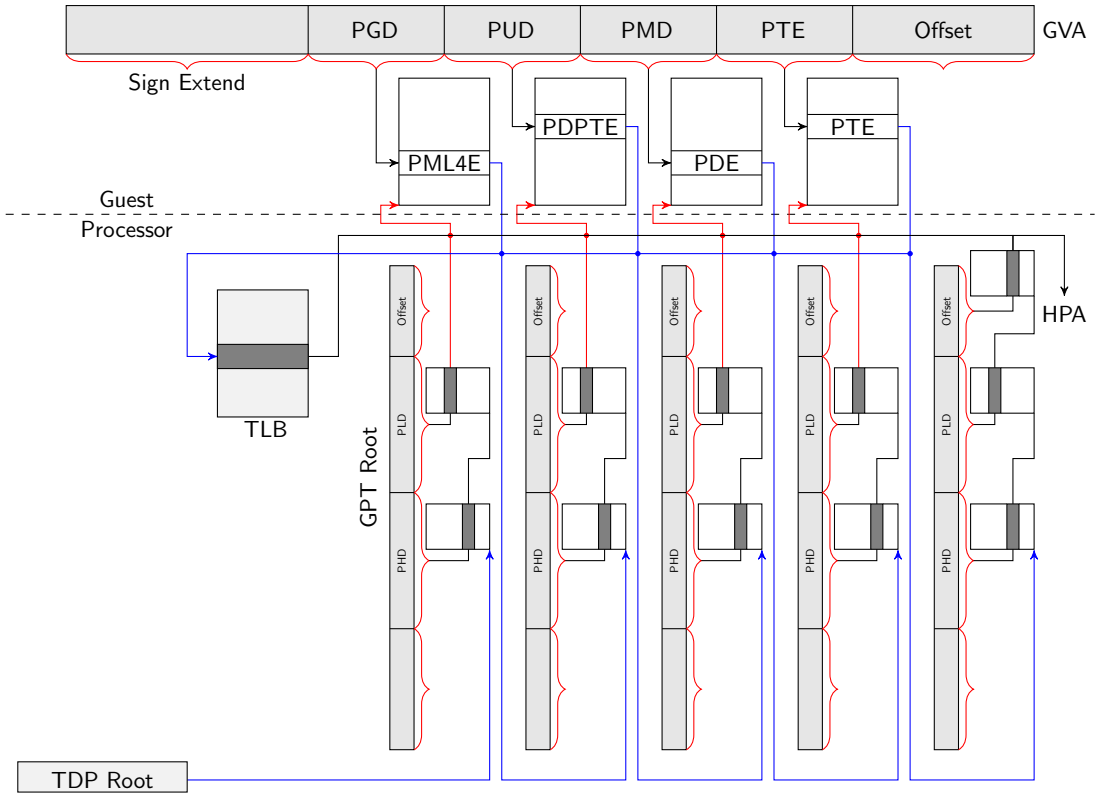
4.4 Summary

For improving the memory virtualization, this chapter has proposed two concepts - DPMS and STDP. DPMS is a pure software solution that applies a strategy to make the hypervisor more adaptive and more intelligent towards the changing workload without changing the paging methods themselves. In contrast, the second solution "STDP" means to achieve the same goal by taking another way – to restructure the page tables used by the *nested paging*, which is more promising among the currently used two standard paging approaches to virtualize the memory in a virtual machine guest. DPMS represents a solutions from a macro perspectives, while STDP represents a solution from a micro perspective. With the proposal of DPMS and STDP, Question 6 in Section 1.4 is answered.

[3] Although `nCR3` belongs to the vocabulary of AMD NPT, for convenience, it is used here in a broader sense to denote the root of the TDP tables, not limited to AMD NPT.



(a) Conventional TDP with four paging levels



(b) Simplified TDP with two paging levels

Figure 4.12 Two TDP schemes

Chapter 5 Implementation

Contents

This chapter presents a concrete implementation of the proposed concepts in the prior chapter. The implementation is based on QEMU-KVM. Section 5.1 provides a general introduction to the components of QEMU and KVM, respectively. In order to provide the necessary information for subsequent implementation, further benchmarks and analysis are launched in Section 5.2.

Section 5.3 presents a detailed analysis on how to implement the components of DPMS and integrate them into the given hypervisor's context. In Section 5.4, discussion is dedicated to the implementation of STDP, focusing mainly on the software rather than hardware side due to the temporary lack of technical support for the latter. The hardware side is described to show that the two parts can work together as a whole. Finally, in Section 5.5, the implementations of DPMS and STDP for QEMU-KVM are summarized.

5.1 QEMU-KVM Hypervisor Analysis

In the enterprise virtualization world today, hypervisors such as the VMware ESX/ESXi Server, Microsoft Hyper-V, Citrix Xen/XenServer and Red Hat QEMU-KVM are undoubtedly the major players. The former two are commercial software that have grown mature over years. The latter two, on the contrary, are relatively young, but free for research and even production. Compared with their pioneers, QEMU and KVM are new to the hypervisor family. However, they are playing increasingly important roles in today's open-source virtualization. The reasons are: 1) KVM is an integral part of the current base Linux kernel and is able to transform the Linux

kernel into a hypervisor without intrusion, which is superior to other standalone hypervisors and has helped it to win more and more popularity among industry; 2) some of the RHEL-based Linux distributions have already included QEMU and `libvirt` as standard packages released for system virtualization. Therefore, the combination of QEMU and KVM is chosen as the platform for testing innovative ideas. Let's have a look inside the QEMU-KVM hypervisor.

5.1.1 About QEMU

As far as computer system emulation and virtualization are concerned, probably no other software is more influential than QEMU. It is a versatile open source machine emulator and virtualizer [201] created by Fabrice Bellard in 2003. The capability to virtualize the entire computer system merely by means of software distinguishes QEMU from the typical hypervisors for system virtualization. Based on the dynamic binary translation, QEMU allows an OS to run without any modification. Furthermore, an excellent feature is its ability to emulate dozens of processor architectures and various peripheral devices, which makes QEMU quite useful for acquiring the desired software development environment in the absence of the target hardware. Over a decade's of development and testing has forged QEMU a mature and valuable software in dealing with processor and device emulation. Therefore, some other hypervisor developers simply take advantage of this convenience by reusing more or less emulation function of QEMU and integrating it into their own code base. In this sense, the hardware emulation technology built in QEMU has a profound influence to the whole open-source virtualization.

QEMU grew out of a PC emulator and ran initially by using the dynamic binary translation as its unique "accelerator"[1]. Emulated platforms can be created to suite the need of an operating system or the user applications compiled for a processor architecture other than the underlying host has. As aforementioned, emulation offers more flexibility at the cost of performance. Unless hardware emulation is really needed, more efficient ways exist for running the guest, especially when the guest and host are targeted at the same processor architecture. In this case, QEMU is ideally used to emulate only a part of the system, I/O devices, while leaving the processors to be managed by a hypervisor. It is more or less reused by KVM, Xen and VirtualBox. For QEMU, these are more efficient alternative accelerators that take advantage of the hardware-assisted virtualization technology. For one of these hypervisors, QEMU serves as the I/O device emulator. In addition to be a hardware emulator, QEMU also provides user-space API virtualization, which enables an application to run on an OS with different ABI other than the one it has targeted at compiling time. By emulating the ABI layer, the system calls applications needed during the run-time, an execution environment is created for a user-space application.

Device Model

QEMU's talent roots in its multi-functional design. While not being written in an object-oriented programming language, QEMU still exhibited much flavor of the object-oriented design patterns. Dozens of processor architectures, countless devices, and the objects derived from them, the large set of entities and their entangled relationships had potentially complicated the implementation and maintenance of this software. A key issue was how to represent, organize and manage these entities efficiently. Unfortunately, things were not in a satisfying condition at the beginning. As Figure 5.1 depicts, prior to v0.11, no device model had ever been adopted. Devices behaved as ad-hoc[2], which means that numerous devices are managed in numerous approaches. It could be a formidable task to deal with a large number of devices. A turning point was reached in 2009 with the turnout of version 0.11, which embraced a new device model - QDev, aiming to unify and simplify the representation of numerous different devices. QDev straightens out QEMU by abstracting and organizing various devices as a hierarchical tree structure with a single-root.

[1] In the context of QEMU, "accelerator" is used, referring to the mechanism with which the guest code is executed
[2] ad-hoc means a solution designed for specific purpose, not intended to be adaptable to other purposes

However, it is not enough in the sense of a well-designed software. QDev underwent a series of further transformations and in 2012 led to QOM - an almost new device model based on QDev.

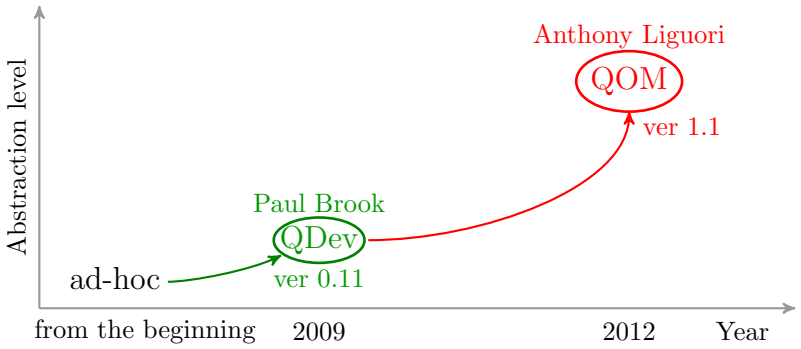


Figure 5.1 Evolution of the QEMU Device Model [197]

As listed in Table 5.1, QDev and QOM are similar in that both are object-oriented, and present unified interfaces to the users. However, more sense lies in their differences. As the two kinds of entity - device and bus are adopted in QDev, the major drawback is the confusion created by the entangled relationship between them. Consequently, to avoid the *multiple inheritance* problem, the rule "each device should have a DeviceState associated with it" does not hold true if a device includes a separate QDev representation referring to a non-DeviceState object [198].

These downsides can be worked around by relying on the "has-a" relationship instead of the "is-a" in inheritance. Guided by this, a multitude of changes [198] were applied to QDev and later leads to QOM, in which the concept "ObjectClass" is exclusively treated as the origin for most of the entities in the system. Since both device and bus share the same origin and uniform representation in QOM, the notion of device-bus connection is replaced by a variety of device-device connections. From ad-hoc to QDEV to QOM, each step involves a level of abstraction and generalization, with the goal to forge the APIs as general as possible to handle a wide range of devices and configurations. Based on a more fully object-oriented model, the QOM, QEMU acquired further capability of taking advantage of the object-oriented features, such as *inheritance* (single inheritance plus interfaces), *class-based polymorphic objects*, *prototype-based polymorphic* properties, *object enumeration*, and the *factory model* for object creation [199].

Figure 5.2 provides an overview of the hierarchy formed by the entities in QOM. Most of these entities are organized according to the nature or behavior into different branches as a tree structure. Except the one at the top and those at the end of each branch, all other entities have a parent, brothers and children. Once a device is implemented, it automatically inherits all the properties from its parent. The siblings share but can override these properties to suit their own needs. In addition, a few entities in QEMU do not share ObjectClass as their ancestor and are out of the tree, either because they are by nature not devices, or merely for auxiliary purposes. QemuOpt, Visitor, TypeInfo and Property are such examples.

Undoubtedly, QOM has delivered a fundamental impact on the QEMU's software architecture. An immediate result is that the notion "Device" becomes a dominant factor in many aspects of QEMU. For example, an important task of a hardware emulator is to create, register and initialize the emulated devices. The common interface in QOM performs this quickly and neatly. QEMU devices are divided into four major types - BLOCK, MACHINE[3], QAPI[4] and QOM, corresponding to the block device, the main board, API for QMP (QEMU Machine Protocol) and the QOM device, respectively. A hash table is used to sort the registered devices. As Figure 5.3 illustrates, the hash table - `init_type_list` is a global variable, with double linked nodes, where the essential thing for devices - callback functions are preserved. At the bottom of the call

[3]Conceptually has been replaced by OPT in the latest QEMU versions, but is still used here for clarity.
[4]It is actually not registered in the context of current QEMU.

Table 5.1: A brief comparison of the QDev and QOM [200, 201, 202]

Aspects	QDev	QOM
Device Relationship	entity: device, bus; represented by DeviceState, BusState; the two share nothing in common; device has properties, bus has no; device-bus-device relation; device has 0 or more buses	entity: device; represented by ObjectClass and Object; share Object as the common thing; device has properties; two forms of relationship: device composition, device backlinks; one device composition, zero or more backlinks
Namespace	three: device, bus, and property namespaces	two: device and property namespaces
Hierarchical Form	multiple devices for a child bus; all devices share a single parent bus; all buses share a single parent device; a strict tree with its level alternatively for bus and device levels; root: SysBus	no explicit notion of parents; bus may be a backlink to the child device; child device may have a backlink to a bus; with device backlink, device composition forms multiple directed graphs, or else a multi-rooted strict tree
Device Namespace	contain names of QDev; allow device but no bus to be anonymous; property namespace local to device; not refer to any child devices	each device has a unique name; bus treated as a device; device name independent of pathnames, can be an opaque blob; relationship treated as named a property
Device Properties	bound to classes and map directly to elements of the device structure; strongly typed; set and parsed from a string type; settable only during construction; read-only afterwards; can not be hooked by using set or get methods	bound to devices; implemented by closures provided by device; accessed and modified by using Visitor through native C type variables; created without properties; set and get after initialization; support realize/unrealize; properties locked when initialized

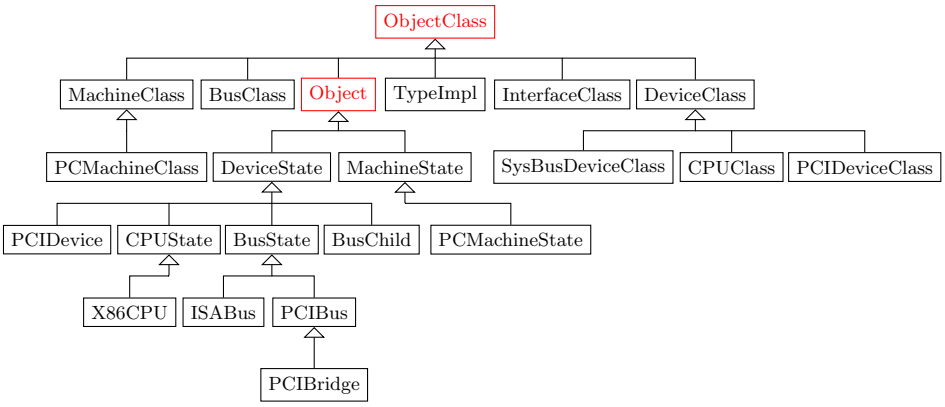


Figure 5.2 Hierarchy formed by the entities in QOM

stack, devices are all managed by manipulating the nodes within the hash table - inserting nodes upon registration, removing nodes upon de-registration, and traversing the lists while searching.

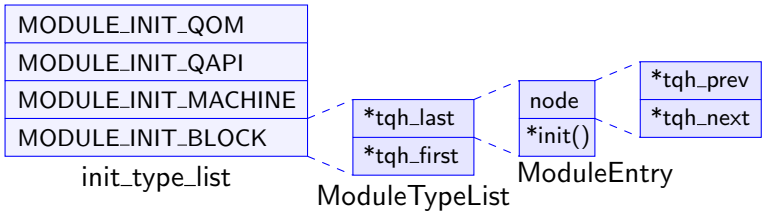


Figure 5.3 Hash table used for managing the emulated devices

Since each of the four types represents a particular kind of devices, QEMU has employed multiple levels of wrapper, with whose uses detail is gradually concealed and common properties are extracted for a set of unified interfaces. These include mainly a chain of macros and functions, from the most general form at the bottom to the most concrete forms at the top of the call stack. Listing 1 depicts the definitions of the most elementary operations for device management and implies the chain formed by these operations. As soon as macro is concerned, a particular one is the `module_init`, which is a wrapper of `register_module_init` modified by the GCC attribute

- `constructor`. In GCC, the use of `constructor` and `destructor` attributes assigns a higher priority to the functions (initializer and finalizer) they modified to control the order of execution relative to the `main()`. In this case, it ensures an execution of the macro `module_init` before `main()` of QEMU, in order to register the devices and create the necessary emulated hardware, such as the virtual processors in the context of QEMU. Devices are registered by filling the callbacks into corresponding nodes of the hash table, and initialized by executing those callbacks.

Listing 1 Common interfaces for device registration and initialization

```
static void init_types(void)
{
    static int inited;
    int i;

    if (inited) {
        return;
    }

    for (i = 0; i < MODULE_INIT_MAX; i++) {
        QTAILQ_INIT(&init_type_list[i]);
    }

    inited = 1;
}

static ModuleTypeList *find_type(module_init_type
                                 type)
{
    ModuleTypeList *l;

    init_types();
    l = &init_type_list[type];
    return l;
}
```

```
void register_module_init(void (*fn)(void),
                          module_init_type type)
{
    ModuleEntry *e;
    ModuleTypeList *l;

    e = g_malloc0(sizeof(*e));
    e->init = fn;

    l = find_type(type);

    QTAILQ_INSERT_TAIL(l, e, node);
}

void module_call_init(module_init_type type)
{
    ModuleTypeList *l;
    ModuleEntry *e;

    l = find_type(type);

    QTAILQ_FOREACH(e, l, node) {
        e->init();
    }
}
```

```
/* This should not be used directly.  Use block_init etc. instead.  */
#define module_init(function, type)                                         \
static void __attribute__((constructor)) do_qemu_init_ ## function(void) {  \
    register_module_init(function, type);                                   \
}

#define block_init(function) module_init(function, MODULE_INIT_BLOCK)
#define machine_init(function) module_init(function, MODULE_INIT_MACHINE)
#define qapi_init(function) module_init(function, MODULE_INIT_QAPI)
#define type_init(function) module_init(function, MODULE_INIT_QOM)
```

Main Components

QEMU maintains a large code base to emulate a wide range of peripheral devices and dozens of ISAs. Furthermore, the code base keeps on growing as new features are added for more flexibility and portability. Nevertheless, the main components still remain relatively stable, which include:

- TCG (tiny code generator): TCG is the central component for emulating processors of different ISAs. It is actually a combination of de-compiler and compiler. First, the de-compiler extracts the semantics from the binary code of source ISA and represents them by means of intermediate code. The compiler generates the binary code for the target ISA from the intermediate code.
- Soft-MMU (Software-controlled MMU): To the guest, Soft-MMU serves as an emulated (physical) hardware MMU that performs the GVA→HVA mapping. Furthermore, it also maps the guest virtual address to the registered I/O device callbacks.
- Device Emulation: It does not refer to any individual component, but designates a large set of the commonly adopted emulation units in QEMU, from PS/2 mouse and keyboard to VGA card, from the USB hub, USB controller to PCI bridge, serial ports, and the entire chip-sets on the main board.

5.1.2 About KVM

KVM, the kernel-based virtual machine, is a software component providing system virtualization function based on the Linux kernel. Unlike a full-fledged hypervisor, KVM is merely an extension to the functionality of the conventional Linux kernel. It exists as a module that is dynamically loadable into the execution context created by other modules and the code permanently compiled into the kernel. In such a manner, while endowed the Linux kernel the capability of running as a hypervisor, KVM maintains its relative independence from the rest parts of the kernel, which is beneficial not only for keeping the hypervisor simpler and more reliable, but also for achieving more flexibility in code reuse and maintenance. As a result, the Linux kernel infrastructure can be utilized not only as the basis of a general-purpose operating system for user applications, but also as the major component of a hypervisor for virtual machine guests.

Although hypervisor and operating system are different kinds of system software, they share some similarity in managing the system resources, and no clear demarcation exists between them. Actually, hypervisor is a special-purpose operating system, which services virtual machine guests instead of normal tasks. For this reason, it may save huge efforts if the components of a mature operating system can be reused in an appropriated form for virtualization. KVM was created with this motivation. Being backed by the Linux kernel, KVM has implemented only a number of core functions, and fulfills its duty by cooperating with the rest of the kernel. Table 5.2 serves as a brief summary of the major functions implemented by KVM.

Table 5.2: Major Functions implemented by KVM

Function	Related Source File	Brief Description
KVM system driver	kvm_main.c	open, control and close the device `/dev/kvm`
VM guest driver	x86.c	APIs for operating the virtualized hardware of a guest
VCPU driver	x86.c, svm.c, vmx.c	APIs for operating the virtualized processor of a guest, including the vendor-specific VT-hardware part
CPUID emulation	cpuid.c, cpuid.h	emualte the CPUID function for VCPU
Virtual MMU	mmu.c, mmu.h, paging_tmpl.h, kvm_cache_regs.h coalesced_mmio.c coalesced_mmio.h async_pf.c, async_pf.h	support for GVA->HPA and I/O device memory mapping, including a few solutions for performance, such as interrupt coalescing and asynchronous page fault handling
Instruction emulation	emulate.c	emulate a number of particular instructions that cannot be natively executed, mainly for the MMIO-related instructions
Timer emulation	i8254.c	emulate the interval timer chip - Intel 8254 PIT
Interrupt controller emulation	i8259.c, ioapic.c, lapic.c, irq_comm.c	emulate a variety of the interrupt controllers, such as the Intel 8259 PIC, APIC, IO-APIC and the local APIC
Device pass-through	assigned-dev.c, iommu.c, vfio.c, vfio.h	support for passing I/O devices from host to guest, mainly dealing with interrupts (INTX, IRQ, MSI, MSI-X) and the I/O-deviced mapped memory
PMU emulation	pmu.c, pmu.h pmu_amd.c pmu_intel.c	support for Performance Monitoring Unit in VCPU, including the vendor-specific features
I/O event message	eventfd.c, irq.c, irq.h, irqchip	support for sending and receiving messages to and from the host kernel space

Guest Creation and Execution

Compared with other standalone hypervisors, such as Xen, VMware ESXi Server, or VirtualBox, KVM is tiny. In addition to the functions listed above, the remaining functionality is provided by Linux kernel and a modified version of QEMU[5]. As an open-source software, QEMU was adopted to support KVM mainly for two purposes. First of all, it provided a user interface that is able to communicate with the KVM driver in the kernel-space and control the execution of the virtual machine guests by sending corresponding commands to a virtual device, `/dev/kvm` by means of `ioctl`. Second, the huge power to emulate a wide range of peripheral devices happens to be the

[5]Since version 1.3.0, the previous QEMU branch for KVM is completely merged into the QEMU code base.

very thing KVM wanted. Figure 5.4 depicts the architecture of the whole hypervisor composed by QEMU and the Linux kernel with KVM module. QEMU is started either by a command-line or graphical user interface as a normal user application. Meanwhile the configuration parameters of a guest are also passed to QEMU. As soon as QEMU enters into the `main()`, the parameters are recognized and sorted by the command parser. The devices which had been registered in the context of `constructors` are initialized by using these parameters.

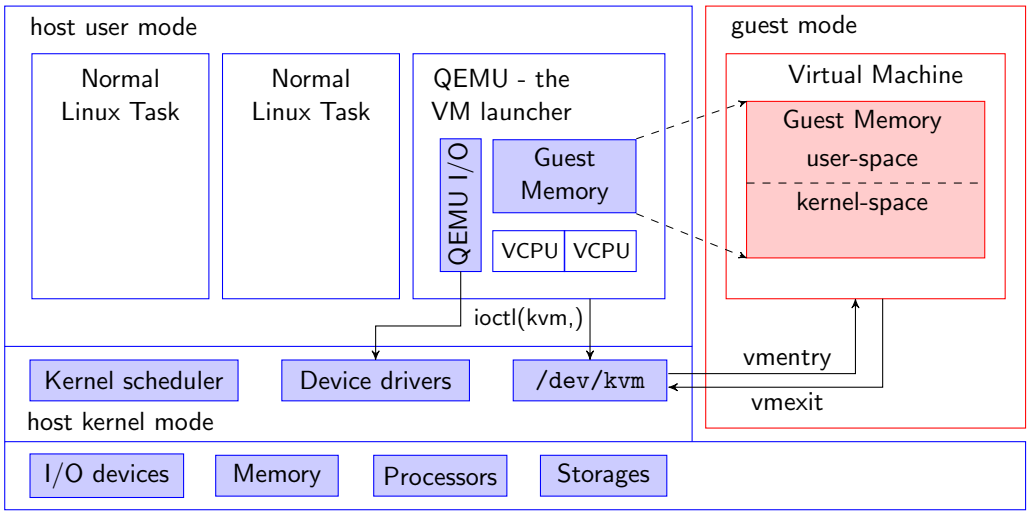


Figure 5.4 QEMU and KVM as a whole hypervisor

After a series of well designed preparation, the execution enters into the `main_loop()`. If the essential thing is extracted from the lengthy details, the skeleton can be depicted by Listing 2.

Listing 2 Skeleton for the guest execution

```
int main()
{
        open("/dev/kvm")
        ioctl(KVM_CREATE_VM)
        ioctl(KVM_CREATE_VCPU)
        for (;;) {
                ioctl(KVM_RUN)
                switch (exit_reason) {
                case KVM_EXIT_IO: /* ... */
                case KVM_EXIT_HLT: /* ... */
                ...
        }
}
```

5.2 Parameter Study for DPMS

Section 4.2 has presented the concepts to design the functional units of DPMS from a hypervisor-independent point of view. As the two units - *Data Processing* and *Decision Making* are more dependent on the choice of input data, a question must be answered before the implementation of them, namely, how is the optimal paging method correlated with the performance data collected from the PMCs? By analyzing the run-time performance data, this section attempts to answer this key question, and make preparation for implementing the whole design of DPMS.

The analysis is practically to mimic the hypervisor's possible reaction to the sampled data. Instead of being fed into DPMS for decision making, the data is redirected to a syslog file in the Linux kernel (`/var/log/syslog`). For each tested workload, data items specified in Section 4.2.1 are logged in this approach, with a sampling frequency of 1 Hz. Table 5.3 lists some of the performance metrics calculated from these raw data from a global view, including: 1) IPC, 2) PF, 3) `vmexit`, 4) PFR, 5) TLBM, 6) TMR, as well as 7) ratio of TLBM for NPT and SPT, 8) ratio of TMR for NPT and SPT. Among these, PF and `vmexit` are the average occurrences per second, IPC, PFR, TLBM and TMR are the global mean values in the statistical sense, which are different from the corresponding instant values calculated on the fly.

In addition to the global statistic view, the data are also analyzed from a real-time perspective. The variation of instant IPC, PFR, and TMR values during the execution of each workload are plotted and smoothed in Figure 5.6 5.7, 5.8 and 5.9 by applying the approximate mean value [6].

For the sake of observation, the workloads in Table 5.3 are marked by different colors according to their nature regarding the paging method. From the results exhibited in Figure 3.6a and 3.6b, `dedup`, `vips` and `fft` are marked as TDP-inclined, while `barnes` as SPT-inclined for Platform 1 (P1); `dedup`, `streamcluster`, `vips`, `x264` and `water_nsquared` are marked as TDP-inclined, while `fft`, `lu_ncb`, `radix` as SPT-inclined for Platform 2 (P2).

By comparing the corresponding statistics between the *nested paging* and *shadow paging*, it is easy to identify some of the obvious correlations between the performance data and the applied paging method.

First, the (global and instant) IPC can really reflect the execution speed of a workload, which proved true for the majority of the PM-sensitive workloads. However, for many other workloads, this may not be the case. Examples are `facesim`, `freqmine`, `lu_cb`, `lu_ncb` and `radiosity` for P1, and `blackscholes`, `bodytrack`, `canneal`, `facesim`, `ferret`, `freqmine`, `lu_cb`, `lu_ncb` and `radiosity` for P2. These illustrate that those workloads which differ significantly in global IPC values may yield quite similar performance in the *shadow paging* and *nested paging*. Considering the possible negative impact on the reliability of this metric due to so many exceptions, IPC is excluded as a factor for decision making.

Second, most of the TDP-inclined workloads exhibit durable high PFR in SPT. Examples are `bodytrack` (P2: 44.76%), `dedup` (P1: 29.89%, P2: 47.51%), `vips` (P1: 22.31%, P2: 47.10%), `fft` (P1: 52.49%). However, the root cause lies on the PF and vmexit, which spike to extraordinary high level – one or two orders of magnitude higher than average values.

Figure 5.5a and 5.5b compared the occurrences of page fault and vmexit for all the workloads. Both platforms showed significantly huge figures for the TDP-inclined workloads than the others. It coincides with the results of the performance benchmark, and clearly revealed the dominant causes for the heavier performance loss for *shadow paging* and *nested paging*. On the other hand, PFR - which has ever been adopted as a major condition for paging method switching cannot identify this characteristic, as the corresponding values in Table 5.3 reflect.

[6] To better illustrate the different behaviors under TDP and SPT, the subfigures in Figure 5.6, 5.7, 5.8 and 5.9 are arranged compactly, which makes the legends too small to read. For this reason, they are specified here. In Figure 5.6 and 5.7, red: Instruction per Cycle, blue: approximated Instruction per Cycle, black: Page fault *vmexit* rate, purple: approximated Page fault *vmexit* rate. In Figure 5.8 and 5.9, red: TLB miss rate, blue: approximated TLB miss rate.

This fact suggests that higher occurrences of page faults and vmexit are sufficient and necessary conditions for inferring a workload to be TDP-inclined. They are more effective than PFR in identifying the TDP-inclined workloads. Furthermore, the adoption of them also helps the hypervisor to avoid division operations in kernel space.

Another aspect is to determine the conditions for switching to the *shadow paging*. Nevertheless, the statistics provide quite limited clue to how the performance of a SPT-inclined workload is influenced by the corresponding performance metrics. Among all items, only TLBM and TMR have established some positive connections with the relative lower performance for `fft` on P2 with the *nested paging*.

The *nested paging* and *shadow paging* are independent solutions. Although their performance can be compared each other, they are irrelevant. Low performance yielded by one of them does not necessarily indicate high performance yielded by its alternative. However, if the workload hits the weakness of a paging method, a switching to its alternative may bring benefits by avoiding such a case. Figure 5.8 and 5.9 illustrate the TLB miss rates under the *nested paging* for each workload and each platform. The sporadic zigzags are smoothed by an approximated curve to show the overall trend of TMR. The figures revealed at least the following facts:

- For most of the workloads, TMR falls in a range of $[10^{-6}, 10^{-3}]$, with only a few exceptions out of this region (above 10^{-3} normally at the beginning of execution).
- A number of workloads yield TMR significantly lower than 10^{-3} during the majority of runtime, such as `facesim` (P1: 10^{-4}, P2: 10^{-4}), `volrend` (P1: 10^{-6}, P2: 10^{-5}), `raytrace` (P1: 10^{-4}), `freqmine` (P2: 10^{-5}), `water_nsquared` (P1: 10^{-5}, P2: 10^{-6}).
- The TMR exhibited by the SPT-inclined workloads is a little confused. For example, `barnes` (P1: around 10^{-3}), `radix` (P2: around 10^{-3}), `fft` (P2: within $[10^{-5}, 10^{-4}]$), and `lu_ncb` (P2: within $[10^{-4}, 10^{-3}]$).

These facts imply that higher TMR is neither a sufficient nor a necessary condition for inferring that a workload is SPT-inclined. However, in spite of such ambiguity of TMR, one thing is clear – workloads yielding TMRs less than a magnitude of 10^{-5} suffer less likely large performance loss with *nested paging*. Examples are `volrend` (P1: 10^{-6}, P2: 10^{-5}), `ferret` (P1: 10^{-5}), `freqmine` (P2: 10^{-5}), and `water_nsquared` (P1: 10^{-5}, P2: 10^{-6}).

On the other hand, workloads yielding TMRs higher than a magnitude of 10^{-5} are quite likely to suffer more performance loss with the *nested paging* than the *shadow paging*. Meanwhile, it is noticable that both page fault and *vmexit* for the SPT-inclined workloads occur less frequently with the *nested paging*. Examples are: `barnes` on P1 (PF: 16.08, ranking 12/25, vmexit: 876.74, ranking 10/25), `fft` on P2 (PF: 0.09, ranking 7/25, vmexit: 22.42, ranking 10/25), `lu_ncb` on P2 (PF: 0.01, ranking 2/25, vmexit: 25.52, ranking 5/25), `radix` on P2 (PF: 0.09, ranking 7/25, vmexit: 75.77, ranking 11/25). All the figures are at least one magnitude lower than those yielded by workloads which proved not SPT-inclined.

As a summary of the discussion, the rules for different types of workloads are the following:

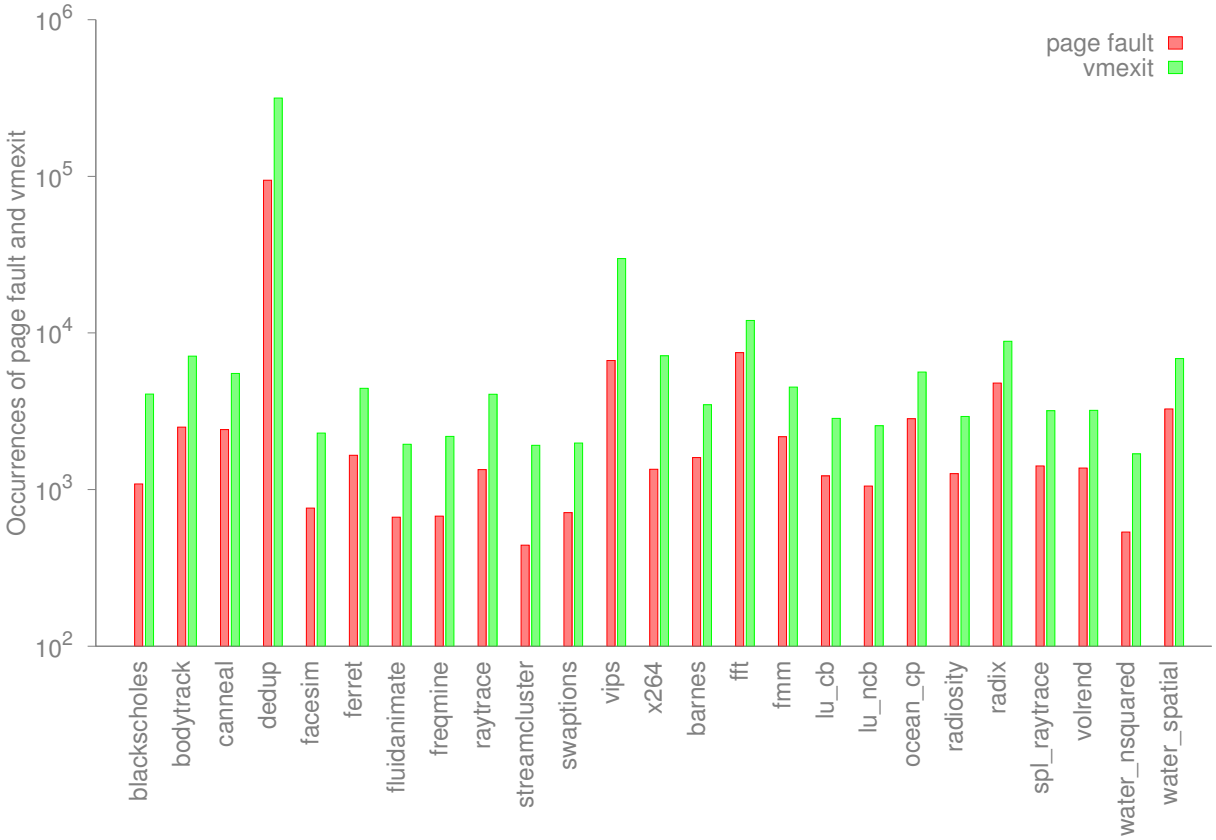
For SPT-to-TDP Switching: For a switching from the *shadow paging* to the *nested paging*, PF and vmexit are much relevant. When the occurrences of durable large values of the two are observed, the *nested paging* should be switched to. In this case, if TMR grows rapidly, the *shadow paging* can be switched back.

For TDP-to-SPT Switching For a switching from the *nested paging* to the *shadow paging*, TMR, PF and vmexit are much relevant. When the occurrence of durable large values of TMR, and meanwhile low occurrences of PF and vmexit are observed, the paging should be performed by *shadow paging*. In this case, if PF and vmexit spike to a significantly high level, the *nested paging* can be switched back.

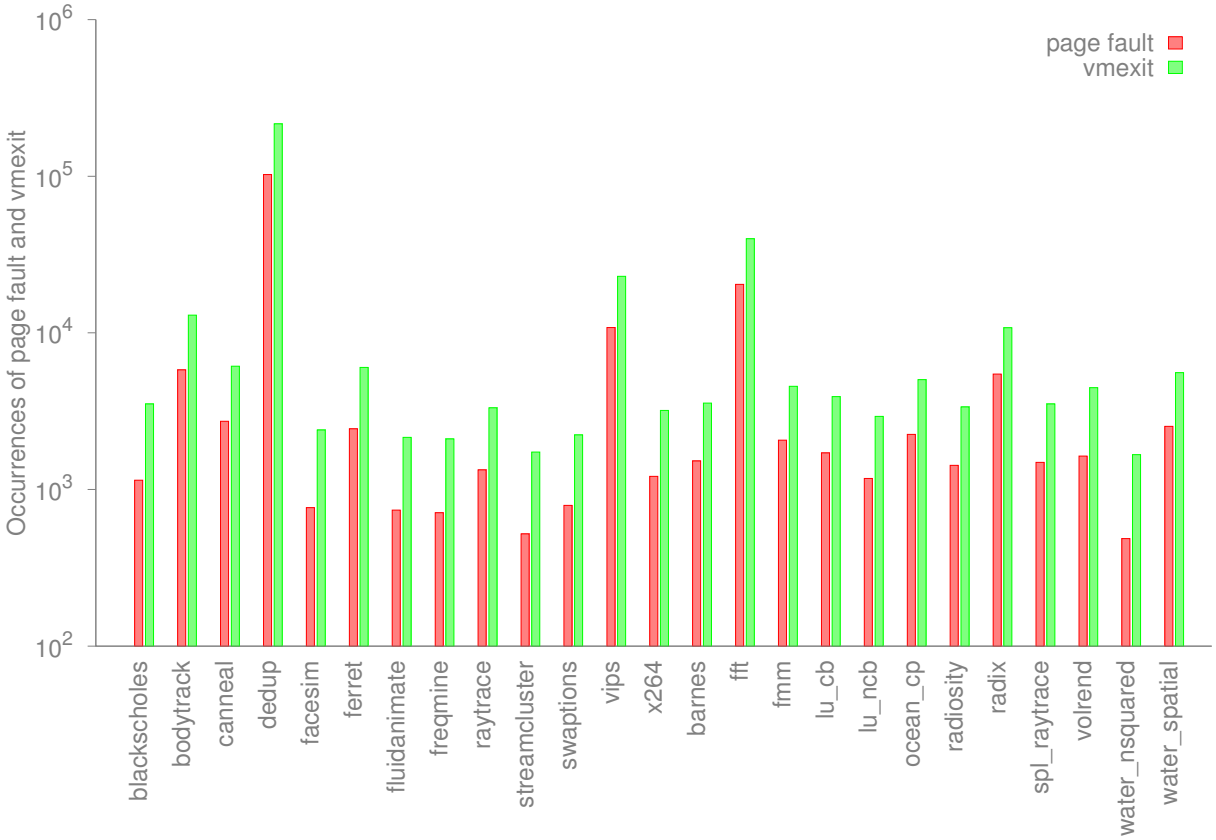
For No-Switching Among others, the conditions for switching are not triggered, therefore, the paging method can remain as it is.

Table 5.3: Statistics of PARSEC-3.0 workloads on P1 and P2

workload	*IPC*		*PF*		`vmexit`		*PFR*		*TLBM* (*TLBmiss*)		*TMR*		$TLBM_{NPT}/TLBM_{SPT}$	TMR_{NPT}/TMR_{SPT}	
	NPT	*SPT*	*NPT*	*SPT*	*NPT*	*SPT*	*NPT*	*SPT*	*NPT*	*SPT*	*NPT*	*SPT*			
blackscholes	1.2500	1.5835	8.50	1087.29	2288.20	4071.93	3.71E-003	2.67E-001	6166.30	688.02	7.47E-004	1.24E-004	8.9624	6.0039	P1
bodytrack	1.0874	1.4365	207.05	2504.67	2778.02	7106.47	7.45E-002	3.52E-001	43033.21	1634.59	1.31E-003	2.15E-004	26.3266	6.0882	
canneal	2.0393	1.9539	120.45	2414.12	1192.00	5516.21	1.01E-001	4.38E-001	815.11	747.80	5.57E-005	6.28E-005	1.0900	0.8863	
dedup	0.9789	1.9603	119.06	94483.95	7708.09	316070.81	1.54E-002	2.99E-001	39323.06	2072.33	1.57E-003	1.24E-004	18.9753	12.6574	
facesim	2.5873	1.5113	9.25	761.91	973.41	2294.51	9.50E-003	3.32E-001	1273274.68	15209.09	1.27E-004	6.62E-004	83.7180	0.1911	
ferret	1.9173	1.6178	17.79	1657.89	11005.79	4427.99	1.62E-003	3.74E-001	1704.84	62652.04	6.46E-005	9.86E-006	0.0272	6.5551	
fluidanimate	2.9258	1.5685	92.00	665.98	933.18	1945.52	9.86E-002	3.42E-001	733.64	16020.30	2.67E-005	5.51E-004	0.0458	0.0485	
freqmine	2.6648	2.2757	66.74	677.18	1088.51	2187.12	6.13E-002	3.10E-001	759.31	1003.78	3.24E-005	4.42E-005	0.7564	0.7329	
raytrace	2.0444	2.0738	34.13	1342.81	1683.61	4062.18	2.03E-002	3.31E-001	768.82	1637.32	3.01E-005	8.15E-005	0.4696	0.3693	
streamcluster	1.7149	2.3677	2.55	442.27	788.56	1916.40	3.24E-003	2.31E-001	373.20	675.35	1.09E-004	3.38E-005	0.5526	3.2151	
swaptions	2.6077	2.6696	5.26	712.58	810.25	1981.40	6.50E-003	3.60E-001	608.40	568.81	3.07E-005	3.90E-005	1.0696	0.7864	
vips	1.2704	1.1830	49.36	6670.15	4177.50	29887.64	1.18E-002	2.23E-001	11196.63	17178.96	4.83E-004	5.34E-004	0.6518	0.9042	
x264	1.1466	1.3297	15.41	1346.77	4530.05	7152.02	3.40E-003	1.88E-001	20356.39	12883.28	5.22E-004	4.86E-004	1.5801	1.0737	
barnes	1.8353	2.5317	16.08	1600.05	876.74	3484.66	1.83E-002	4.59E-001	667.09	872.44	1.04E-004	4.22E-005	0.7646	2.4505	
fft	2.0225	0.8124	744.35	7497.69	1870.89	12002.62	3.98E-001	6.25E-001	877.05	1164.76	3.09E-005	5.62E-004	0.7530	0.0550	
fmm	2.6528	2.4756	13.16	2172.69	869.98	4517.39	1.51E-002	4.81E-001	1061.34	1050.59	6.85E-005	5.79E-005	1.0102	1.1824	
lu_cb	2.3309	1.6129	5.28	1223.48	798.65	2846.59	6.61E-003	4.30E-001	685.09	468.25	4.23E-005	8.34E-005	1.4631	0.5071	
lu_ncb	2.7761	2.4131	9.07	1054.28	807.81	2559.93	1.12E-002	4.12E-001	45328.16	764.41	4.20E-006	3.04E-005	59.2982	0.1380	
ocean_cp	2.3373	0.4267	160.52	2830.78	1190.65	5630.37	1.35E-001	5.03E-001	535.40	487.96	2.46E-005	7.74E-004	1.0972	0.0317	
radiosity	2.8005	1.4625	16.36	1265.84	820.40	2929.69	1.99E-002	4.32E-001	743.08	453.89	2.10E-005	1.34E-004	1.6371	0.1570	
radix	2.7230	1.5770	9.14	4793.37	869.79	8859.44	1.05E-002	5.41E-001	1107.72	826.37	7.03E-005	1.58E-004	1.3405	0.4460	
spl_raytrace	2.5586	1.8051	0.55	1414.07	811.75	3187.39	6.77E-004	4.44E-001	577.73	826.68	5.37E-005	1.30E-004	0.6989	0.4130	
volrend	1.6541	2.1536	8.04	1373.80	970.53	3203.21	8.28E-003	4.29E-001	8897.50	887.90	1.41E-006	7.50E-005	10.0209	0.0188	
water_nsquared	1.9877	2.3973	0.13	536.28	792.37	1690.03	1.64E-004	3.17E-001	704.99	788.06	3.96E-005	4.72E-005	0.8946	0.8381	
water_spatial	2.4730	2.6769	44.60	3273.20	878.76	6863.91	5.08E-002	4.77E-001	817.86	647.91	4.89E-005	4.15E-005	1.2623	1.1783	
blackscholes	1.9309	0.3091	0.38	1145.28	911.38	3519.67	4.18E-004	3.25E-001	6324.64	5437.49	2.74E-005	3.83E-005	1.1632	0.7138	P2
bodytrack	0.9063	0.2716	14.38	5807.38	1413.81	12973.82	1.02E-002	4.48E-001	21024.00	10695.30	3.39E-004	4.32E-005	1.9657	7.8425	
canneal	1.5812	0.2770	11.56	2730.15	844.48	6124.20	1.37E-002	4.46E-001	3182.29	1370.52	8.02E-005	1.11E-005	2.3220	7.2106	
dedup	1.1261	1.0330	82.31	102581.16	1427.80	215897.78	5.77E-002	4.75E-001	11935.96	40536.20	2.09E-004	1.74E-004	0.2945	1.1984	
facesim	0.2271	1.9632	13.27	765.98	964.16	2395.37	1.38E-002	3.20E-001	11164.98	283531.15	2.44E-004	3.86E-005	0.0394	6.3382	
ferret	1.3346	0.5747	16.20	2438.56	1177.99	6013.20	1.37E-002	4.06E-001	48130.89	1176.12	1.44E-005	3.05E-004	40.9233	0.0473	
fluidanimate	2.0261	1.7240	76.84	739.30	900.10	2150.26	8.54E-002	3.44E-001	2662.97	5895.29	1.06E-004	2.23E-006	0.4517	47.7366	
freqmine	1.7229	0.6849	12.23	710.50	846.67	2103.39	1.45E-002	3.38E-001	72625.36	785.59	1.14E-005	1.33E-004	92.4468	0.0859	
raytrace	0.2749	0.6333	11.58	1333.22	1274.26	3320.33	9.09E-003	4.02E-001	7673.23	6501.77	1.42E-004	2.05E-004	1.1802	0.6922	
streamcluster	0.7949	0.5184	0.01	521.12	1415.04	1736.00	7.92E-006	3.00E-001	2882.93	1242.82	3.80E-006	8.66E-005	2.3197	0.0439	
swaptions	1.9764	1.7204	0.12	791.21	1418.63	2233.81	8.45E-005	3.54E-001	1130.28	2724922.20	3.15E-005	9.23E-004	0.0004	0.0341	
vips	0.9843	1.2704	52.68	10810.33	1177.81	22948.19	4.47E-002	4.71E-001	57833.14	36562.28	4.03E-004	1.38E-004	1.5818	2.9196	
x264	0.8887	1.1953	0.43	1212.88	1239.54	3199.00	3.48E-004	3.79E-001	37300.29	21172.45	2.64E-004	9.09E-005	1.7617	2.9013	
barnes	0.3828	1.1502	0.32	1522.42	26.56	3561.79	1.21E-002	4.27E-001	2552.31	1058.75	6.19E-005	6.94E-005	2.4107	0.8918	
fft	1.8127	1.7921	0.09	20390.50	22.42	39939.98	4.15E-003	5.11E-001	6757670.56	2382.00	1.06E-003	9.55E-005	2836.9734	11.0518	
fmm	0.3718	1.2597	0.47	2064.48	51.40	4563.13	9.09E-003	4.52E-001	8810.72	941.35	2.67E-004	3.92E-005	9.3596	6.7955	
lu_cb	0.4338	3.3015	0.04	1710.50	16.97	3922.53	2.20E-003	4.36E-001	1736.74	35056.02	5.76E-005	2.89E-006	0.0495	19.9232	
lu_ncb	0.6798	2.1368	0.01	1176.17	25.52	2928.01	2.48E-004	4.02E-001	1662.24	29205.10	5.26E-005	6.25E-006	0.0569	8.4290	
ocean_cp	0.6756	1.7111	0.00	2246.77	36.01	5030.08	0.00E+000	4.47E-001	8798.00	1383.48	1.10E-003	6.24E-005	6.3593	17.5662	
radiosity	0.3818	1.5106	0.03	1427.40	15.38	3365.93	2.02E-003	4.24E-001	1595.76	71411.49	4.33E-005	1.29E-005	0.0223	3.3567	
radix	1.3580	1.5737	0.09	5457.26	75.77	10786.15	1.23E-003	5.06E-001	10754.33	1126.44	4.08E-004	9.90E-005	9.5472	4.1202	
spl_raytrace	0.4681	0.8401	0.37	1489.15	19.95	3514.36	1.85E-002	4.24E-001	2610.37	1166.89	7.03E-005	2.91E-005	2.2370	2.4145	
volrend	1.4947	1.5036	0.49	1635.59	64.14	4462.41	7.66E-003	3.67E-001	55439.53	29960.15	1.05E-005	5.49E-006	1.8504	1.9091	
water_nsquared	2.0934	2.0839	0.00	485.42	12.77	1667.47	0.00E+000	2.91E-001	17253.87	11572.77	2.21E-006	1.47E-006	1.4909	1.4995	
water_spatial	0.8953	0.5777	0.14	2531.29		5575.65	5.77E-004	4.54E-001	8180.96	23861.49	7.62E-006	4.38E-004	0.3429	0.0174	

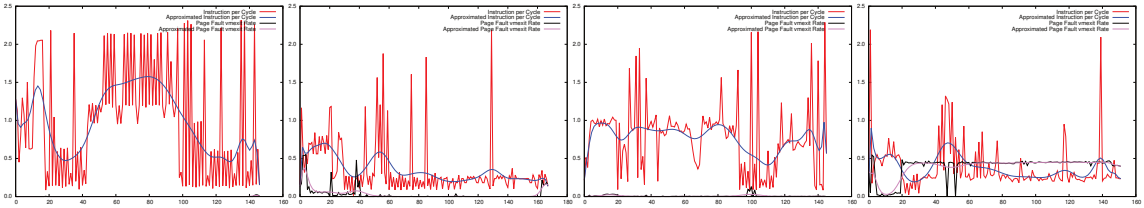


(a)

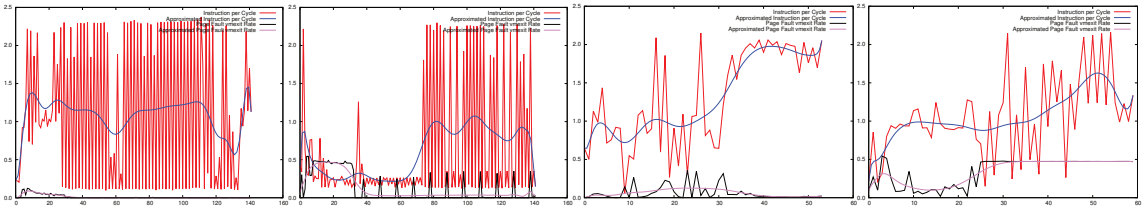


(b)

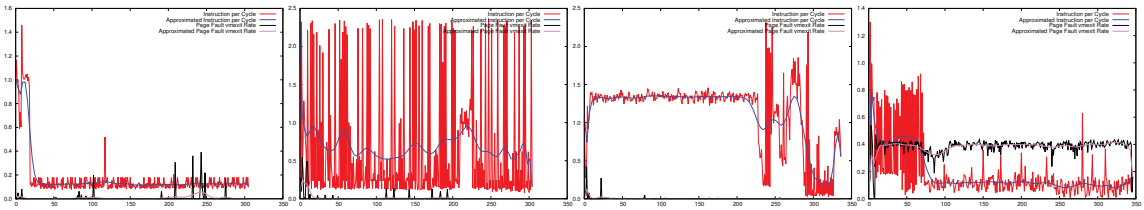
Figure 5.5 Occurrences of page fault and vmexit for PARSEC-3.0 workload on (a) Platform 1 (Intel Core i7-6700K), (b) Platform 2 (Intel Xeon e5-1620-v2)



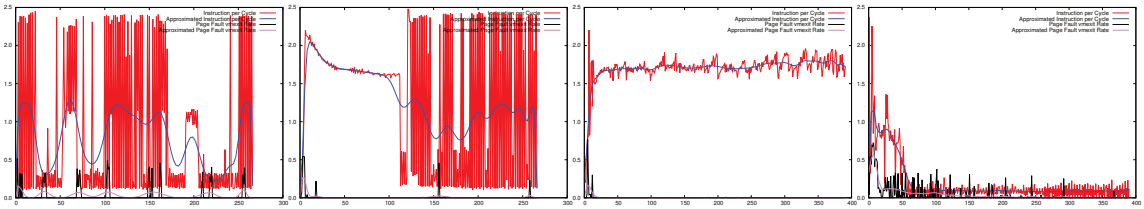
(a) blackscholes in EPT (b) blackscholes in SPT (c) bodytrack in EPT (d) bodytrack in SPT



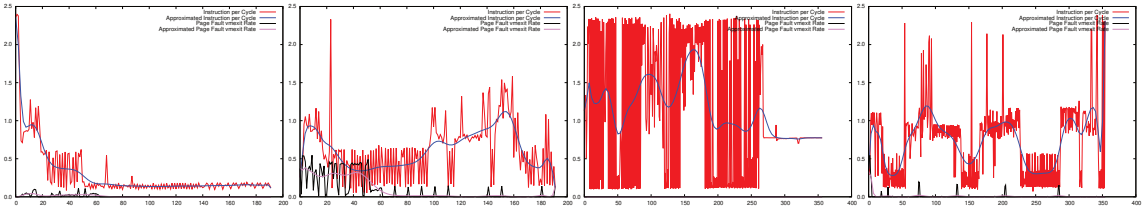
(e) canneal in EPT (f) canneal in SPT (g) dedup in EPT (h) dedup in SPT



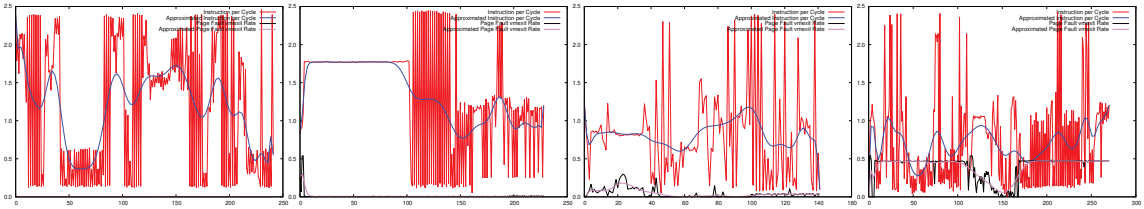
(i) facesim in EPT (j) facesim in SPT (k) ferret in EPT (l) ferret in SPT



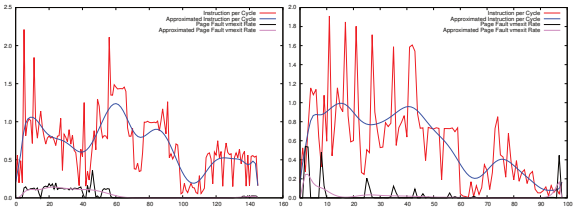
(m) fluidanimate in EPT (n) fluidanimate in SPT (o) freqmine in EPT (p) freqmine in SPT



(q) raytrace in EPT (r) raytrace in SPT (s) streamcluster in EPT (t) streamcluster in SPT

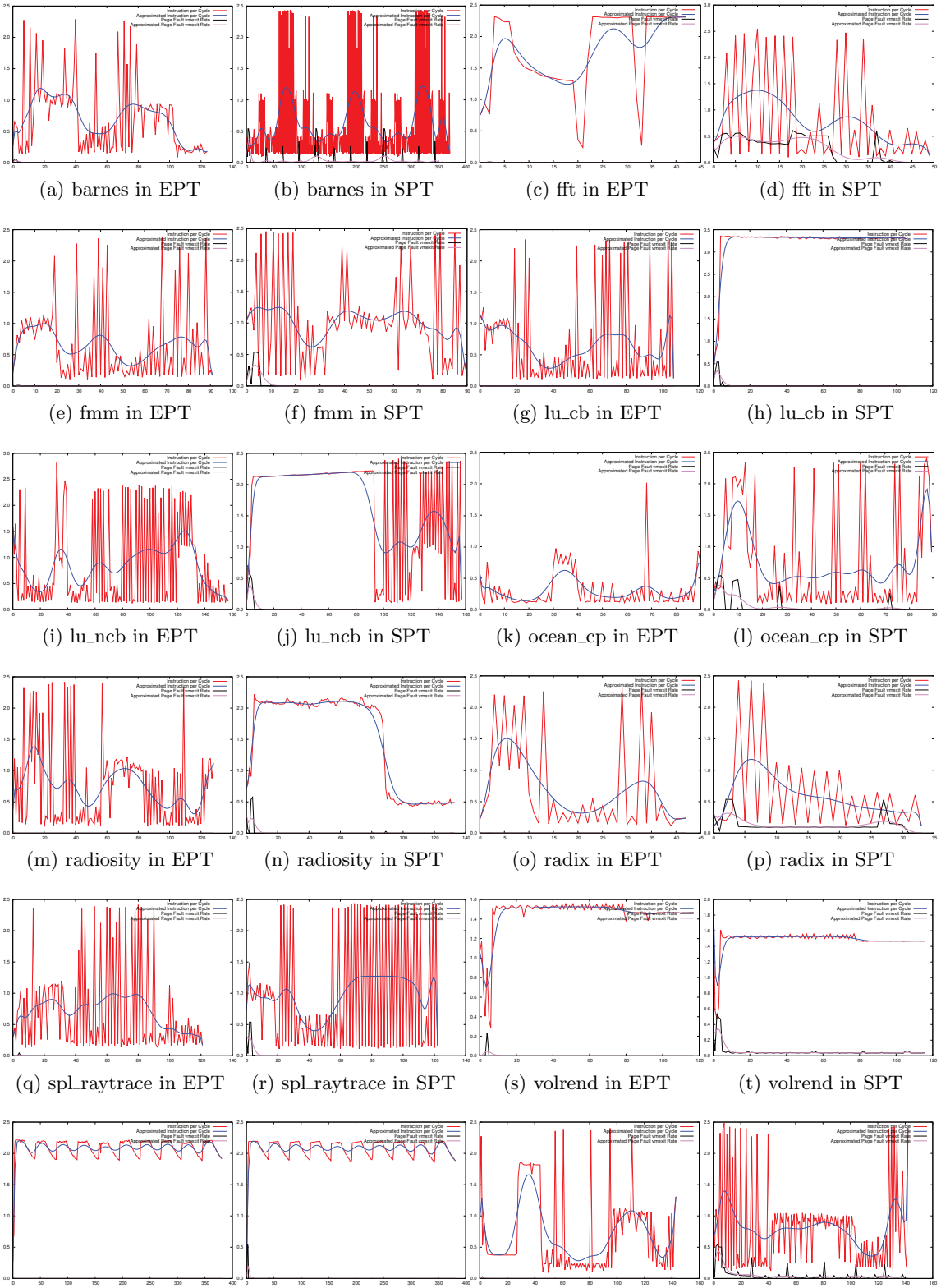


(u) swaptions in EPT (v) swaptions in SPT (w) vips in EPT (x) vips in SPT



(y) x264 in EPT (z) x264 in SPT

Figure 5.6 PFR and IPC for PARSEC-3.0 Benchmark Suite on Platform 2



(a) barnes in EPT (b) barnes in SPT (c) fft in EPT (d) fft in SPT

(e) fmm in EPT (f) fmm in SPT (g) lu_cb in EPT (h) lu_cb in SPT

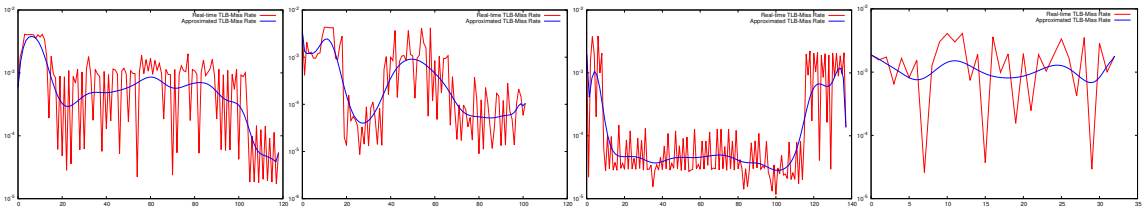
(i) lu_ncb in EPT (j) lu_ncb in SPT (k) ocean_cp in EPT (l) ocean_cp in SPT

(m) radiosity in EPT (n) radiosity in SPT (o) radix in EPT (p) radix in SPT

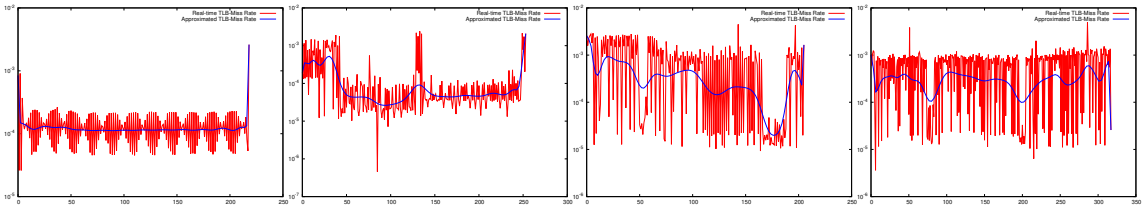
(q) spl_raytrace in EPT (r) spl_raytrace in SPT (s) volrend in EPT (t) volrend in SPT

(u) water_nsquared EPT in(v) water_nsquared SPT in (w) water_spatial in EPT (x) water_spatial in SPT

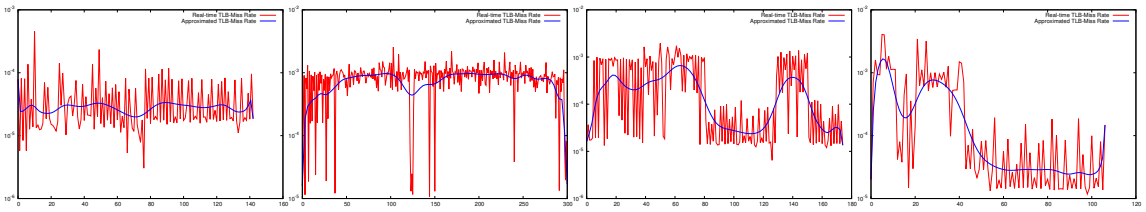
Figure 5.7 PFR and IPC for PARSEC-3.0 Benchmark Suite on Platform 2



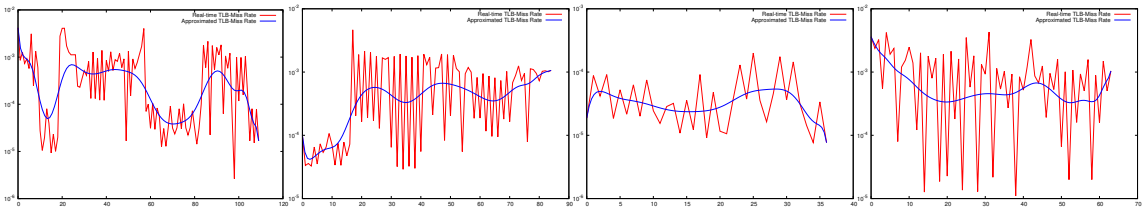
(a) blackscholes (b) bodytrack (c) canneal (d) dedup



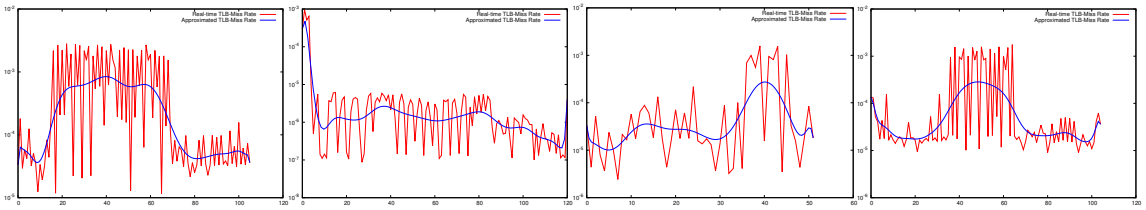
(e) facesim (f) ferret (g) fluidanimate (h) freqmine



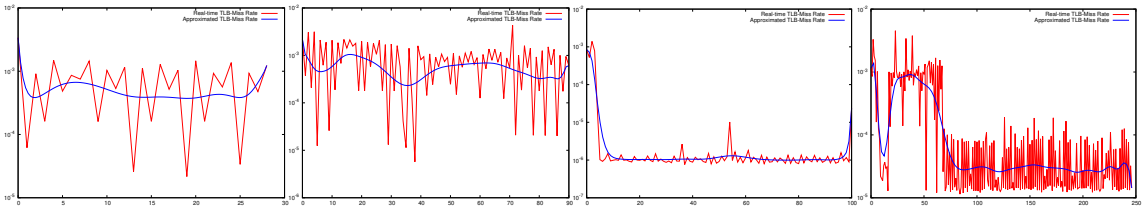
(i) raytrace (j) streamcluster (k) swaptions (l) vips



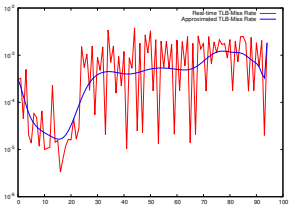
(m) x264 (n) barnes (o) fft (p) fmm



(q) lu_cb (r) lu_ncb (s) ocean_cp (t) radiosity

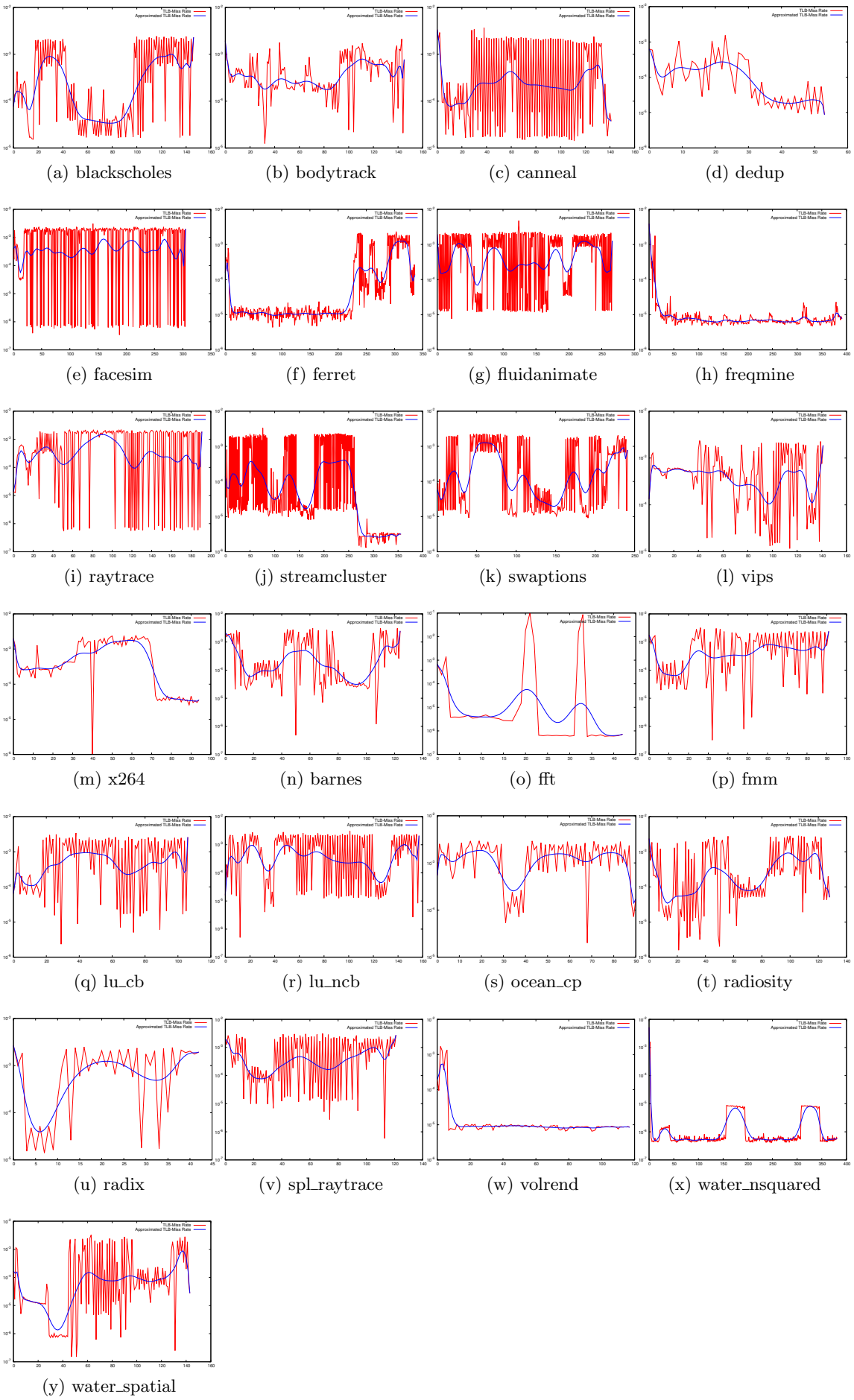


(u) radix (v) spl_raytrace (w) volrend (x) water_nsquared



(y) water_spatial

Figure 5.8 TLB Miss Rate for PARSEC-3.0 Benchmark Suite on Platform 1



(a) blackscholes (b) bodytrack (c) canneal (d) dedup

(e) facesim (f) ferret (g) fluidanimate (h) freqmine

(i) raytrace (j) streamcluster (k) swaptions (l) vips

(m) x264 (n) barnes (o) fft (p) fmm

(q) lu_cb (r) lu_ncb (s) ocean_cp (t) radiosity

(u) radix (v) spl_raytrace (w) volrend (x) water_nsquared

(y) water_spatial

Figure 5.9 TLB Miss Rate for PARSEC-3.0 Benchmark Suite on Platform 2

5.3 DPMS on QEMU-KVM for x86-64

5.3.1 Performance Data Sampling

Performance metrics are related with some specific OS kernel activities, such as *task scheduling*, *interrupt handling* and *context switch*, or hardware events, such as the *instruction fetching*, *cache missing*, *branch miss-prediction* and *pipeline stall*. As nowadays there is a rapidly growing need to analyze and tune the performance, software and hardware tools are becoming commonplace. Software tools are portable, but relatively intrusive, inefficient, and also incapable of measuring the hardware events. In contrast, hardware tools are less intrusive, efficient, capable of measuring a rich set of events of the hardware, especially the processor at the cost of a certain portability due to the difference in hardware architecture. To provide direct support for performance monitoring, the modern processor builds a set of special-purpose registers, the PMC (performance monitor counter) inside.

Since Pentium processors, PMC has entered into the x86 architecture as a set of MSRs (model-specific registers), aiming to monitor a number of events by selecting the appropriate parameters. Over a few generations, the PMC function has been enriched by an increasing number of selection events and a variety of control events, together with a division between the events specific to and independent of the micro-architecture, which are referred to as architectural and non-architectural events. Architectural events can be monitored on almost any processor regardless of its micro-architecture, therefore are more widely used for performance research. The associated hardware facility includes a finite number of performance-event counter MSRs and performance-event selector MSRs, of which the latter must be configured with the selected events and some other bits before the former begins counting.

As PMC is well documented in the system development manuals [196, 203] by specific processor vendors, it suffices to mention only the related parts for brevity. In Intel x86-64, the performance-event counter and selector MSRs are known as `IA32_PMCx` and `IA32_PERFEVTSELx`, respectively. Figure 5.10 depicts the bits layout of the performance-event selector MSRs, followed by a brief explanation of each field. Bits 0 through 7 contain the value of the selected performance event, and bits 8 through 15 the corresponding mask for the event (can be understood as a criterion for the validity of the event). Bits 16 and 17 indicate the privilege level at which the selected event is to be counted - PL 0 in OS kernel space, or PL 1, PL 2, PL 3 in user space. Bit 22 is the switch of the counter. When set, performance counter begins to count, or else it does nothing. Finally, bits 24 through 31 contain the counter mask, which controls how the specific event is to be counted, with the counter being incremented each cycle by the event count associated with multiple occurrences when it is zeroed out. Otherwise it is incremented by one or zero, depending on the comparison result between this mask and the occurrences of the event in a single cycle.

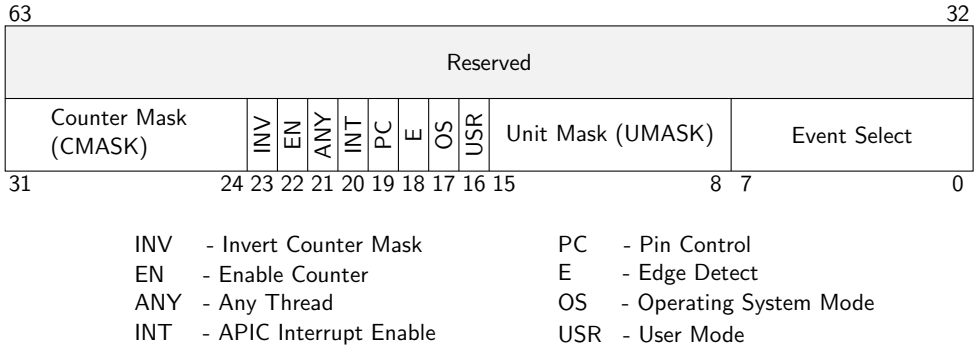


Figure 5.10 Bit field layout of `IA32_PERFEVTSELx` MSR [203]

The architectural feature is reflected by the following aspects [203]:

- Bit field layout of `IA32_PERFEVTSELx` MSRs is consistent across processor micro-architecture
- Addresses of `IA32_PERFEVTSELx` MSRs remain the same across processor micro-architecture
- Addresses of `IA32_PMCx` MSRs remain the same across processor micro-architecture
- Each logical processor owns a set of `IA32_PERFEVTSELx` MSRs and `IA32_PMCx` MSRs. The configuration facilities and counters are not shared between logical processors on the same physical processor core

In addition, both `IA32_PERFEVTSELx` MSRs and `IA32_PMCx` MSRs are arranged consecutively, with `0x186` and `0xc1` as the starting addresses, respectively. The number of counters available in a logical processor, the number of bits in the counter, and the number of supported performance monitoring events are all enumerated by the CPUID functions invoked by software.

To use a counter, the first step is to configure its associated selector by writing the event code and a proper mask into the selector MSR pointed by its address. Then it is cleared. The counter begins immediately to count the occurrence of the selected event when the `EN` field (bit 22) of the selector is set, and stops when `EN` is cleared, leaving a number in the counter to be read. This indicates the occurrences of the specified event. For practical use, however, the result for a single-shot counting is far from sufficient. What is normally expected by a user is a data stream, obtained in a continually repeating process to monitor the ever-changing behavior. Therefore, the counter usually undergoes a sequence of operations repeatedly: cleared, turned on, count, turned off, cropped, cleared and so on ... Listing 3(b)(c)(d) illustrate how the basic operations are realized for starting, stopping as well as cropping PMCs.

Table 5.4: Allocation and configuration of PMC MSRs

Event	Selector address	Counter address	Event Code	Mask
i-TLB miss	0x186	0xc1	0x85	(0x01ull << 8) | (0x03ull << 16)
d-TLB miss	0x187	0xc2	0x49	(0x01ull << 8) | (0x03ull << 16)
clock cycle	0x188	0xc3	0x3c	(0x00ull << 8) | (0x03ull << 16)
retired instructions	0x189	0xc4	0xc0	(0x01ull << 8) | (0x03ull << 16)

The hardware-related performance data proposed in the prior chapter is sampled in the above manner. For this, the performance-event selectors are configured with the parameters specified in Table 5.4. By revoking the function for cropping periodically, a stream of performance data is sampled in an arbitrary period of time, which outputs n_{tm}, n_{ret} and n_{cycle} mentioned previously. The other two, n_{pf} and n_{vmexit}, are software-related, therefore are sampled from the context of KVM. More specifically, the structure - `vcpu->stat` contains a number of statistics regarding the status of the virtual processors (cores), including `pf_fixed` and `exists` which serve as accumulators for the occurrences of fixed page faults and vmexits, respectively. More precisely, `pf_fixed` counts the number of fixed shadow page table entry (PTE) maps – not exactly the event occurs to the guest page table. However, as the shadow page table update is incurred by the updates in the guest page tables, `pf_fixed` serves as an approximate indicator to the number of fixed page faults in the guest page tables, thus is simply treated as n_{pf}.

Listing 3(e) illustrates a data structure used by DPMS for saving the performance data, the calculated results, as well as a few heuristic data used for other purposes.

So far, the functions for manipulating the PMCs have been defined. The next problem is to get them run. While this is straightforward for uniprocessors, things are complex for multi-core processors. A multi-core processor owns a multiple of identical but independent processing units in the form of cores or logical processors, including the units for performance monitor and hardware-assisted virtualization.

The problem boils down to: how many PMUs are needed for sampling the performance data per VM guest. The answer is: only one. Although each core owns a set of `VMCB/VMCS` and even MMU, all the cores that run the same guest are bound to use only a single set of shadow page

Listing 3 Functions and a data structure used by DPMS

```
#define IA32_PMC0               0xc1    /* Performance Event Selecter 0 */
#define IA32_PERFEVTSEL0        0x186
#define ITLB_MASK               ((0x01ULL << 8) | (0x03ULL << 16))
#define MISSED_ITLB             0x85
#define EventRegisterLow0       MISSED_ITLB | (ITLB_MASK & (~(1ULL << 22)))
```

(a) Macros used in functions for manipulating the performance monitor counters

```
static void pmu_init_launch(void *ignored)
{
        uint64_t evt_sel;
        uint64_t evt_msr, cnt_msr;

        /* set counter 0 for ITLB */
        evt_msr = IA32_PERFEVTSEL0;
        evt_sel = EventRegisterLow0;
        cnt_msr = IA32_PMC0;
        wrmsrl(evt_msr, evt_sel);
        wrmsrl(cnt_msr, 0);
        evt_sel |= 1 << 22;
        wrmsrl(evt_msr, evt_sel);
        ... ...
}
```

(b) Function to start the PMCs

```
static void pmu_get_counter_launch(void *info)
{
        uint64_t evt_sel;
        uint64_t evt_msr, cnt_msr;
        struct pmc_val_t *pv;

        pv = (struct pmc_val_t *)info;

        /* get and reset counter 0 for ITLB */
        cnt_msr = IA32_PMC0;
        rdmsrl(cnt_msr, pv->itmis);   data sampling
        evt_msr = IA32_PERFEVTSEL0;
        evt_sel = EventRegisterLow0;
        wrmsrl(evt_msr, evt_sel);
        wrmsrl(cnt_msr, 0);
        evt_sel |= 1 << 22;
        wrmsrl(evt_msr, evt_sel);
        ... ...
}
```

(c) Function to crop the PMCs

```
static void pmu_stop_launch(void *ignored)
{
        uint64_t evt_sel;
        uint64_t evt_msr, cnt_msr;

        /* set counter 0 for ITLB */
        evt_msr = IA32_PERFEVTSEL0;
        evt_sel = EventRegisterLow0;
        cnt_msr = IA32_PMC0;
        wrmsrl(evt_msr, evt_sel);
        wrmsrl(cnt_msr, 0);
        evt_sel |= 0 << 22;
        wrmsrl(evt_msr, evt_sel);
        ... ...
}
```

(d) Function to stop the PMCs

```
struct pmc_val_t {
        unsigned long itmis;
        unsigned long dtmis;
        unsigned long instr;
        unsigned long cycle;                    } sampled data
        unsigned long pf_fixed_last;
        unsigned long pf_fixed;
        unsigned long exits_last;
        unsigned long exits;

        unsigned long cur_pff;
        unsigned long cur_ext;
        unsigned long cur_tmr;

        struct {
                unsigned long flag_tmr;
                int flag_pff_spt;
                int flag_ext_spt;               } results
                int flag_pff_tdp;
                int flag_ext_tdp;
        } his[10];

        unsigned long sum_tmr;
        int sum_flag_pff_spt;
        int sum_flag_ext_spt;
        int sum_flag_pff_tdp;
        int sum_flag_ext_tdp;
        int sum_flag_tmr;

        unsigned long num;
        int paging_method;
        int need_switch;                        } heuristic data
        int pm_ready;
        int first_time;

        hpa_t root_hpa[4];
};
```

(e) Structure used by the DPMS

tables for address translation at a moment, no matter how many there are, and which paging method they are using. Limited by this fact, cores are not allowed to choose their own paging

Listing 4 Functions and a data structure used by DPMS (Continued)

```
int pmu_counters_crop(int mask, struct pmc_val_t *pmc_val_info)
{
        struct cpumask cpumask;
        int cpuid = 0;

        cpumask_clear(&cpumask);
        for (; mask; cpuid++, mask = mask >> 1) {
                if (mask & 1)
                        cpumask_set_cpu(cpuid, &cpumask);
        }
        on_each_cpu_mask(&cpumask, pmu_get_counter_launch, pmc_val_info, 1);
        return 0;
}
```

(f) Launch a PMC function on multi-core processor

method, but have to adhere to the same way. Therefore, a single set of PMU suffices to sample the performance data for all cores [7]. The core on which the PMU resides is elected as the leader among all others.

As an example, Listing 4(f) depicts how a function defined for cropping the PMCs is launched on a multi-core processor, although only one core is needed in reality.

5.3.2 Data Processing

Listing 3(e) depicts the data structure, `pmc_val_t`, which is introduced specific for DPMS. The purpose is rather straightforward. For example, to store the hardware-related performance data sampled from the PMCs and, more importantly, to keep the result after processing the raw data, and heuristic information that is critical for subsequent operations.

Section 5.2 has mentioned that the PF and *vmexit* for TDP-inclined workloads tend to exhibit a durable unusual high value under the *shadow paging*. For workload-detecting at run-time, these provide the ideal inputs. A strategy is to set thresholds, and for each metrics determine whether the current value is higher than the corresponding threshold. Depending on the comparison, it scores 1 or 0. Scores are kept in a ring buffer until the buffer is full. On the other hand, the SPT-inclined workloads may be detected by checking the combination of TMR, PF and *vmexit*. For this purpose, two members are introduced to the structure, `pmc_val_t` to count the bottom values of the PF and *vmexit* for the *nested paging*. Depending on the comparison between the current values of the two and their bottom thresholds, `flag_pff_tdp` and `flag_ext_tdp` score 1 if the result is *less than* or else 0. Listing 5 shows the implementation of the logic and procedure.

5.3.3 Decision Making

The decision is based on the sum of the currently kept scores. If the sum is higher than a certain percent of the maximum value (length of the ring buffer), it suggests that the workload is very likely suffering from frequent page faults and *vmexit*s. Therefore, the *nested paging* is preferred at this moment. The threshold for each metric, and the percentage are empirical values, which can be obtained by benchmark or learned by the machine. Do it by means of machine learning is left as the future work.

The percentage is applied to ensure that the majority of the recent PF and vmexit values are higher than the pre-determined thresholds, meanwhile leaving a certain degree of freedom for the sequence of occurrence (order is unimportant). Under *nested paging*, similar procedure is followed with the comparison result of TMR as the major criterion. In addition, as TMR has ambiguity for heavy performance loss, PF and *vmexit* are also monitored, but with the bottom thresholds to guard against some falsified cases (high TMR but little performance loss).

[7] There are micro-kernel hypervisors for embedded platform, which use separate page tables for each core.

Listing 5 Calculation of the performance metrics

```
void data_proc(struct pmc_val_t *pmc, struct kvm_vcpu_stat *stat)
{
        int pos;
        pmc->cur_tmr = (pmc->itmis + pmc->dtmis) * 1000 * 1000 * 1000 / pmc->instr;
        pmc->cur_pff = pmc->pf_fixed;
        pmc->cur_ext = pmc->exits;

        pos = pmc->num % 10;

        if (pmc->num > 9) {
                pmc->sum_flag_tmr -= pmc->his[pos].flag_tmr;
                pmc->sum_flag_pff_spt -= pmc->his[pos].flag_pff_spt;
                pmc->sum_flag_ext_spt -= pmc->his[pos].flag_ext_spt;
                pmc->sum_flag_pff_tdp -= pmc->his[pos].flag_pff_tdp;
                pmc->sum_flag_ext_tdp -= pmc->his[pos].flag_ext_tdp;
        }

        pmc->his[pos].flag_tmr = pmc->cur_tmr > TMR_THRESHOLD ? 1 : 0;
        pmc->his[pos].flag_pff_spt = pmc->cur_pff > PFF_SPT ? 1 : 0;
        pmc->his[pos].flag_ext_spt = pmc->cur_ext > EXT_SPT ? 1 : 0;

        pmc->his[pos].flag_pff_tdp = pmc->cur_pff < PFF_TDP ? 1 : 0;
        pmc->his[pos].flag_ext_tdp = pmc->cur_ext < EXT_TDP ? 1 : 0;

        pmc->sum_flag_tmr += pmc->his[pos].flag_tmr;
        pmc->sum_flag_pff_spt += pmc->his[pos].flag_pff_spt;
        pmc->sum_flag_ext_spt += pmc->his[pos].flag_ext_spt;
        pmc->sum_flag_pff_tdp += pmc->his[pos].flag_pff_tdp;
        pmc->sum_flag_ext_tdp += pmc->his[pos].flag_ext_tdp;
        /* always sum the 10 recent metric values */

        pmc->num++;
}
```

The current status is determined by querying `paging_method` in the structure of `pmc_val_t`, which is assigned the value of `tdp_enabled`, one of the global variables in the KVM context to control the use of hardware-assisted virtualization facility. In the conventional distributions of KVM, this is assigned with either 1 or 0 when the kernel is booting up, depending on the processor identification (CPUID) and BIOS (basic input and output system). Listing 6 illustrates the implementation of the above logic.

5.3.4 Switching Mechanism

The major tasks for switching mechanism has been sketched by the five questions in Subsection 4.1.4. However, those questions have only been answered halfway due to the lack of a hypervisor's implementation details. Should these details be clear, it is in a position to turn the general ideas into reality. Having made a general impression on the architecture of QEMU-KVM, more effort is still needed to get a clear understanding on how it works and related with the proposed ideas.

Figure 5.11 depicts the execution path of QEMU-KVM hypervisor when creating and running a guest. Execution is kicked off by starting QEMU in host user-space. As described previously, prior to `main()`, devices are already registered by some `constructors`. `main()` in QEMU does a lot of initialization for the devices and emulated hardware (such as the chipset and main board) to prepare for the startup of guest. During this process, control flow is guided to a specific point, where one callback of a device is triggered off, a chain of callbacks is also triggered off, leading eventually to the creation of various devices, including the VCPUs.

Compared with other devices which are emulated by a single dedicated QEMU I/O thread, the VCPU is a little different. Each VCPU is created as a "File" by QEMU, but attached by a kernel thread in kernel space. All VCPUs are forked out by the function - `qemu_thread_create` at the end of the call chain in user space. As the name suggests, it creates the required number of VCPUs by calling the POSIX API - `pthread_create`, with `qemu_kvm_cpu_thread_fn` as the

Listing 6 Decision making logic for the optimized data processing

```
void paging_mode_decide(struct kvm *kvm)
{
        struct pmc_val_t *pmc;
        struct kvm_vcpu_stat *stat;
        static int cur_num;

        pmc = &kvm->arch.pmc;
        stat = &kvm->vcpus[0]->stat;
        cur_num = (pmc->num < 10) ? pmc->num : 10;

        if (pmc->num <= 10)
                goto ret;
        switch (kvm->vcpus[0]->arch.mmu.direct_map) {
        case PM_EPT:
                if ((pmc->sum_flag_tmr > 6) && (pmc->sum_flag_pff_tdp > 6)
                 && (pmc->sum_flag_ext_tdp > 6))
                {
                        pmc->paging_method = PM_SPT;
                        pmc->need_switch = 1;
                }
                break;
        case PM_SPT:
                if ((pmc->sum_flag_pff_spt > 6) && (pmc->sum_flag_ext_spt > 6)) {
                        pmc->paging_method = PM_EPT;
                        pmc->need_switch = 1;
                }
                break;
        default:
                break;
        }
ret:
        return;
}
```

routine to be executed. The latter interacts directly with the KVM device (`/dev/kvm`) in kernel space via a sequence of the device-specific function - `ioctl` (input-output control) to set up and initialize the data structure needed by VCPUs.

Although `qemu_kvm_cpu_thread_fn` is not the interface between QEMU and KVM, it contains the functions - `kvm_init_vcpu` that directly communicates with the KVM device to initiate VCPUs and `kvm_cpu_exec` to execute it, during which the guest code is entered and executed. From the host user mode to host kernel mode to guest mode, control is passed gradually from outside to inside, with each transfer as a step further towards the ultimate goal - guest execution.

Due to the insufficient privilege, a guest needs to suspend its execution and return control to the host when certain events are encountered, such as the exceptions, errors, faults, and interrupts while acquires control again and resumes its execution after the hypervisor's intervention. These are known as *vmexit* and *vmentry*, respectively. Depending on the reason for *vmexit*, it can be sufficient to handle the event in host kernel space before the guest resumes. Otherwise it exits further to the host user space to get the I/O operation performed by QEMU's `iothread`. Therefore, the guest execution is "embedded" into a nested loop, with the inner one in host kernel space and the outer one in host user space. In view of the guest, a *vmexit* is lightweight if it falls only in the inner loop, or else heavyweight if it extends to the outer loop. A heavyweight *vmexit* involves context switch (transition to QEMU).

According to the statistics [204], 93% of the total *vmexit* in the early version of KVM (KVM-18) is lightweight triggered by the fault in shadow page table, and the rest by I/O and signal processing. Although currently no statistics has shown how these figures are for the latest version of KVM and QEMU, one thing is clear: paging fault remains always lightweight and should be handled in host kernel space. Based on this fact, the answer to the first question is, paging method switching must occur in `vcpu_enter_guest` inside the inner loop to be more efficient and sensitive to the changing workload.

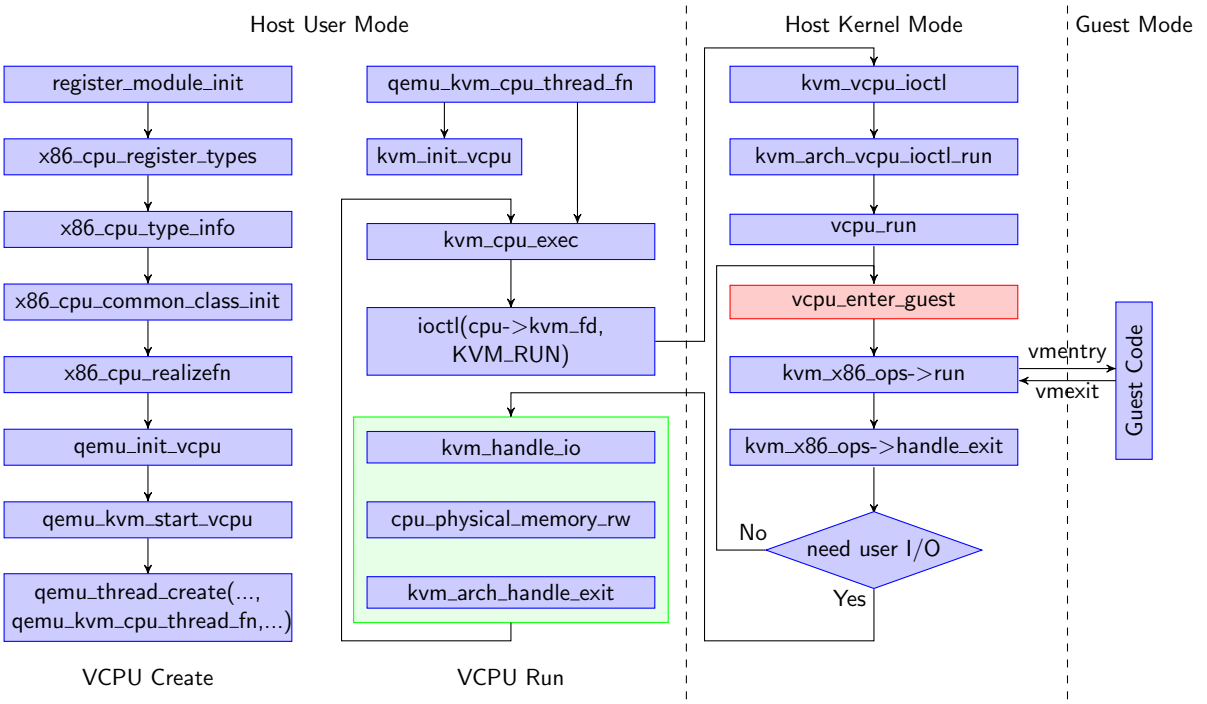


Figure 5.11 Execution path of the QEMU-KVM hypervisor

`vcpu_enter_guest` is the function containing the vendor-specific code for hardware-assistant virtualization extension - `kvm_x86_ops->run`, which launches the guest directly on the physical processors. Before that, the condition of the guest, more precisely, VCPUs, needs to be checked. The condition represents the overall status the guest has reached before the first execution or since the last exit from execution. A hypervisor maintains them properly on behalf of the guest. In the context of `vcpu_enter_guest`, a large portion of code is dedicated to handle dozens of requests raised during or after the guest execution, which may include time bookkeeping, clock updating, MMU[8] synchronization, TLB flushing, triple fault handling, interrupt handling, page fault handling and so on.

All these events are signaled by a bitmap - `requests` in the VCPU, with each bit representing an event. On the other hand, since no reconfiguration can be done when the guest is still running, and the hypervisor obtains control only after a *vmexit*, the inner function - `vmx/svm_vcpu_run` in the execution path comes therefore as the optimal place to decide whether to switch the current paging method. The decision is made based on a comparison of the previous and current optimal paging methods hinted by *Decision Making*. Nothing needs to be done if the two are identical, or else a flag, `need_switch` in the `pmc` structure is set when a request for MMU reloading is set in the `request` bitmap.

As soon as `vcpu_enter_guest` is entered next time, the bitmap gets checked. `need_switch` is used to tell whether the request for MMU reloading is the intent to switch the paging method, or to do something else[9]. Even if the MMU must be reloaded, the task is deferred to the function `kvm_mmu_reload` for a better reuse of the existing code and let the request for other reasons unaffected. In this way, the hypervisor is notified of the request for paging method switching intended by the running workload without much modification in the current hypervisors. The related code sections are shown in Listing 7(a) and (b).

In the current hypervisor, `kvm_mmu_reload` is called to reload the MMU after the previous one had be unloaded for a certain reason. For this implementation, it is slightly modified to add the new function without much impact on the original code. The virtual MMU contains mainly

[8]Refers exactly to the emulated MMU - SPT/TDP-associated callbacks in this context.
[9]There are many other reasons that require the reloading of MMU.

a tree of page tables whose entries store the PFNs (physical page frame number) of the referred virtual address. The starting address of the root table serves as the entrance of MMU, known as the *root*. Originally, the function calls `kvm_mmu_load` if the *root* is still invalid[10]. `kvm_mmu_load` allocates a page and links it to the page table tree as the root page, followed by a shadow page synchronization[11]. Finally, the *root* is saved by the MMU structure in each VCPU. To minimize the impact of modification, the MMU unloading and loading due to paging method switching are arranged in this function, without messing up with the switchings due to other reasons.

Listing 7 Notification of the request for paging method switching

```
static void __noclone vmx_vcpu_run(struct kvm_vcpu *vcpu)
{
        ... ...
        int pre_pm, cur_pm;
        pre_pm = vcpu->kvm->arch.pmc.paging_method;
        ... ...

        cur_pm = vcpu->kvm->arch.pmc.paging_method;
        if (cur_pm != pre_pm) { /* if any change in paging method */
                vcpu->kvm->arch.pmc.need_switch = 1;
                kvm_make_all_cpus_request(vcpu->kvm, KVM_REQ_MMU_RELOAD);
        }
}
```

(a) Request signalled

```
static int vcpu_enter_guest(struct kvm_vcpu *vcpu)
{
         ... ...
        if (vcpu->requests) {
                if (kvm_check_request(KVM_REQ_MMU_RELOAD, vcpu)) {
                        if (!vcpu->kvm->arch.pmc.need_switch)
                                kvm_mmu_unload(vcpu);
        ... ...
        r = kvm_mmu_reload(vcpu);
        ... ...
}
```

(b) Request checked

```
static inline int kvm_mmu_reload(struct kvm_vcpu *vcpu)
{
        int need_switch;
        int direct_map;

        need_switch = vcpu->kvm->arch.pmc.need_switch;
        direct_map = vcpu->arch.mmu.direct_map;

        if (likely(vcpu->arch.mmu.root_hpa != INVALID_PAGE))
                if (!need_switch)
                        return 0;
        if (need_switch && direct_map && !vcpu->vcpu_id) {
                vcpu->kvm->arch.pmc.need_switch = 0;
                kvm_x86_ops->tdp_to_spt(vcpu);    <-- switch to SPT
                return 0;
        }
        else if (need_switch && !direct_map && !vcpu->vcpu_id) {
                vcpu->kvm->arch.pmc.need_switch = 0;
                kvm_x86_ops->spt_to_tdp(vcpu);    <-- switch to TDP
                return 0;
        }
        return kvm_mmu_load(vcpu);
}
```

(c) Switching launched

[10] In KVM, an address is valid if it is not -1, represented by a sequence of 1's in binary
[11] Eventually performed by `mmu_sync_children`, which marks *sync* and *write-protect* of its children pages

Since a switching may occur either from the *shadow paging* to *nested paging*, or reversely, the appropriate conditions are constantly checked for the due operation. In either case, the current configuration for paging, which is reflected by the variable `direct_map` in the MMU of each VCPU is the first condition. The *nested paging* is in use if the variable is set, or else it is the *shadow paging*. Next, the flag `need_switch` is checked to see whether the request is posed for paging method switching. Finally, considering that a multi-core virtual processor is running on a multi-core physical processor, a reconfiguration such as the paging method which tends to affect the state of the machine should be performed only once, usually by a "leader", VCPU0 in this case, among all cores. Therefore, the `vcpu_id` is the third condition to be checked.

If the combination of conditions for the branch is true, the corresponding function is called. As their names imply, `tdp_to_spt` and `spt_to_tdp` are the two functions performing the paging method switching operations. What they achieve are the reverse effect. It deals with annulling the configuration of the current paging method and then configure the chosen one. In fact, the *shadow paging* and *nested paging* are similar in that both of them serve as the alternative or complement towards the original page tables in the guest. For this reason, the *nested paging* is able to seamlessly fit into the original framework exclusively for *shadow paging*. The advantage is that most of the software infrastructure is reused by the new design. In theory, a root page table and the associated page fault handling mechanism are sufficient for a workable virtual MMU to support the guest. That is exactly what the mentioned two functions should accomplish. However, in practice, they are different in behavior.

One major difference is that the configuration of the *nested paging* involves not only software, but also hardware. To anull the configuration of the *nested paging*, the related hardware must also be taken care of. The related hardware is known as VMCS (virtual machine control structure) for Intel processor, and VMCB (virtual machine control block) for AMD processor. The *nested paging*'s operation relies on the hardware in the following aspects:

- The VMCB or VMCS hardware contains the appropriate information to create or recover the state of the physical processor for the guest execution, including dozens of system registers
- Root of shadow/nested page tables gets loaded into the MMU when the guest is running
- The paging status of the processor, a bit implies whether the *nested paging* is enabled

With those in mind, the task each function must perform is clear. The `tdp_to_spt` invalidates the nested page tables, prepares a set of shadow page tables, clears the TDP-*enabled bit* in the VMCB or VMCS hardware, and returns for further execution. In contrast, the `spt_to_tdp` does just the reverse. It invalidates the shadow page tables, prepares a set of the nested page tables, reconfigures the VMCB or VMCS hardware, finally sets the TDP-*enabled bit* to inform the processor of a state change in paging method, and proceeds to run the guest. The two sets of page tables work independently in their own cycles, respectively. In this case, an optimization is to allocates memory for both of them in the hypervisor's context. However, compared with the multiple sets of process-specific shadow page tables in a single guest, only a single set of the nested page table is available. That means, the nested page table is actually much more stable and permanent in nature than shadow page tables. Consequently, the former could be retained even during the domination of the latter, and reused next time as soon as the *nested paging* regains control. Due to the volatility of its content, this is not the case for the *shadow paging*.

For this reason, the nested page tables are left with its root saved in the `pmc` structure and retrieved when *nested paging* is resumed, but the shadow page tables are simply destroyed with nothing being saved and restored. The hardware configuration involves a few aspects, such as the `exception bitmap`, in which the `PF_VECTOR` is set or cleared to enable or disable `vmexit` due to page faults in guest. The TDP-*enabled bit* is indicated by `SECONDARY_EXEC_ENABLE_EPT` in the `SECONDARY_VM_EXEC_CONTROL` region of the VMCS, and `nested_ctl` in the `control` region of the VMCB. To summarize the discussion and description, the code for launching paging method switching is listed above in Listing 7(c), but the definition of the switching functions, as well as the entire patch for the DPMS implementation, can be found at https://github.com/zhayu.

Till now, question 3 and 4 were already answered with the above passages. Since the last question deals with the convergence of the algorithm for decision making, it could be more appropriate to be discussed in Chapter 6.

The major aspects of design and implementation were described above. However, a few critical detail issues, which are closely related with the implementation have not yet been covered. These include the following aspects:

- Repetitive mechanism that ensures a periodical output of performance data and fulfillment of the due operations for paging method switching
- Integration of the performance monitoring mechanism into the QEMU-KVM context
- Synchronization of the DPMS in an execution environment with multi-core processor
- Optimize algorithms for making decision of the paging method

For each aspect, a subsection is dedicated to discuss the problems and corresponding measures.

5.3.5 Repetitive Mechanism

In Linux kernel, a bunch of APIs was created for the purpose of repetitive work deferral and periodic timer scheduling. Among them, a critical issue is the timing, which provides the time base and accounting the time for the whole system. A commonly used timing source is the *jiffies* based on a global variable and counts the number of ticks occurred since the moment when the system was started up. Although how the ticks are counted depends on the specific hardware, a basic approach is by means of the interrupt in the processor. In kernel *jiffies* is used to deliver both the absolute time, and calculate the time-out value for a timer, which is frequently needed for many different purposes [205].

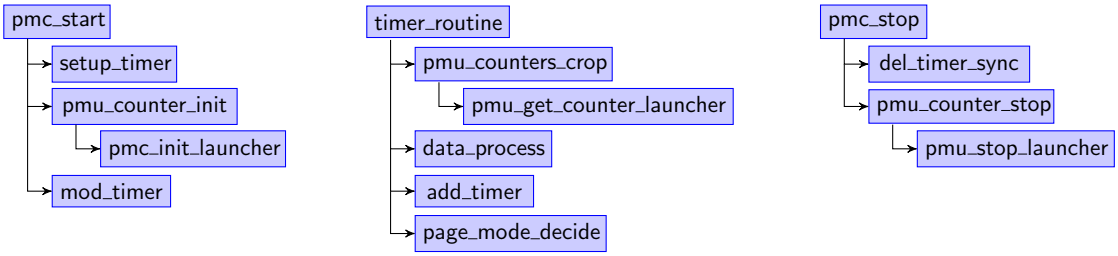
According to the ticking granularity, there are the standard and high resolution timers, which take a tick of *jiffies* (1 to 4 ms) and *nanosecond* as the basic ticking unit, respectively. A standard timer is less accurate than a high resolution timer, but is already sufficient for a considerable number of the current applications. DPMS is also no exception. A standard timer in the Linux kernel is defined by the `timer_list` structure, mainly containing a pointer to a list of timers, the value for expiration, a user-definable callback function, a pointer to a region as the parameters of the callback, and a few optional variables for various uses. Linux kernel provides a group of APIs to create, initialize and control this timer, of which the commonly used types are:

- `void init_timer(struct timer_list *timer);`
- `void setup_timer(struct timer_list *timer,`
  `void (*function)(unsigned long), unsigned long data);`
- `void mod_timer(struct timer_list *timer, unsigned long expires);`
- `void del_timer(struct timer_list *timer);`

The API names are self-explanatory. Nevertheless, something still deserves to be mentioned. First, a timer can be initialized by either `init_timer` or `setup_timer`. With the latter, the timer can be set up without extra efforts, while with the former, the call routine and its parameters need to be assigned explicitly. In fact, the latter is merely a macro wrapper of the former for more convenience. The choice is pretty a matter of one's preference. In addition, if the latter is used, a caveat is that type-casting is needed when passing the user-provided parameters from an upper-level function. A pointer of the user-defined type needs to be type-cast into `unsigned long`. Inside the callback, however, parameters are unpacked by performing the reverse type-casting. Finally, `mod_timer` is used to advance the timer by increasing a specific number of *jiffies* to the expired moment to ensures that the timer will go off again in another cycle. `del_timer`, opposite to `init_timer`, removes the timer from the observed list, thus the timer's associated event will not be handled any more.

So far, the repetitive mechanism has been introduced. The next step is to apply it in DPMS and let the desired tasks periodically repeat itself. As the DPMS is a "machine-wide" mechanism, a guest needs only one set of the infrastructure. A possible choice is to include the previously mentioned `pmc` structure and a timer in the `kvm` or `kvm_arch` structure. The latter is chosen since it is better to maintain the DPMS-related infrastructure as centrally as possible to minimize the impact of modification on the original code[12]. In the context of a guest, they are effectively serving as global variables accessible anywhere when dealing with the same guest.

The topmost functions for PMC control are declared as `pmc_start` and `pmc_stop`, respectively. The former is used to start the PMCs on a specific core (normally Core 0), and do initialization work necessary for performance data sampling as soon as the guest has been booted up, such as creating the timer and initializing the `pmc` instance. The other one, conversely, is used to remove the timer and stops the PMCs when the guest is about to go down. The most interesting part, data sampling function, is fulfilled by the callback routine of the timer. Furthermore, *Data Processing*, *Decision Making*, and "timer renewal" are all encapsulated in this function. Similarly, the APIs defined for PMC control also form a call chain, as illustrated in Figure 5.12(a,b,c).



(a) PMC and timer starting (b) PMC sampling and timer renewal (c) PMC and timer stopping

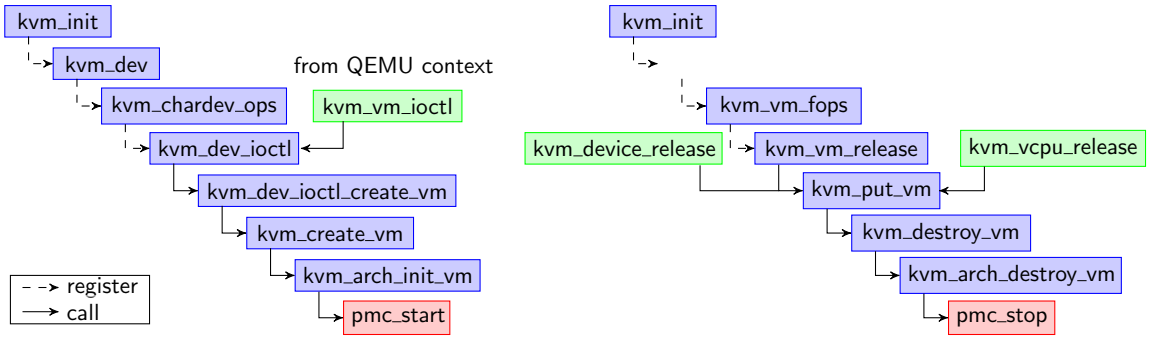
Figure 5.12 Call chains formed by the APIs for PMCs control

5.3.6 PMC Mechanism in QEMU-KVM Context

For a workable DPMS, it is necessary to integrate the PMC mechanism into the QEMU-KVM context and let the performance monitoring counters be controlled automatically when needed. The target is that the PMCs are started as soon as the guest enters into execution, sample data afterwards, and are stopped as the guest is shutdown.

Naturally, a more flexible approach is to let the user decide when to start and stop this function by issuing commands from user-space. However, as it involves modifying not only KVM, but also QEMU, for simplicity, it is better to be left as a further work, and focus first on the core task – to merge the PMC mechanism into the current KVM context. Although the startup process of the QEMU-KVM hypervisor has already been briefly described in Figure 5.11, it drops little hint on this as only the guest's VCPUs are illustrated there. A similar situation is faced when deciding where the timer, as well as the `pmc` structure should be. Hinted by the fact that the DPMS is a "machine-wide" mechanism, and a guest needs only a single set of this facility, they were appended into `kvm_arch`, a "machine-wide" data structure. Comparably, as the PMC mechanism is a part of the DPMS, it also belongs to the "machine-wide" operation which in KVM context corresponds to those functions performing guest creation, initialization, and destruction. Figure 5.13 depicts the cascading call chain formed by them and the way PMC mechanism is integrated in the hypervisor.

[12] In `kvm_arch` modification is already there to introduce another hash table for storing *SPT* and *TDP* separately.



(a) Call chain for `pmc_start`

(b) Call chain for `pmc_stop`

Figure 5.13 Cascading call chain where PMC mechanism is integrated in

Being consistent with Linux kernel in coding style, KVM also adopted the callback feature when dealing with file system operations. Three basic entities of KVM, the char device, the VM guest instance, and the VCPU are all implemented and operated based on the file system. Therefore, KVM device drivers fall into three categories, namely, the hypervisor-device (`/dev/kvm`) oriented, VM guest oriented, and VCPU oriented drivers. As shown in Figure 5.13, these KVM drivers are organized as callbacks of the related devices, registered once during device initialization, and triggered off under specific conditions. As the device-oriented driver, `kvm_dev_ioctl` is assigned to a callback of the KVM char device, and registered by `kvm_init`, which is always and only called when the KVM modules are being loaded. Similarly, `kvm_vm_release` serves as a callback of the VM-oriented driver and is registered by the same function.

Once all drivers are registered, they may be invoked by different callers. Figure 5.13(a) depicts that for the call chain headed by `kvm_dev_ioctl`, besides at the first time being triggered when loading KVM modules, it is also called during guest creation by QEMU through the `ioctl` with command `KVM_CREATE_VCPU`. As for the call chain in Figure 5.13(b), the `kvm_put_vm` function, which eventually turns off the PMCs, has three callers, but the PMC finalization is normally performed by the branch headed by `kvm_vcpu_release` after the guest has been gracefully shutdown, or unexpectedly terminated for various reasons. In this way, the PMC mechanism is integrated into the KVM context and ensured an automatic control by the hypervisor itself.

5.3.7 DPMS for Multi-Core Processor

In Section 5.2.1, a multi-core processor has already been concerned when dealing with the choice of PMU. In principle, PMCs on any core may serve as a source of performance data. However, considering that the paging method is only determined per machine rather than per core, one source suffices for this. The behavior of a workload may be non-uniform, so different core may demand different paging method. It cannot be handled with by the current *nested paging*. In other words, the multi-core feature has little influence on the Performance *Data Sampling* and *Data Processing*. What really matters is whether and how can the paging method be switched synchronously on all cores. A guest enters into execution only when its VCPUs had been scheduled on different physical processors or cores, which implies that for a single guest at the same time, there are as many execution contexts as the number of physical processors or cores that are running VCPUs. Thus, the *vmexit* and *vmentry* for one VCPU on one core are performed independently unless task is synchronized. In fact, the *vmexit* occurs non-synchronously due to the different instructions being executed at that moment. However, since the paging method switching requires that all VCPUs to be suspended so that the hypervisor has the chance to adjust itself, it involves the synchronization and interrupting the running VCPUs among the multiple cores.

5.4 STDP on QEMU-KVM for x86-64

Based on the analysis of the performance limitations associated with the four-level paging scheme for the *nested paging*, Section 4.3 proposes a two-level paging scheme among all possible choices. To study the feasibility and effectiveness of such a solution, a further step is to implement it for a specific hypervisor. This section focuses on the implementation for QEMU-KVM on x86-64.

As mentioned previously, the *nested paging* reuses the data structures and most of the functions for the *shadow paging*. Figure 5.14 provides an overview of the data structures for shadow page tables in KVM, of which `kvm_mmu_page` is the basic unit which glues all the concerned parts together. In this structure, the shadow page tables, a pageful of 64-bit `sptes` (shadow page table entries) is pointed at by `spt`, and the attributes of this table such as paging mode, dirty and access bits, level etc. are indicated by the corresponding bits in `role`.

The page pointed to by `spt` has its `page->private` pointing back at the shadow page structure. `sptes` in `spt` point at either leaf pages, or lower-level shadow pages [206]. As the `sptes` contained in a shadow page could be either one level of the PML4, PDP, PD and PT, `pte_parents` provides the reverse mapping for the `pte/ptes` pointing at the current page's `spt`. Bit 0 of `parent_ptes` is used to tell the difference from one to many. `0` indicates that only one `spte` is pointing at this page, so let `parent_ptes` point at this `spte`, and `1` means that multiple `sptes` are pointing at this page, so let the `parent_ptes&~0x1` point at a data structure containing those `sptes`.

`kvm_mmu_page` also maintains a minimal set of data to mark the current status and keep the `sptes` updated. `unsync` indicates whether the translations in the current page are still consistent with the translations in the guest page tables. Inconsistency is incurred when the translation has been modified before the TLB is flushed, which has been read by the guest. `unsync_children` counts the `sptes` in the page pointing at pages that are `unsync` or have unsynchronized children. `unsync_child_bitmap` is a bitmap indicating which `sptes` in `spt` point (directly or indirectly) at the pages that may be unsynchronized.

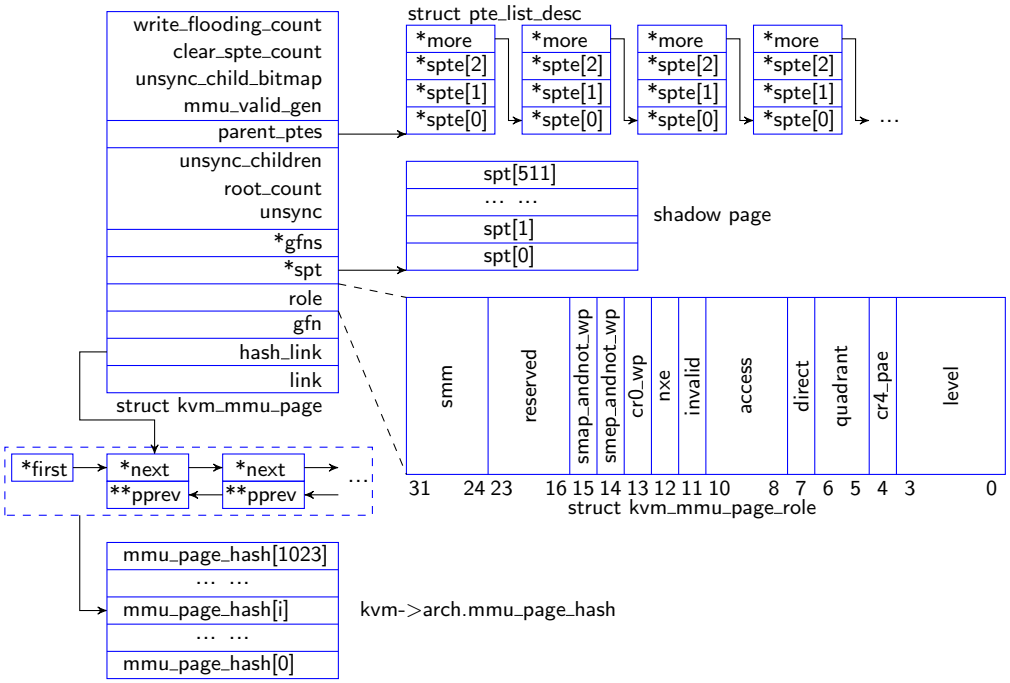


Figure 5.14 Data structure shared by the *shadow paging* and the *nested paging*

Multiple `kvm_mmu_page` instances are linked by an `hlist_node` structure headed by `hlist_head`, which form the elements in the hash list - `mmu_page_hash` contained in `kvm->arch`. Meanwhile it is also linked to either the list - `active_mmu_pages` or `zapped_obsolete_pages` in `kvm->arch`, depending on the current status of the entries it contains.

The above description reveals which parts are affected by the new implementation of *nested paging*. First of all, with less page levels, the tree structure formed by page tables is shrunken in height, but expanded in width (the number of entries a table contains). The page tables, which contain the address translations for guest memory pages are the major aspect involved. Multiple increase in the number of page table entries means multiple increase in memory demand for this kind of page tables.

Since the functions come along with the original paging scheme proved effective and mature during the evolving of the KVM, the central task is to adapt them to the new paging scheme.

5.4.1 Restructured Page Table

In Linux kernel, three basic functions, namely, `vmalloc()`, `kmalloc()` and `__get_free_pages()` are used to allocate large chunk of memory. `vmalloc()` is used to allocate memory continuous only in virtual address, which is relatively easier to obtain, but not ideal for performance. The second and third allocate memory chunk continuous in both virtual and physical addresses. However, due to the size limitation for memory allocation, they tend to frustrate the attempts for huge and continuous memory chunk. Furthermore, `kmalloc()` fails easily for large memory allocation, especially in a low-memory case [207]. The amount of memory `__get_free_pages()` can allocate is also limited within $2^{MAX\_ORDER-1}$, where `MAX_ORDER` is 11 in the current Linux kernel for x86 architecture, implying that at most 4MB memory can be obtained in the hypervisor for each allocation. In this condition, a single-level page table is neither practicable nor affordable from a perspective of the balance between resource and performance. Even in practice, a more reasonable choice is also the two-level page tables.

Compared with the four-level paging scheme, a philosophy behind the two-level paging scheme is to reduce the paging level by expanding the volume of page tables. Although this involves only assigning larger chunk of memory to `spt` for a `kvm_mmu_page` instance, it demands a fundamental change to the entire infrastructure. As the enlarged page table contains 2^{18} entries, or 2^9 of the 4KB-size pages, which claim a total of 2^9 meta-data for all the conventional size pages it contains. A major problem is how to maintain the one-to-one mapping between a page table and its meta data in the `kvm_mmu_page` instance for the new scheme.

For this purpose, a new structure, `page_entity` is introduced. As illustrated in Figure 5.15, `page_entity` plays a similar role for the new scheme as `kvm_mmu_page` for the original scheme. More importantly, it contains a pointer to a region of 2MB – sufficient to accommodate the 2^{18} page table entries, or 2^9 normal 4KB page tables. In order to reuse the exiting code and minimize the impact, it is better to retain the original form for managing the page table in the new scheme, which means that the page table entries are grouped in 2^9 4KB-page tables and maintained by their meta data as previously.

The `page_entity` holds an array of 2^9 original `kvm_mmu_page` to trace the 2^9 pages, with the structure and link almost unmodified except a new member - `idx` marking the index of this meta data in the aforesaid array. `*spt` is also retained for compatibility with the *shadow paging*. The instances of `page_entity` are linked by a list `page_entity_list`[13]. `level` and `index` indicate the level of a `page_entity` instance, and the position at that level. PFN can be found by locating the `page_entity` instance by PHD, and then the page table by PLD.

The major operations affected by the above changes in that data structure are the following:

- Large page table (2MB in size) allocation and deallocation in host physical memory
- Calculation of a page table entry's index in the normal table
- Normal page table (4KB in size) allocation and deallocation from the hash list of page tables

[13] A normal list rather than a hash table is used simply because currently the number of `page_entity` instances is quite limited (5 instances for a guest with 4GB RAM), and searching overhead does not matter too much.

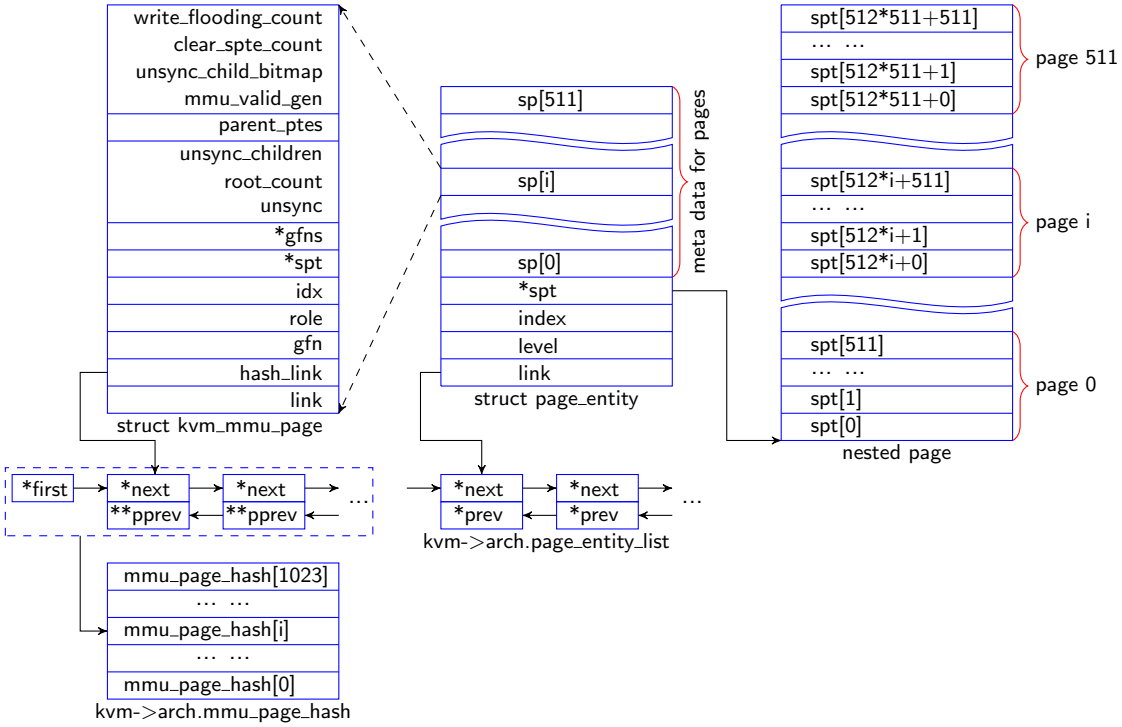


Figure 5.15 Data structure for the restructured page tables

In KVM, the 4KB-page tables are continuous memory regions and pre-allocated by the function `mmu_topup_memory_caches` for future use. Similarly, as in the new scheme 2^9 4KB page tables are merged into a larger chunk of 2MB, memory chunk of this size also needs to be pre-allocated; therefore, functions are defined to allocate and free them as kernel objects belonged to the VCPU of the guest [14]. `__get_free_pages()` and `free_pages()` are the functions performing these tasks, respectively. However, since the total `sptes` in such a huge table controls a region of 1GB for the guest physical memory, it is unlikely to be reclaimed as freely as a 4KB page table during the lifetime of a guest.

For the new paging scheme, the index of an `spte` in the page table is frequently used (such as in a reverse mapping), but the calculation in a normal 4KB-page table becomes non-trivial since the direct acyclic graph structure in the original scheme has somewhat changed. The calculation, which was once performed by the code path:

```
sp = page_header(__pa(spte)); index = spte - sp->spt;
```

follows a new path:

```
sp = page_header(__pa(spte));
pe = container_of(sp, struct page_entity, sp[sp->idx]);
index = spte - (pe->spt + (sp->idx << 9));
```

As previously mentioned, the reason is that in the original scheme, the page pointed at by `spt` has its `page->private` pointing back at the `kvm_mmu_page` structure. In the new scheme, `spt` has been the common base for 2^{18} `sptes` it contains. The starting address of its nearest normal page table (4KB) is needed for obtaining the index of the `spte`. Therefore, if the corresponding `kvm_mmu_page` structure is set pointed at by this normal page table's `page->private` when first allocated, the needed address can be simply deduced by using `sp->idx` as shown in the path. But the first step is to obtain the `page_entity` it is associated with by using a macro `container_of` in Linux kernel.

[14] Such caches are a part of the architecture in the VCPU of KVM guest.

A more difficult part is the page fault handling in the hypervisor. In the conventional scheme, all page faults are handled by the function "`tdp_page_fault`". The major task it performs is to figure out the level of page table a faulting address falls into, compute the host physical address, and fill it into the corresponding entry of the page table. The last step performed by the function "`__direct_map`" is the core task for the page fault handling.

`__direct_map` is in a loop of maximum four repetitions, traversing from the root of the shadow page tables down to the page table of the fault level. During the loop, any missing `spte` on this path is filled, and a `kvm_mmu_page` instance is allocated at the next level pointed by the newly filled `spte`. If a `kvm_mmu_page` instance is obtained by the function "`kvm_mmu_get_page`", it gets linked to the page table acyclic graph by filling its address into an upper-level `spte`.

`kvm_mmu_get_page` obtains a `kvm_mmu_page` instance either by retrieving one from the hash table "`kvm->arch.mmu_page_hash`", or by freshly allocating if no one has ever been found there. The same logic is followed to obtain an instance of `page_entity`. When a faulting address falls into a region not yet covered by any `page_entity` in the current `page_entity_list`, a new instance must be allocated. The corresponding meta data also needs to be deduced from the available parameters. The two concerned are the index of this `page_entity` and that of the `kvm_mmu_page` instance associated with the page table. They are calculated in a way as below:

```
pe->index = (role.level == 2)? -1 : SHADOW_PT_INDEX(gaddr, 2);
sp->index = SHADOW_PT_INDEX(gaddr, 1) >> 9;
```

where the macro is defined as:

```
#define PT64_INDEX(address, level)\
        (((address) >> PT64_LEVEL_SHIFT(level)) & ((1 << PT64_LEVEL_BITS * 2) - 1))
#define SHADOW_PT_INDEX(addr, level) PT64_INDEX(addr, level)
```

Last but not least, the normal page table's `page->private` is set as the following:

```
base = pe->spt + (sp->index << 9);
set_page_private(virt_to_page(base), (unsigned long)sp);
```

The patch about the software part of STDP will be found at https://github.com/zhayu.

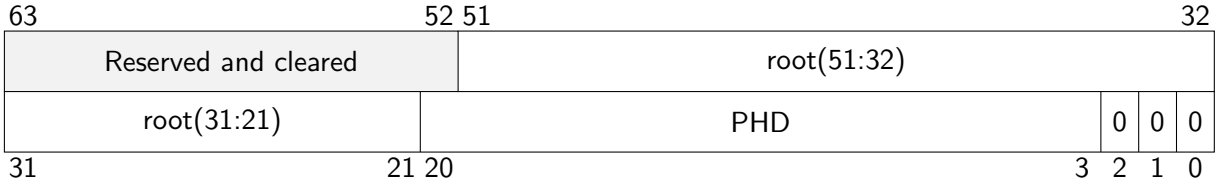
5.4.2 Adaptive MMU for TDP

Now the software part of STDP has been implemented, whose task is to create and maintain the nested page tables in the second dimension. Nevertheless, it does not work without the support from hardware side. The processor's MMU is unaware of such a change in the paging structure thus can not make correct use of these page tables for address translation. A particular difficulty is the lack of hardware support for this function. The common practice is naturally to create the customized hardware by using a hardware description language and a developing board (such as FPGA) and get it integrated into the environment. However, this is beyond the scope and capability of the hypervisor researcher. An alternative is to create it by an software emulator, which tends to offer the user a certain degree of freedom to customize the targeting hardware and bring it under one's own control. This subsection focuses on an analysis and explore the possibility in this respect.

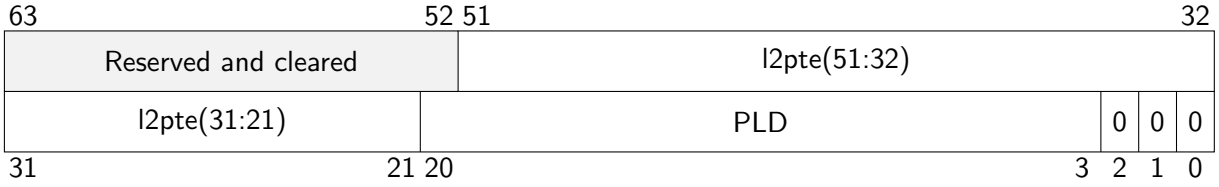
Currently the x86-64 architecture supports only a walk-length of 4 in the second dimension for all paging modes of the guest OS. If the adaptive physical MMU is implemented, a walk-length of 2 can be supported in the second dimension, meaning that at most 2 paging structure entries need to be accessed to translate a guest physical address. 48-bits of a 64-bit GPA are effectively used by being partitioned into a PHD|PLD|offset format by the logical processor to traverse the paging structures in the second dimension. The paging procedure is described below.

The 2MB-aligned level-2 table is located at the physical address specified by bits 51:22 of the root for the second-dimension page table (EPTP in VMCS, or CR3 in VMCB). It contains 2^{18} 64-bit entries. The entry is selected by using the physical address as depicted in Figure 5.16(a).

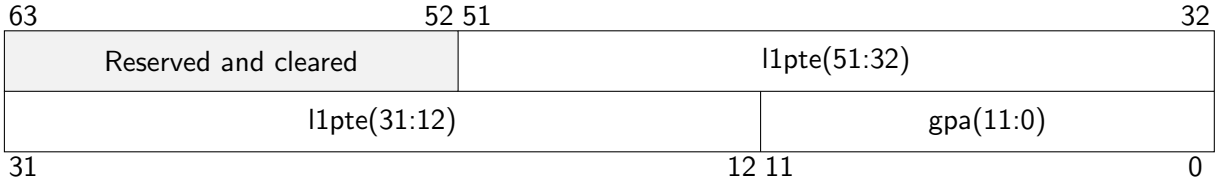
An entry of level-2 table is selected by PHD (bits 47:30) and controls a region of 1GB of the guest physical address space. The format is specified by Table 5.5.



(a) Bit format of GPA used for locating the level-2 page table entry



(b) Bit format of GPA used for locating the level-1 page table entry



(c) Bit format of HPA computed in the second dimension

Figure 5.16 Bit formats used by the paging process in STDP

Table 5.5: Format of the entry in level-2 table in the second dimension

Bits	Specification
0	Read access, indicates the read privilege for the 1GB region controlled by this entry
1	Write access, indicates the write privilege for the 1GB region controlled by this entry
2	Execute access, indicates the privilege for fetching instructions from the 1GB region controlled by this entry
7:3	Reserved and cleared
8	Access flag, indicates whether the 1GB region controlled by this entry has been accessed by any software
9	Ignored
10	Execute access for user mode linear address, indicates the privilege for fetching instructions by user-mode address from the 1GB region controlled by this entry
11	Ignored
(N-1):21	Physical address of the 2MB-aligned level-1 page table pointed at by this entry
51:N	Reserved and cleared
63:52	Ignored

The 2MB-aligned level-1 table is located at the physical address specified by bits 51:12 of the level-2 page table entry (l2pte for convenience). It contains 2^{18} 64-bit entries. The entry is selected by using the physical address as depicted in Figure 5.16(b). An entry of level-1 table (the last level) is indexed by PLD (bits 29:12) and maps a region of 4KB page in the guest physical address space. The format is specified by Table 5.6. Finally, the host physical address is computed in a format shown in Figure 5.16(c).

With this scheme, GPA is translated into HPA by accessing the memory only three times in the second dimension, which can save at least 40% efforts compared to the conventional TDP scheme. While minimizing the interference by the hypervisor, it avoids the high overhead associated with the multi-level nested page table walking.

For software, especially the OS or hypervisor researchers, a common problem could be the lack of true hardware when developing the functions that rely on currently non-existing hardware. One may design and implement the hardware themselves using the reconfigurable developing board, however, it adds more challenge, diverts the focus of the software developer, and increases

Table 5.6: Format of the entry in level-1 table in the second dimension

Bits	Specification
0	Read access, indicates the read privilege for the 4KB region controlled by this entry
1	Write access, indicates the write privilege for the 4KB region controlled by this entry
2	Execute access, indicates the privilege for fetching instructions from the 4KB region controlled by this entry
5:3	Memory type for the 4KB page controlled by this entry
6	Ignore PAT memory type for the 4KB page controlled by this entry
7	Ignored
8	Access flag, indicates whether the 4KB region controlled by this entry has been accessed by any software
9	Ignored
10	Execute access for user mode linear address, indicates the privilege for fetching instructions by user-mode address from the 4KB region controlled by this entry
11	Ignored
(N-1):12	Physical address of the 4KB-aligned page pointed at by this entry
51:N	Reserved and cleared
62:52	Ignored
63	Used for suppressing the “convertible virtualization exception”

the risk of failure. To address this problem, researchers have come up with the means to create the target hardware execution environment by using software, which is known as a full system simulation or emulation[15]. Although numerous simulator software exist, very few of them are able to simulate the system to the circuit level. Two commonly used software are QEMU and Wind River Simics [208, 209, 210]. As an emulator, QEMU emulation has the ability to present the target hardware platform for system software developing.

In recent years, an increasing number of processor features such as the Intel VMX, EPT and AMD SVM, NPT, are added to the its CPU model for nested virtualization. It means that those features merely present when nested virtualization is enabled. There has also been discussion adding such support for pure emulation purpose, yet this is not yet available. On the other hand, as a commercial software, Simics is more mature in this respect. Not only Intel VT-x, but also EPT can be emulated since the recent version of Simics. Even so, as the modification of the MMU behavior involves the access to the source code of the Simics CPU core model, which is not open for the normal users, the current Simics is not the right tool for accomplishing this task on x86-64 platform. However, Simics do provides such possibility for modifying the MMU's behavior, but for an old UltraSPARC[16]. In other words, although it is non-trivial to modify the MMU, the major barrier does not lie in the technology.

5.5 Summary

Based on the QEMU-KVM hypervisor, the proposed ideas - DPMS and STDP are implemented on the software side. For DPMS, most of the functional units are implemented and integrated with the other parts of KVM. A unsolved problem is the switching in case of multi-core processor, which makes more sense than merely for a single-core processor. Furthermore, the question – How fast the sampling and switching should be, will be discussed further in the next chapter.

For STDP, the software part has been fully implemented, which involves an adaption of data structures and functions for the proposed paging scheme. However, as the proposed adaption also involves hardware, limited by the capability and current technical status, neither the proposed physical hardware nor the hardware emulator has been created or made available. Even so, from a technical perspective, it is practicable for the processor vendors to implement this proposal. With these implementations, Question 7 in Section 1.4 is answered.

[15]The two indeed have a few subtle distinctions, but are still used interchangeably probably for convention.
[16]Informed by the communication with Dr. Robert Guenzel - a Simics expert in WindRiver.

Chapter 6 Experiments and Evaluation

Contents

In this chapter, the feasibility and effectiveness of the proposed design for improving the performance of memory management system virtualization are tested in a series of experiments. Due to the lack of a practically workable implementation, currently the testing cannot be done for STDP. Therefore, the first design - DPMS is the target for the tests.

Section 6.1 specifies the objective and motivation of these experiments, which are elaborated in Section 6.2 in more detail. Section 6.3 presents the results of the corresponding tests and a discussion about these results. Finally, the tests of the design and the evaluation of the results are summarized in Section 6.4.

6.1 Objective

From the software engineering point of view, software testing is the process of executing a given program or system with the intent of finding errors [211]. It is also the process of evaluating the attributes and capability exhibited by the program or system and decide whether they meet the requirements stipulated at the beginning of design. Due to the human's limited ability to manage complexity of a system, as well as the dynamic nature of software, testing cannot be exhaustive, and bugs can never be completely eliminated. The testings involved in this experiment are also no exceptions. After a series of testings, many design flaws and coding errors have been spotted and fixed, still missing a few undiscovered or unsolved. And the ultimate purpose is to determine and demonstrate that the proposed designs

1. can be implemented
2. the implementation meets the requirements for the design, and
3. the implementation is superior to the original one in the concerned respect, and makes sense in solving the encountered problems

In fact, the above mentioned aspects boil down to two different issues of a new design, namely, the functional testing and the performance testing, which are typically used in software engineering to improve the quality of a software product. As a logical product, software is intangible, the quality is invisible and cannot be measured directly. However, just as Figure 6.1 illustrates, the quality of software is determined largely by three sets of factors - exterior, interior and future quality, each of which represents a dimension in the software quality space [212]. A well-designed testing always tries to cover as many factors as possible in this space. Depending on the type of the product, a testing may lay different importance on these factors. For a full-fledged, especially commercial software, all these factors could be taken into consideration, and the testing tends to be involved all through the process from design to delivery.

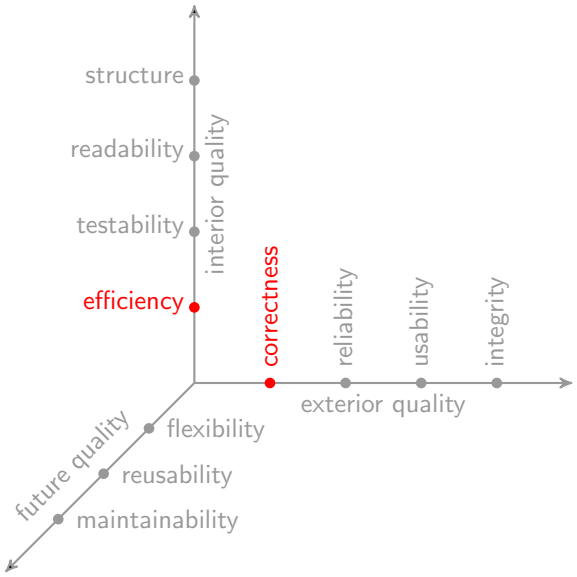


Figure 6.1 Typical quality factors involved in software engineering [212]

DPMS has two top priorities, functional correctness and efficiency (performance). Most of the other factors, in contrast, are beyond the scope of this dissertation thus left out for brevity. Therefore, among many aspects, the test will primarily focus on the following points of the proposed design and its implementation:

1. Does DPMS switch the paging method for guest workloads in different cases?
2. Does DPMS outperform the conventional static approaches?
3. How much speedup can it bring for the normal workloads?
4. Can the selected workloads benefit from the use of DPMS?
5. What kind of workload will most likely benefit?

6.2 Testing Design

To answer the above questions, tests are designed with two focuses - function and performance. First, the correctness of DPMS, including the *Performance Data Sampling*, *Decision Making* and *Paging Method Switching* will be tested with the hypervisor being initially configured with the *shadow paging* and *nested paging*, respectively. On conditions that all these function work properly and fulfill their tasks as expected, the focus of the testing will be shifted to a higher level - the performance, with the purpose of evaluating the efficiency of the proposed design and implementation. The following two subsections are dedicated to the basic thoughts when performing these kinds of tests.

6.2.1 Functional Correctness Test

Performance Data Sampling

The test focuses on whether the PMCs can be enabled, disabled, and performance data can be sampled periodically from the hardware PMCs as well as the counters in the hypervisor software. By inserting a few debug information (wrapped up by `printk()`) to the source, and directing the output into a log-file (`/var/log/dmesg`, or `/var/log/syslog`) of the Linux kernel, the activity of DPMS can be brought under close watch.

Decision Making

This test stresses on how reasonable and sensible the decision for a paging method can be made based on the observed performance data. The term "reasonable" indicates that the choice of the paging method reflects the intrinsic demand of the workload and is in favor of a higher run-time performance, while the term "sensible" is used to describe the promptness as the decision is made after the data has been sampled and processed.

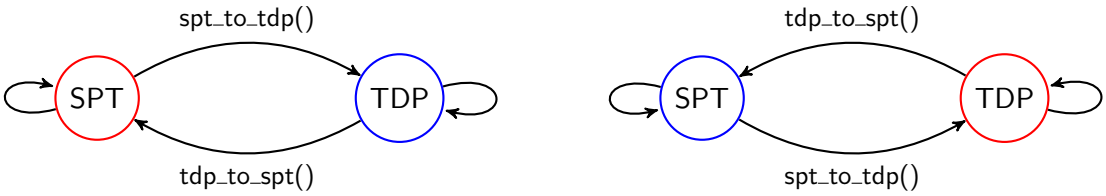
If the correctness for *Performance Data Sampling* indicates that PMC-related operations can be performed properly as expected, the same word for *Decision Making* is richer in its meaning. Thus the reasonability and sensibility come as the two indicators of the correctness for *Decision Making*[1]. In practice, the two metrics are neither visible nor measurable in a micro perspective. Nevertheless, they can be reflected by their impact on the performance of the workload in a macro perspective. The resultant of reasonability and sensibility is the performance gain or loss against that without the use of DPMS. By analyzing the predictive result of the workload whose preference is already known in the previous benchmarks, the reasonability and sensibility can be evaluated as a whole in a general form.

Paging Method Switching

As the core of DPMS, this part is in charge of accomplishing the intent of the running workload, therefore has an immediate impact on the performance and determines whether the total effort is worthwhile for the performance gain. In this case, functional correctness should be defined as the consistency between the reconfiguration of a dynamically switched-in paging method and the initial configuration without the use of DPMS. In other word, a correctly performed switching ensures an almost identical configuration for the paging method as if the configuration is done from the very beginning. In practice, however, this tends to be a quite rigid requirement due to the hassle of the hardware-specific features. A full restoration of the configuration is practically neither possible nor necessary. To take a step backwards, a compromised standard is a partially restored, yet workable configuration for the concerned paging method. The test attempts to find out whether the paging method switching works under this condition, and if not, what hinders.

Considering that the hypervisor can be booted up with the *nested paging* feature either enabled or disabled, the test needs to cover the switching in four cases, namely 1) SPT to TDP, initially with SPT; 2) TDP to SPT, initially with SPT; 3) SPT to TDP, initially with TDP; and 4) TDP to SPT, initially with TDP. Figure 6.2 depicts these cases in which the hypervisor is initially configured with SPT and TDP, respectively.

[1] Though another factor - the stability can be equally important due to its immediate impact on performance, it belongs more to the nature of performance than to the functional correctness. To be strictly, sensibility has to do with both functional correctness and performance, thus should be involved as the former is concerned.



(a) Initially with SPT (b) Initially with TDP

Figure 6.2 Switching in four cases

6.2.2 Performance Test

Provided that all the above function correctly, the testing enters into performance stage, through which the essential value of the whole design and implementation are evaluated. The performance testing means to find answers to, and is therefore guided by the questions 2 to 5 in Section 6.1.

First of all, the primary intention, or value of DPMS lies in the expectation that this system is helpful for the workloads, which exhibit strong inclinations to a particular paging method in the conventional static approaches to avoid large performance loss by dynamically adapting to their favorite ones at run-time. In an ideal case, DPMS is supposed to exhibit the quality of suiting their needs, no matter with which paging method the hypervisor has entered into execution.

Accordingly, the testing is about the comparison of the performance yielded in an environment with, and that in an environment without the use of DPMS. To illustrate the effect DPMS brings about, the same benchmark applications are used, which have been discussed in Section 3.3 and are used to seek the performance drawback of the hypervisor. The effect is considered positive if the performances of these workloads yielded in an environment with the use of DPMS is higher than those yielded without the use of DPMS, or else negative.

When it comes to the speedup - a direct indicator of DPMS's usefulness, one fact is that it is not only closely related with the execution environment, but to a large extent also depends on the workload. The speedup is calculated for each benchmark application based on the comparison suggested above, with the intention to demonstrate the capability of DPMS when dealing with workloads, especially those which have a bias towards the paging method.

Although from an empirical point of view, the ideal benchmark applications are those having a bias towards the paging method, the real-world workloads can be diverse, therefore are not limited to this small set. These are inexhaustible. If the workloads is diversified by making the benchmark applications more representative, the conclusion will be more convincing. For this reason, a number of other applications, which showed no bias towards paging method are also involved in the testing.

The last question deals with the scope of application for DPMS. From the results accumulated in the above testings, it naturally comes to a position to identify the cases in which DPMS's use is recommended, or on the contrary, not necessary.

6.3 Test and Benchmark Results of DPMS

6.3.1 Performance Data Sampling

The *Performance Data Sampling* function has been tested on the three platforms. The results show that this function works as expected on all these platforms. PMCs can be turned on upon the starting of a guest, sampled periodically by the hypervisor, and turned off as the guest is shut down. This clearly demonstrates that the call chain in Figure 5.13 can be triggered correctly from the top to the bottom level, therefore is suitable for serving the DPMS.

One problem is that the PMCs cannot be turned off in the cases where the guest execution is terminated unexpectedly, for example, crashed due to a certain reason, because the due path which performs the task has no chance to be called in this situation. As a result, these PMCs get out of control unless the host is not rebooted. In fact, it has no negative impact on the function, as when the guest is started again, PMCs on another core will be used by the hypervisor. The only negative side is that PMCs are not guaranteed to be turned off gracefully in this case.

6.3.2 Dynamic Paging Method Switching

Dynamic Paging Method Switching function has been tested on the above three platforms. The results show that the dynamic switching works as expected on the two Intel platforms, but on the AMD platform the switching immediately causes a reboot of the guest rather than a recovery or restoration of the desired page tables. Although the reason is not yet clear, an assumption is that the AMD NPT is implemented slightly different from the Intel EPT, and treats the switching operation as a kind of failure, thus passes the control flow simply to the exception-handler and triggered the reboot.

Similar operation works on one platform but fails on the other. The failure suggests the lack of support in hardware, or more likely, a few bugs in the implementation of DPMS, but does not harm the practicability of the basic idea. At least it has been implemented on two Intel platforms. It reveals a vendor-specific difference between the implementations of TDP by Intel and AMD. Further efforts are still needed to seek the fundamental reasons.

6.3.3 Problem Analysis

DPMS works as expected on the two Intel platforms. The paging method can be switched as soon as the hypervisor feels necessary, in response to the changing behavior of the guest workload in memory accessing. This proves the basic idea for such a change in the current framework is feasible and practicable. However, the current implementation is still bothered with a problem. The guest may quite unexpectedly be crashed by two types of the so-called “KVM internal error”, as the following error log shows:

```
KVM internal error. Suberror: 1
emulation failure

KVM internal error. Suberror: 3
extra data[0]: 80000306
extra data[1]: 31
```

After a careful examination in the source code, it is believed that the first type of error is caused by an attempt of the guest to access the MMIO region of the memory, and the second by a fault occurred during the delivering of an event to the guest which caused a *vmexit*. According to the annotations in the related source code, it is also due to an attempt of the guest to access the MMIO region. It indicates that the MMIO region may have been corrupted due to the page method switching. Limited by experience, an effective solution have not been found yet. Things becomes increasingly worse with an increase of the number for switching. Until now, a basic

way to overcome this is to slow down the pace for switching, and perform the switching only when it is really necessary.

In Section 5.3.4 and 5.5, the convergence of the algorithm fro decision making, as well as the sampling frequency are mentioned, but more detailed discussions are deferred to this Chapter. With the testing and benchmark results, it becomes more possible to answer these questions.

6.3.4 Convergence and Stability

The switching is governed by the conditions for decision-making. The convergence and stability of a switching process are therefore closely related with these conditions. The testing has showed that the combination of conditions in the initial design was not able to guarantee the convergence and stability of a switching process, because IPC, TMR and PFR are too easy to be influenced by a few unexpected factors, such as errors for data sampling, which tend to disturb the outcome quite randomly. Switching may fall into a live lock, as one page method has been just applied, but begins to trigger a move towards the opposite direction, and so on, without end. Eventually it may end up in a guest crash due to the aforementioned errors, or damage the overall performance even if not. For this reason, after a careful study of the run-time statistics in Section 5.2, the page fault, vmexit, and TMR are selected as the conditions for switching. It proves that by doing this, the swing (jitter or oscillation) between SPT and TDP has been effectively eliminated. As a result, the DPMS becomes more stable and efficient.

6.3.5 Sampling Frequency

The sampling frequency is also a critical factor for the performance and effectiveness of DPMS. An ideal sampling frequency is expected to ensure that the paging method can be switched timely in response to the changing behavior of the workload, meanwhile remain so in a relatively longer time to avoid unnecessary switching. This is actually a compromise between the sensitivity and stability, but with the same target – to maximize the performance gain. As both the two metrics are influenced by the sampling frequency, the relation can be expressed as functions of the latter, of which the sensitivity is monotonically increasing, while the stability monotonically decreasing for the sampling frequency. The intersection point gives the optimal sampling frequency in this case. Figure 6.3 depicts this kind of relation in a theoretical approach.

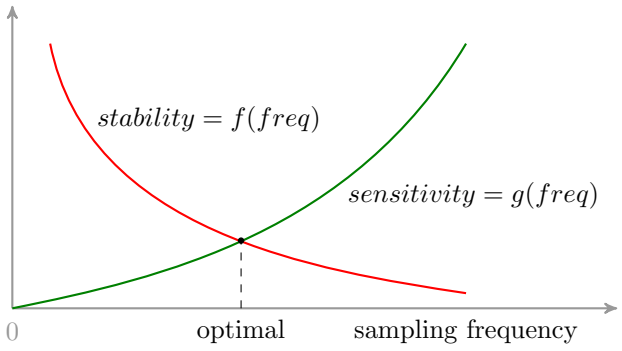


Figure 6.3 Theoretical relation between stability and sensitivity

Indeed, the run-time behavior, or characteristic of a workload may change during its execution. This can be reflected by the variation of IPC, PFR, TMR in Figures 5.6, 5.7, 5.8 and 5.9. The performance of a workload may benefit more from the switching during its execution. However, currently, considering that crash may be incurred due to too frequent switching, it is reasonable to keep the frequency relatively low, not only for the sake of avoiding the crashes, but also for better performance. The frequency is lowered to a level that paging method may rarely change during the execution of the same workload.

6.3.6 Performance of DPMS

The performance of DPMS is evaluated by the following way: first, the individual TDP-inclined workloads are executed for a number of times (ten time by default) with the *shadow paging* initially, to test whether they could be benefited from DPMS. Next, the individual SPT-inclined workloads are executed for a number of time (also ten times) with the *nested paging* initially, to test whether they can be benefited from DPMS. Then, the other normal workloads are executed ten times from the *shadow paging* and *shadow paging*, respectively. Finally, all the workloads are started sequentially with either the *shadow paging* or the *shadow paging*, to check the stability of DPMS over a large time-span.

The last step makes sense especially because it is quite common for HPC workloads to execute within a large time-span, hence the execution environment must be stable or robust enough. It is a pity that the current DPMS is not stable enough to meet this requirement, merely due to the aforementioned errors. But it is already stable to withstand a number of switchings when running most of the normal workloads. Even a guest crashed, the execution will be resumed by restarting the guest. By this means, the performance data were collected, and plotted as Figure 6.4a and 6.4b. With the results illustrated in the two Figures, it comes to a position to answer the five questions posed at the beginning of this chapter:

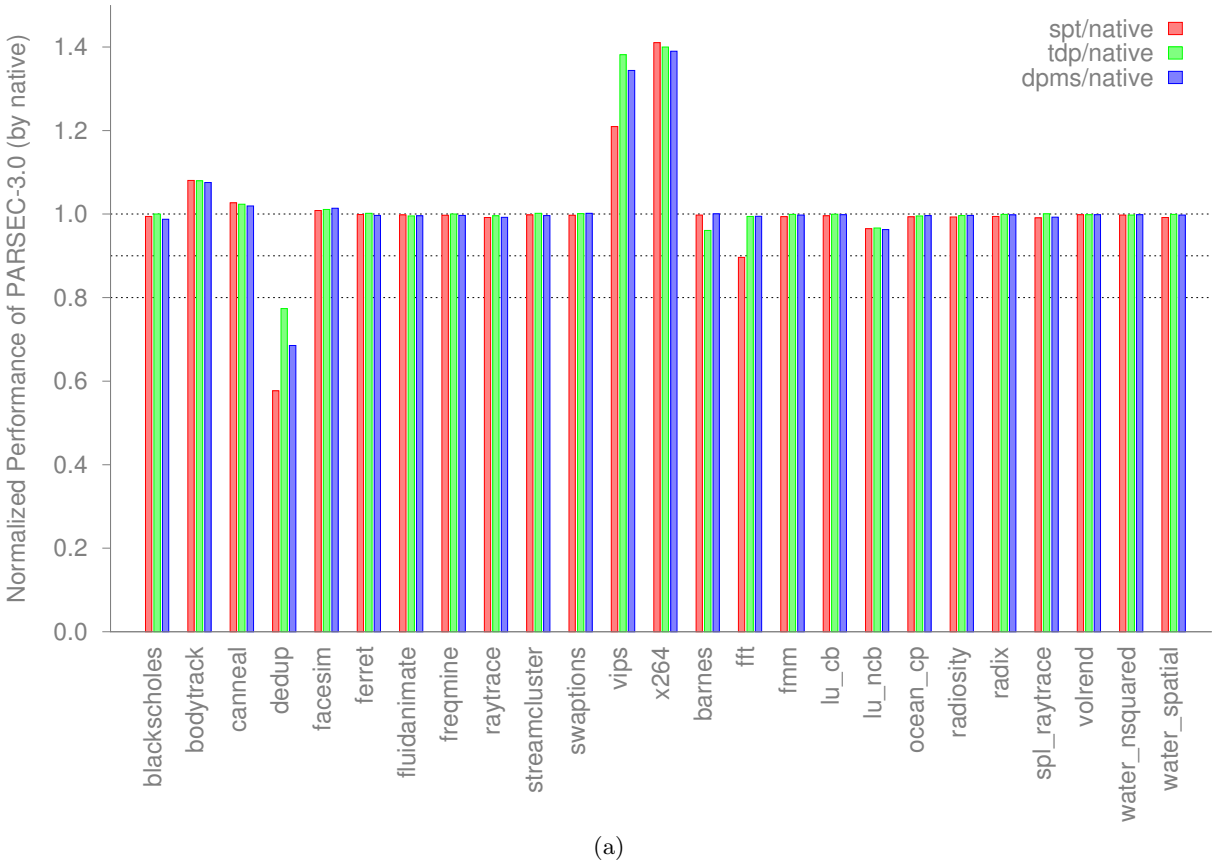
- Does DPMS switching paging method for workloads in guest in different cases?
  Yes, DPMS switches the paging method for the guest workload when it "feels" necessary, and from either *shadow paging* and *shadow paging*, or vice versa.
- Does DPMS outperform the conventional static approaches?
  Yes, by dynamically adjusting the paging method, significant performance loss can be reduced, compared with the conventional static approaches.
- How much percent speedup can it bring for a normal workload?
  The performance gains: `dedup` (P1: 10.9%, P2: 20.95%), `vips` (P1: 13.47%, P2: 9.09%), `barnes` (P1: 3.92%, P2: 1.03%), `fft` (P1: 0%, P2: -2.75%).
- Can most of the selected workloads be benefited from the use of DPMS?
  Most of the workloads for benchmarking show little bias towards paging method, so normally they are not benefited from DPMS, but DPMS ensures at least no harm to their performance.
- What kind of workload will most likely be benefited, what not?
  The design of DPMS means to improve the performance of the workloads which suffers due to an inappropriate paging method in the hypervisor. Therefore, only those workloads are likely to be benefited from the use of DPMS.

6.4 Summary of the Evaluation

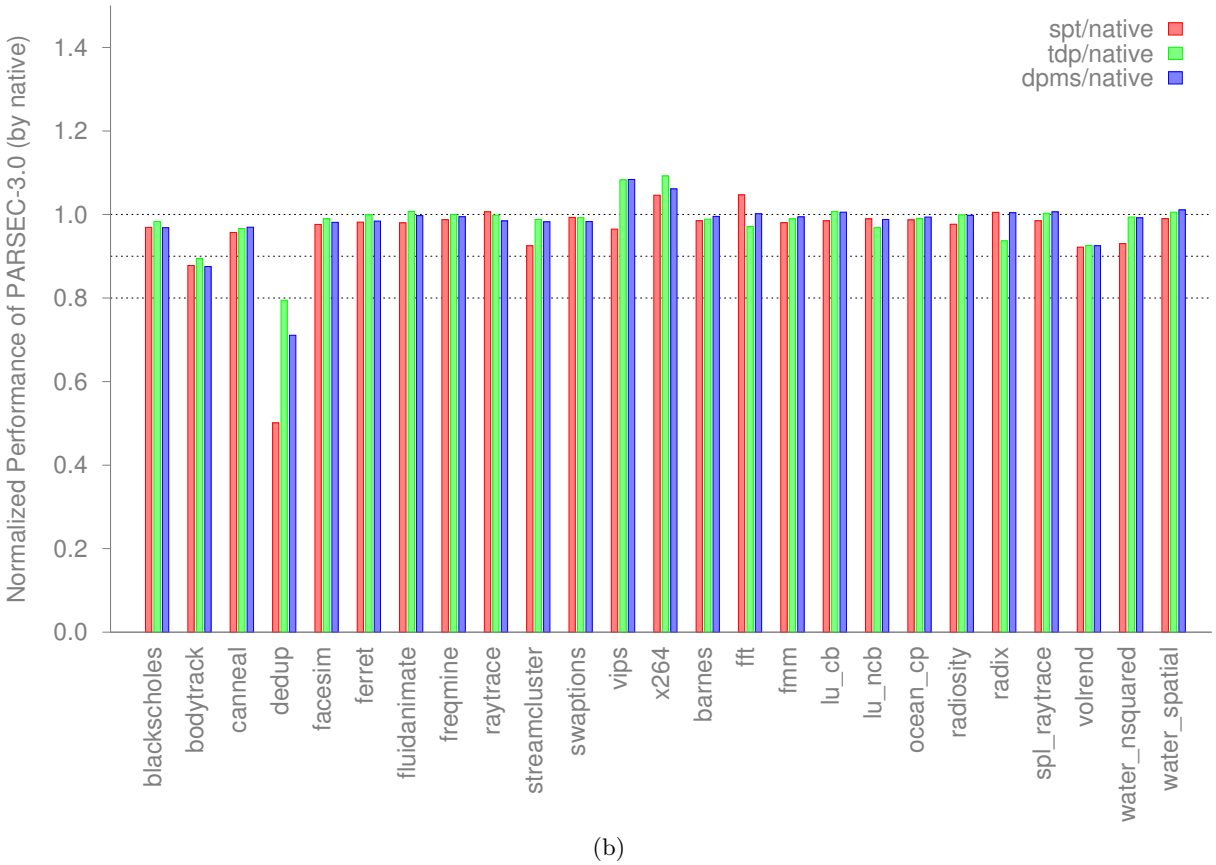
With the aim to verify the feasibility of the proposed design, the DPMS has been tested for both functionality and performance. It shows that DPMS works properly as expected, except for a few issues which do not harm too much. The performance data are sampled by the PMCs, and the paging method is switched dynamically on the Intel platforms.

By running a mix of the benchmark workloads sequentially, the hypervisor is free to choose the paging method desired by the workload. The benchmark results show that both TDP- and SPT-inclined workloads are benefited from the use of DPMS, meanwhile other workloads are not harmed in performance.

From these practice, the feasibility and the usability of DPMS for HPC workloads are proved. With the test and benchmark, Question 8 in Section 1.2 is answered.



(a)



(b)

Figure 6.4 Performance comparison among DPMS, SPT and TDP
(a) Platform 1 (Intel Core i7-6700K), (b) Platform 2 (Intel Xeon e5-1620-v2)

Chapter 7 Other Aspects of Performance Loss and Future Work

Contents

In this chapter, performance-related factors other than memory virtualization will be discussed. The virtualization of processor and the I/O system may also be potential source of performance loss in a certain cases.

Section 7.1 summarized the related work for performance loss due to processor virtualization. Section 7.2 surveys the efforts for reduing the performance related to I/O virtualization, selects the benchmark for investigating the performance loss in the inter-node pattern and presents the benchmark results. Section 7.3 anticipates the future works for all aspects.

7.1 Processor and Scheduling Aspect

Related work

Due to the continuous breakthrough, processor has gradually been removed as a major source of performance loss, especially since the adoption of hardware-assisted virtualization technology, Intel-VT and AMD-V. Represented by these processors, x86-based ISA become classic virtualizable, therefore reduced much development effort for the hypervisors. The VM guest can be executed almost natively on the hardware processors (cores), with only a small portion of its instructions still having to be emulated by software[1]. Consequently, the processor virtualization is normally not a problem if the instructions to be emulated (mainly for device I/O emulation) take a small share in the total.

Computational-intensive workload may yield nearly bare-metal speed if few instructions need to be emulated. In contrast, I/O intensive workload suffer significantly due to the more frequent context switch between VM guest and the hypervisor. In this case, the performance loss incurred by processor virtualization is hidden by that of the I/O device virtualization, which ultimately emerges as the dominating factor in the overall performance loss [132].

[1]These are mainly instructions dealing with I/O operations, whose ports or address don't exist therefore software emulation is needed.

However, processor virtualization may incur performance loss due to other reasons, especially in the cases when multi-core virtual machines are running on multi-core physical machine. This has for long been the default configuration of servers in data centers and HPC supercomputer. Multi-core virtual machines are adopted, to create a similar execution environment especially for multi-threading applications, and to harness the power of such physical multi-core machines. When a multi-core virtual machine runs on a multi-core physical machine, heavy performance loss may be encountered by a certain applications performing thread-synchronization operations.

Normally the VCPU (virtual processor) of the guest is implemented as thread or task for the host OS kernel or hypervisor, being scheduled in and out as a normal task by the hypervisor's scheduler. Conceptually, the VCPU itself is the device where task scheduling occurs as soon as the guest enters into execution.

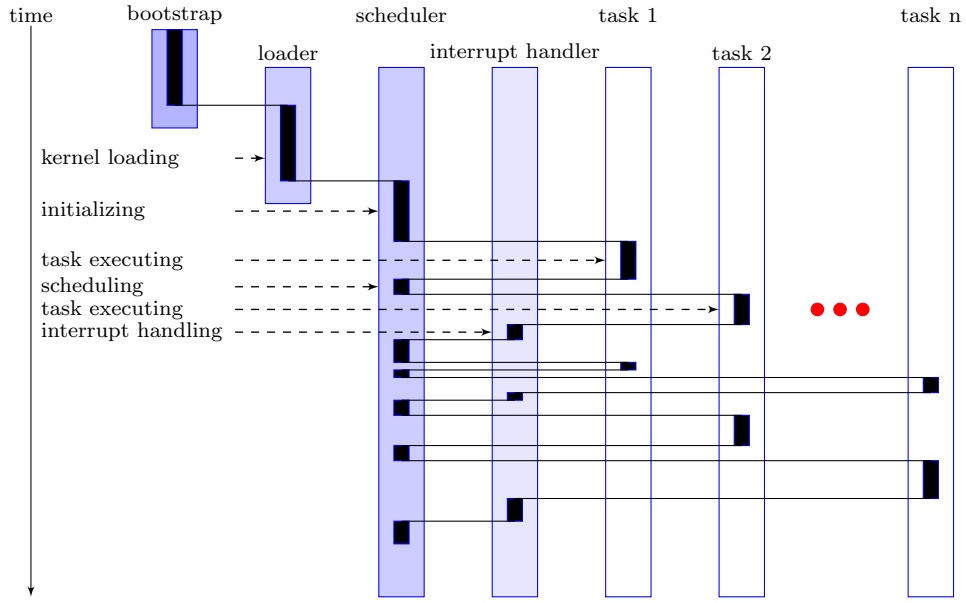


Figure 7.1 Task scheduling and the associated kernel activities

Figure 7.1 depicts the scenario of task scheduling on the processors (cores) of the host OS kernel. After the bootstrap and loading stages, the kernel scheduler initializes itself and then chooses a task for execution. The task runs until it meets certain blocking condition and cannot proceed. The reasons may be: 1) The task needs data from outside and stops to wait for that; 2) The task has to wait for a task with higher priority; 3) The task decides to yield the resource voluntarily; or 4) The task simply has finished its job.

With virtualization, the scheduling entity is a VM guest. Meanwhile, scheduling also occurs in the guest. This leads to a two-level hierarchical scheduling [133, 134, 135]. Figure 7.2 depicts a simplified imaginary scenario for the VCPU scheduling. A VCPU shares many qualities in common with a task, but it is also somewhat different by nature, due to the scheduling of guest tasks on the VCPU itself. Suppose that a multi-threading task, with $\{T_i|(i = 1, 2, 3)\}$, is running on the guest, being scheduled on the virtual processors. If synchronization is needed among these threads, T_0 on VCPU0 enters into a critical region and is holding thea lock, T_1 on VCPU1 is waiting for the release of this lock. In this case, VCPU0 may been preempted by another task with higher priority on the host or hypervisor. T_1 aquires the lock probably only the next time VCPU0 resumes its execution and gets the chance to release the lock.

In such condition the workload may suffer more performance loss than in the native execution. This is known as *lock holder preemption* (*LHP*) problem for processor virtualization [136, 137].

The *LHP* problem arises since the current hypervisor or OS are unaware of the tasks running on the VCPU, thus treats them as normal tasks to schedule. *gang-scheduling* [138] was proposed for this, with which all VCPUs involved in a thread synchronization are scheduled simultaneously

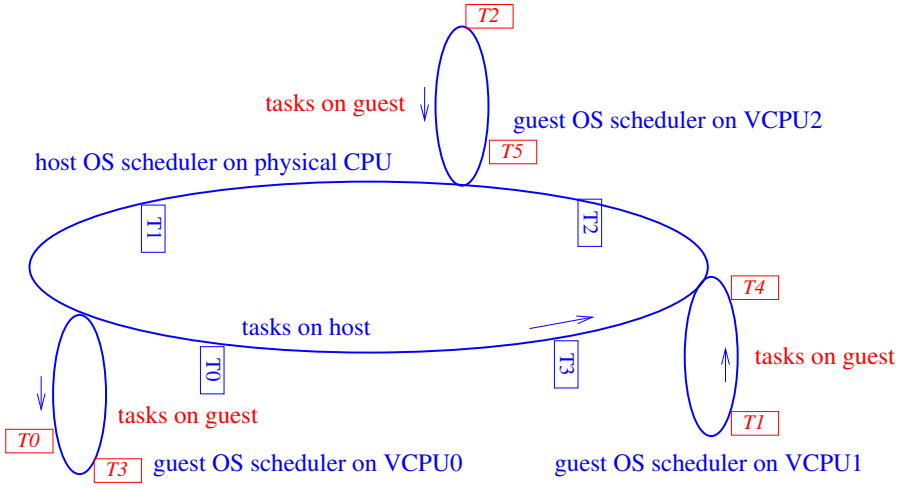


Figure 7.2 Scenario for VCPU scheduling

as a group on different physical processors (cores). As all related VCPUs run at the moment of synchronization, as the physical processors always do, theoretically the overhead should be eliminated. However, in practice, due to the negative effects such as *CPU fragmentation*, *priority inversion*, *VCPU stacking* and reduced utilization of resources [139], *gang-scheduling* has rarely been put into practical use in combination with virtualization.

VMware has illustrated how *co-scheduling*[2] is evolved with their product line in past decades. In the first generation (ESX Server 2.0), *co-scheduling* was used in the VMware bare-metal hypervisor, taking a rather rigid strategy by scheduling all VCPUs of a guest simultaneously. The hypervisor scheduler maintains a counter (the skew) for each VCPU of the multi-core guest. The skew grows when the associated VCPU lags behind the others until exceeding a threshold, at which point all VCPU would be stopped to prevent the skew from further growing. This leads to CPU fragmentation. In ESX Server 3.x this rigid version was replaced with an updated one known as *relaxed co-scheduling*. Instead of a single *co-start* or *co-stop* decision, each individual VCPU is now allowed to make this decision itself based on the comparison of its own skew against that of the slowest sibling. "By not requiring multiple VCPU to be scheduled together, the *co-scheduling* wide multiprocessor VM (those has more VCPUs than the underlying host) becomes efficient" [140]. In the subsequent ESX/ESXi server releases, *relaxed co-scheduling* was further "relaxed" and introduced more support for resource affinity, hyper-threading, and also exposed the host NUMA architecture to the guest. By doing this, the *relaxed co-scheduling* gains the capability of supporting "wide VM guest", meanwhile achieves more load balance.

The current schedulers of open-source hypervisors can also not handle this problem well. In Linux kernel there are rare efforts to improve the scheduler for QEMU-KVM. Although a patch for the *CFS* (*Completely Fair Scheduler*) has been developed for Linux kernel 3.2 [141], it is "highly experimental" and not recommended to be merged into the vanilla version. Although it could reduce the overhead due to *LHP*, it "could not establish all the positive aspects of the technique when the benchmarks were applied" [142].

balance scheduling [143] was proposed to cope with the problems left by *relaxed co-scheduling*, with which simply only one VCPU per processor core is run, without forcing all the VCPUs to be scheduled simultaneously. Compared with *co-scheduling*, *balance scheduling* can achieve similar (up to 8%) or better performance without the *co-scheduling* drawbacks (e.g. *CPU fragmentation*, *priority inversion*, *VCPU Stacking* and reduced resource utilization). Nevertheless, the *VCPU preemption* problem remains. A further improvement was suggested by [133, 144], which employs

[2]The concept of *co-scheduling* is similar to *gang-scheduling* in concept, but normally looser than the latter in selecting the set of tasks to be simultaneously scheduled. The latter requires all threads of the same process to run concurrently, while the former allows only a subset.

a mechanism to dynamically change the number of VCPUs to guarantee only one VCPU per physical processor core, and eliminate the need of VCPU scheduling from the hypervisor level. This way, as claimed by the authors, avoids all the negative sides of the former solutions. However, it relies on a module that performs the *VCPU ballooning* or *VCPU-hotpluging*, which may not be available for all OSes.

In summary, the performance loss due to processor virtualization can theoretically be a serious problem for the multi-threading applications, especially those involving frequent thread-synchronization on the multi-core physical machine. Normally this is very little and almost negligible for most of the computation-intensive applications.

7.2 Network Communication Aspect

7.2.1 Related work

Networking is the critical foundation for the message-passing based workloads on a distributed HPC system. Not only the communication between the processes on different computing nodes, but also the access to files on storage servers depends on the network. Since all the I/O traffic occur via network, the overhead of the network becomes a major limiting factor for the overall performance.

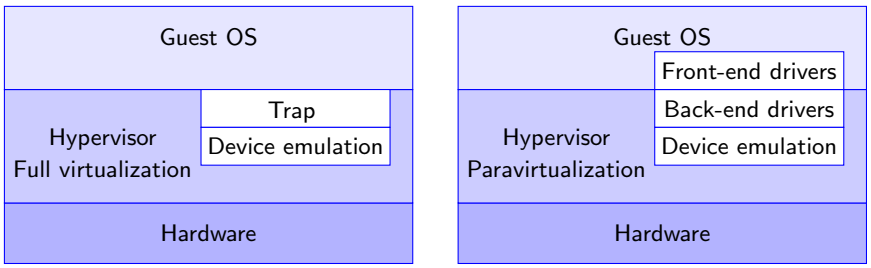


Figure 7.3 I/O device emulation in full and paravirtualization [145]

The virtualization of I/O devices, especially the network has witnessed the evolution through a few generations. At the beginning, as shown in Figure 7.3 (a), I/O virtualization was implemented by means of pure emulation by software. When an I/O operation was encountered, the guest saves the necessary information, such as the port number, operation type, data size for port I/O, or memory address for memory-mapped I/O etc. in a region accessible by the hypervisor and exits into the hypervisor (device emulator). The device emulator effects the operation either with the help of the physical devices, or simply by means of software. Finally, the results or feedback is returned back into the guest.

Emulation can achieve maximum portability and flexibility in terms of hardware-decoupling. However, the guest execution is constantly bothered by *vmexit* due to I/O device emulation, which is considered to be a potential source of performance loss for the I/O intensive workloads. Thus emulation technique is by nature not capable of providing high performance.

Hinted by paravirtualization, VirtIO [146] was developed to aid more efficient transition from guest to hypervisor. VirtIO is an abstraction layer for devices in a paravirtualized hypervisor. As Figure 7.3 (b) depicts, the guest OS is aware that it runs in a virtualized environment, and operates the drivers acting as the front-end. The other half, the drivers that act as the back-end are accommodated by the hypervisor. The front-end drivers abstract the common operations for a variety of specific devices in the guest.

With the front-end drivers, the command and data for an I/O operation issued by the guest are encapsulated as an entry in *ring buffer*s accessible by the back-end part in the hypervisor. When the *ring buffer*s are full, the request queue is emptied and the I/O tasks are dispatched

and performed by the back-end drivers to corresponding physical devices or emulator. The hypervisor responds to the request by putting the result into the buffer.

The paravirtualization approach, VirtIO has significantly simplified the device emulator hence the development of hypervisor. It reduces the frequency of *vmexit*, without sacrificing too much flexibility. Nowadays, VirtIO is the optimal I/O virtualization for most purposes, even compared with hardware-based I/O virtualization approach, the PCI-device pass-through, with which the guest is granted direct access to the physical I/O devices. PCI-passthrough can yield near native performance at the cost of much flexibility.

Many efforts were invested to reduce the performance loss from I/O virtualization, and significant results had been achieved. Further efforts [147] were made to speed up the VirtIO by assigning separate cores to host functionality dedicated to serving multiple guest's I/O. This is combined with a fine-grained I/O scheduling and exit-less notifications. It outperforms the baseline approach by 1.2 to 3 times, approaching or even exceeding the non-interposing I/O virtualization performance. *netmap* [148, 149, 150] enables the commodity operating systems to handle millions of packets per second via a 1 to 10 Gbit/s link without requiring custom hardware or changes to application. VALE [151], a virtual local ethernet that can be used by virtual machines such as QEMU-KVM and others, as well as normal processes, has achieved a speed over 17 million packets per second between host processes, and over 2 million packets per second between QEMU guests, without any hardware assistance. *ptnetmap* [152], an implementation of the pass-through mechanism for virtual network devices, can saturate a 10 Gbit link at a rate of 14.88 million packets per second, meanwhile removing the constrains of hardware pass-through. With VALE as a network back-end, and a slight modifications to the QEMU-KVM hypervisor, the host's virtual switch and device drivers, millions of packets can be transmitted per second for the *netmap* applications in virtual machines, a 20 times or more improvement in comparison to the baseline [153].

As the *interrupt handling* has an immediate impact on the I/O performance, an exit-less I/O virtualization [154] was implemented to boost the performance of the I/O operations in virtual machines combined with *PCI hardware pass-through*, yielding a 97% - 100% of bare-metal performance for the most demanding I/O intensive workloads [154]. Specific to the MPI mechanism, an efficient share memory message passing approach was created for inter-VM communication of MPI-applications [155]. It uses a virtual device which provides a simple message passing API to the guest OS. Benchmarks show near native performance in terms of the network bandwidth and latency.

7.2.2 Benchmark Selection for Inter-Node Pattern

Capable and well-tuned host delivers good single-node performance. Efficient network ensures excellent overall system performance if the task is well scalable across multiple computing nodes. To a certain extent, the inter-node pattern makes HPC workload fundamentally different from others. The inter-node pattern enables to solve large-scale problems by using more computing resources. While performance loss for virtualization in a single node is easy to determine, it is not so for network between virtual nodes. Possible reasons are:

- Program execution involves the joint-effort of all components in a computer. Performance loss may occur in each component due to virtualization, and the performance loss incurred by one component may overlap with that by other components, which makes it hard to divide the overall performance loss across the component;
- Performance of the network implies the data transmission speed between a pair of processes, either inside a physical intra-node, or between two physical nodes. The pair of processes act as server and client, respectively. While the performance of computation for intra-node can be easily calculated by comparing the performances of virtual and physical machines, performance of communication for inter-node is somewhat obscured by various allocation ways;

- The diversity and non-trivial building process of HPC workload makes it hard to select a set of representative real-world HPC applications and to organize them as an HPC benchmark suite for inter-node performance analysis.

For these reasons, it is helpful to select a bunch of the real-world HPC applications from various domains to form a benchmark suite. The selection criterion is based on the following considerations:

- Open source, freely available;
- Diversity, from various typical HPC application domains;
- Building must not be too complex, not too many dependencies on external libraries;
- Proper size of the test input data and appropriate execution time[3] of the workload;
- The executable binary can be explicitly run by `mpirun` or `mpiexec`, which ensures the control over the location and number of processes when a workload is started on a virtual cluster.

According to these criteria, 16 commonly used HPC applications were selected from various domains, such as quantum physics, molecular chemistry, weather forecast, environment, biology, financial mathematics and so on. The package is not yet a mature benchmark suite with unified tools for building and testing, but merely a group of binary executables with automated building by bash scripts. In case that the testing input data is too small, several benchmarks can be executed sequentially to yield longer execution time. These applications are useful for benchmarking the network of a computing cluster in inter-node pattern. The suite includes the following application:

Abinit This application focuses on atomic physics and chemistry. It calculates the electronic structure, total energy, and charge density of atomic systems (nuclei and electrons) using DFT (Density Functional Theory), plane waves, and pseudopotentials to solve *Schrödinger equation*s numerically [156]. Other usages include geometry optimization, molecular dynamics (MD), and many-body perturbation simulations.

CM1 It is used for atmospheric research, and designed for the studies of relatively small-scale processes in the Earth's atmosphere, such as thunderstorms [157]. CM1 is specially designed for distributed-memory systems, but can also run on shared-memory, or hybrid OpenMP/MPI systems [158].

Quantum Espresso This application is an integrated software suite for atomistic simulations based on electronic structure, using density-functional theory, plane waves and pseudopotentials. The acronym Espresso stands for opEn Source Package for Research in Electronic Structure, Simulation, and Optimization. It is freely available to researchers under the GNU General Public License terms [159]. The Quantum Espresso codes work on various types of UNIX machines, including parallel systems with OpenMP, MPI, and GPU-acceleration [160].

GROMACS This application is a high-end, high-performance software designed for the study of protein dynamics using classical molecular dynamics theory [161]. Its package is available under the GNU General Public License terms. The code runs on UNIX, Linux, and Windows. As the benchmark involves an extra regression testing package, the execution is not tuned as usual by giving parameters in an MPI command line.

HPCC A collection of benchmark programs used to measures a range of memory access patterns. It contains basically 7 testing programs, namely, HPL to measure the number of floating-point operations for solving linear system of equations; DGEMM to measure the number of double precision floating-point operations for real matrix-matrix multiplication; STREAM to measure the sustainable memory bandwidth and computation for simple vector kernel; PTRANS to measure the performance of network between a pair of processors; RandomAccess to measure

[3]Too short execution can not ensure sufficient precision, and too long execution takes too much time.

the rate of integer random update of memory (GUPS); FFT to measure the double-precision floating point operations for one-dimensional DFT (discrete fourier transform); and a number of tests to measure the bandwidth and latency for a few communication patterns [162, 163].

HPL This is the High-Performance Linpack benchmark package used as the standard tool to evaluate the performance of a computing system (especially a supercomputer). It measures the time of solving a uniformly random system of linear equations, and the floating-point operations by a standard formula. It is witten in C and parallelized by MPI, thus can be scaled across multiple computing nodes [164].

LAMMPS An open-source tool developed by Sandia National Labs for molecular dynamic studies. LAMMPS stands for Large-scale Atomic/Molecular Massive Parallel Simulator. It is rich in feature and functionality, but can still be extended by users. The platform can be a single processor, or a distributed-memory system supporting message-passing [165].

MILC A set of C programs developed by the MIMD Lattice Computation (MILC) collaboration for doing simulation of four-dimensional SU(3) [166] lattice gauge theory on MIMD parallel machines. It can perform large-scale numerical simulations to study quantum chromodynamics (QCD) [167, 168]. MILC supports parallelization by making use of multi-core, message-passing and graphic processor unit, on machines with a variety of architectures.

mpiBLAST In biology and medicine, a growing need is to process genomic data. An example is the GenBank, which has a huge capacity, and is keeping on doubling itself roughly per 18 months. The basic problem is to query the GenBank, which involves comparing a given sequence with a large set of sequences in the GenBank, basically a one-to-many alignment operation. BLAST (basic local alignment search tool, also known as NCBI BLAST) is developed for querying large sequence databases. It works well for small number of queries in small databases. Thus, BLAST gets parallelized to deal with a large number of queries and rapidly growing database. Two variants are mpiBLAST [169] and ScalaBLAST [170]. The former can improve the performance of BLAST almost linearly by adopting fragmented database and query, intelligent scheduling, and parallel I/O. It supports many ISAs and operating systems. (Source: [171, 172])

MrBayes *Phylogenetic Systematics* is a branch of biology to classify organisms by using phylogenetic methods [173]. A phylogeny is a hypothetical relationship between groups of organisms being compared. Phylogeny is often depicted by a phylogenetic tree to describe the evolutionary relationships between different genera [173]. MrBayes is an application to aid study in this area. It performs Bayesian inference of phylogeny using a variant of Markov chain Monte Carlo (MCMC) techniques. MrBayes-3 uses MPI for parallel execution. As the communicated among the multiple chains is not intensive, near linear speedups can be achieved. (Source: [174])

Nek5000 *Computational Fluid Dynamics* is a branch of fluid dynamics providing a cost-effective means of simulating real flows by numerical solution of governing equations. Nek5000 is developed to aid the study of CFD with parallel computing systems. It can be used in a variety of applications including vascular flow, heat transfer, combustion, ocean modeling, fundamentals of turbulence, astrophysics, accelerator physics, and nanophotonics. The programs are written in F77, C and parallelized purely by message-passing approach. Its execution is highly scalable, ranging from single-processor laptop to most powerful supercomputers of the world. The prove scalability was shown with over a million MPI ranks running the application. (Source: [175, 176, 177])

Octopus In material sciences, computation is needed to solve the partial differential equations (PDE) modelled by applying the density-functional theory (DFT) and its time dependent variant - TDDFT. Based on quantum mechanical theory, the *ab initio* (means from the beginning) DFT calculations can predict the material behaviour without requiring higher order parameters such as fundamental material properties. DFT and TDDFT are widely adopted in material sciences, solid physics and chemistry to study the electronic structure of many-body systems, particularly atoms, molecules, and the condensed phases. Octopus is an application for solving the time-dependent *Kohn-Sham* [178] PDEs based on the DFT and TDDFT by numerical simulation. It

is highly parallelizable. With MPI and OpenMP, its execution can scale to tens of thousands of processors. Support for graphical processing units (GPUs) through OpenCL and CUDA is also available. Octopus is freely released under the GPL license. (Source: [179, 180, 181, 182, 183])

OpenAtom In science and technology branches such as chemistry, solid physics, bio-physics, geophysics, electronic physics, material sciences, a growing need is to simulate the molecular and atomic systems based on the principles of quantum chemistry. The vast amount of entities and complex interaction in such systems limit the accuracy, computational efficiency and applicability of today's supercomputers. Currently, only a few hundred atoms can be simulated at such level of detail due to the communication-intensive Fast Fourier Transformations (FFTs). Charm++ is a parallel programming framework to decompose a task in a way natural to the application domain. OpenAtom is able to make use of Charm++'s asynchrony and object-based overdecomposition approach to overcome these challenges. OpenAtom uses the Car-Parrinello Ab Initio Molecular Dynamics (CPAIMD) approach, which allows to study complex atomic and electronic physics in semiconductor, metallic, biological and other molecular systems. It can scale to thousands of cores for realistic scientific systems with only a few hundred atoms. Such excellent scalability enables it to use the parallelism via fine grains of data and computation. (Source: [184, 185, 186])

Tachyon Many scientific areas demand high-quality rendering of three-dimensional geometry and vector fields. Algorithms and software tools have been designed to render such photorealistic images of according objects. However, since image rendering is a computation-intensive task, high-quality three-dimensional images are generated not quite efficiently by sequential algorithm and software tools. By making image rendering process parallel, more than two orders of magnitude of time cost can be saved. Ray-tracing [187] is an approach among the image rendering techniques. It simulates rays of light in producing images, sped up by parallelizing the ray-tracing process. It runs on both distributed- and shared-memory systems. Tachyon is used in visualizing molecular dynamics (VMD) and serves as a benchmark for parallel systems in this domain. (Source: [188, 189])

Wombat In biology, a *phenotype* is the observable trait presented in an organism. It may include the observable structure, function and behavior. *Quantitative Genetics* is a branch of *Population Genetics* that treats the quality of phenotypes as continuous variable quantities. This makes it possible to use mathematical statistical approaches to study the connection between genes and their physical manifestations. WOMBAT is a software package for qualitative genetic analyses of continuous traits, fitting a linear, mixed model via restricted maximum likelihood (REML). It can be used to analyze large data sets from livestock industry and simulate data for a given data and pedigree structure. WOMBAT is distributed in the form of pre-compiled binary code, and can be executed in parallel by using MPI. (Source [190, 191, 192])

WRF In weather research and forecasting, WRF is a set of software tools for atmospheric research and operational forecasting by means of numerical weather prediction (NWP) . The latter is an approach to predict the weather by using mathematical models and numerical computations. For this enormous amount of data collected by observing networks must be processed rapidly and accurately. Supercomputers are able to provide such computing power. WRF uses a fully compressible and nonhydrostatic model (with a run-time hydrostatic option). *Runge-Kutta* time integration schemes, and advection schemes are applied in both horizontal and vertical dimensions. WRF runs on both shared- and distributed memory systems, and has excellent scalability. (Source: [193, 194])

In summary, the benchmark application domains include: physics, chemistry, atmospheric and weather research, electronics, molecular dynamics, fluid dynamics, biology, genetics, material sciences, and image rendering. HPCC and HPL are used as single benchmarks in this context.

7.2.3 Benchmarking for Network Communication

Assuming virtual machines are deployed in a supercomputer, each computing node may host a few guests. In such an environment, different ways exist to run the MPI-based workloads. When execution occurs on the virtual platform, the workload can be run in the intra-node pattern on a guest, or in the inter-node pattern on many guests belonging to a virtual cluster. Similar to the conventional cluster, a virtual cluster is a collection of mainly virtual computers interconnected networks. Figure 7.4 shows a few virtual clusters residing on the physical nodes of a cluster. Normally the computing nodes are virtual machines, however, physical machine can also be used as a node for computing or more commonly for management.

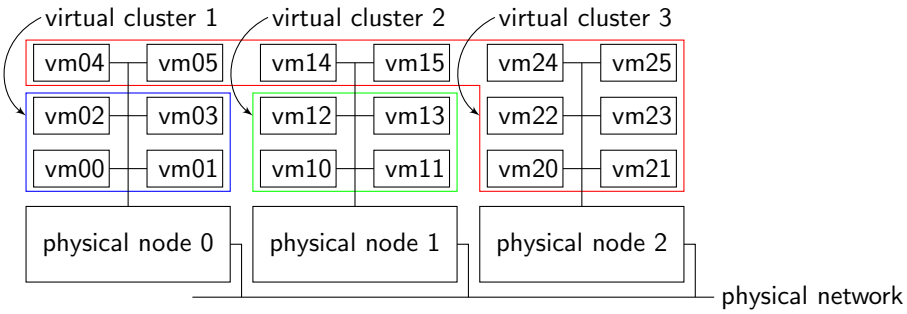


Figure 7.4 Virtual clusters [10]

Figure 7.5 depicts the construction and configuration of an example benchmark platform for such a network. It is a cluster consisting of one management node and two computing nodes, connected by Gigabit Ethernet network. The system software is the same as specified in Section 3.3. All the benchmark applications are stored in an NFS partition visible by all physical and virtual nodes in the cluster. Each physical computing node may host a progressively increasing number of identical guests as the virtual nodes.

The performance loss over network cannot be directly calculated, but estimated by comparing the performances yielded in a number of cases. The performance loss for intra-node pattern serves as the basis of comparison, as it does not involve any overhead incurred by the physical network. Furthermore, performance data yielded in a number of virtual cases are also necessary. In a virtual cluster, there are various ways for running an MPI-based workload. Both virtual and physical nodes can be allocated. A workload with two MPI processes, for example, can be scheduled in four ways, as depicted in Figure 7.6. For convenience, they are labeled as PsnPsn, PdnPdn, VsnVsn, VdnVdn, respectively, where P stands for physical node, V for virtual node, s for shared-memory, d for distributed-memory, and n for the number of MPI process allocated on a single node. The number of processes each node runs, and the virtual nodes each physical node hosts may also increase for more MPI processes. These schemes represent the basic patterns for scheduling an MPI-based HPC workload in a virtual cluster, and remain unchanged.

Due to the optimization for inter-VM communication on the same physical node, the VsnVsn may take the advantage of memory-sharing for inter-process communication. The PsnPsn does the same at less cost. Performance comparison between these two reveals more about the performance loss due to virtualization than that due to network communication. On the other hand, both VdnVdn and PdnPdn involve inter-process communication over real physical network. Performance comparison between them therefore reflects the impacts of both network and virtualization, and serves as a criterion for determining whether a workload could be migrated into a virtual cluster with distributed memory. Similarly, both VsnVsn and PsnPsn involve no communication over real physical network, thus the performance comparison determines whether a virtual cluster with shared memory is suitable for this type of workload.

The performance comparison between PdnPdn and PsnPsn mainly reveals the difference of distributed and shared memory, so does the comparison between VdnVdn and VsnVsn, but combined with the difference due to virtualization.

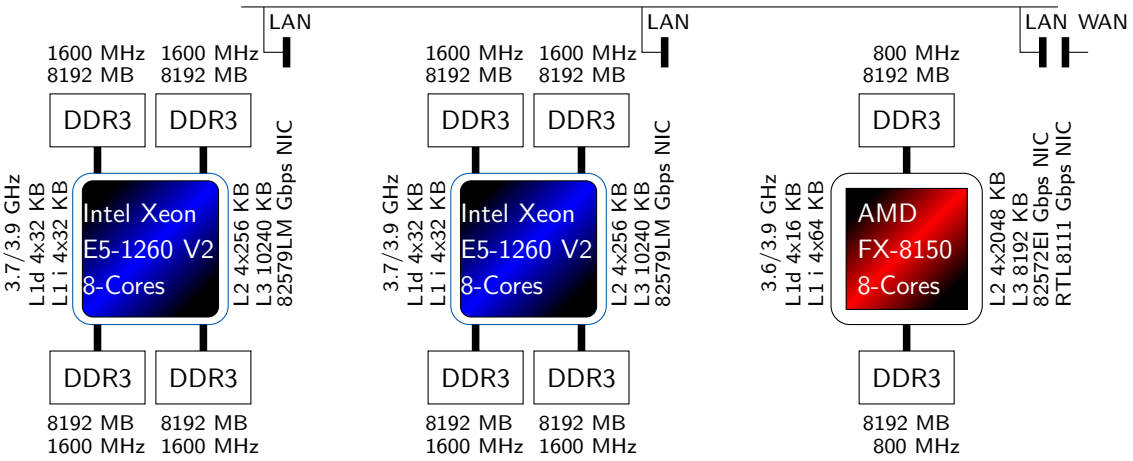
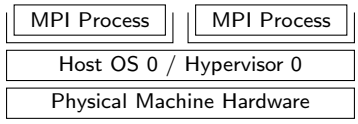
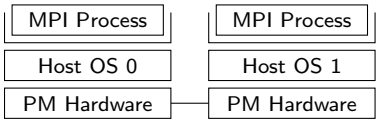


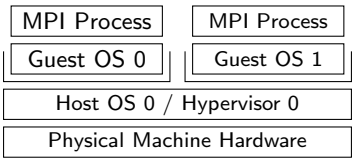
Figure 7.5 Cluster used for benchmark



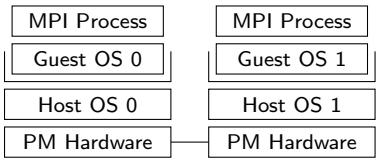
(a) PsnPsn



(b) PdnPdn



(c) VsnVsn



(d) VdnVdn

Figure 7.6 Basic ways to run an MPI application on (a) the same physical node; (b) different physical nodes; (c) different virtual nodes but the same physical node; (d) different virtual nodes and different physical nodes.

Figure 7.7(a) presents the benchmark results for the selected real-world workload executed as one, two, four and eight MPI processes, respectively[4]. Figure 7.7(b)(c) illustrate the performance comparisons between the above mentioned various cases for 2 and 4 MPI processes, respectively.

Due to the requirements for some of the workloads, execution may not be possible in some cases, thus missing data exist in the figures. The available data reveals the performance loss for a batch of real-world HPC workloads over the network for both physical and virtual clusters.

Figure 7.7(a) reveals the following: 1) 11/16 workloads reached 90% of native performance regardless of the number of MPI processes; 2) 14/16 reached 80% of native performance regardless of the number of MPI processes; 3) 2 workloads (`octopus` and `wombat`) lose 20% or more of native performance as the number of MPI process increases.

Figure 7.7(b)(c) reveals the following: 1) In the case of 2 MPI processes, 8/12 workloads are above 90%, 11/12 above 80% of native performance for Vd1Vd1 scheme (Vd1Vd1/Pd1Pd1); 5/12 are above 90%, 7/12 above 80% of native performance for Vs1Vs1 scheme (Vs1Vs1/Ps1Ps1); 2) In the case of 4 MPI processes, 10/14 are above 90%, 12/14 above 80% of native performance for Vd2Vd2 scheme (Vd2Vd2/Pd2Pd2); 8/14 are above 90%, 9/14 above 80% of native performance for Vs2Vs2 scheme (Vs2Vs2/Ps2Ps2).

[4]The missing items are due to various reasons: `HPCC`, `HPL` and `mpiBLAST` can only be executed by at least 4 MPI processes; `octopus` and `wombat` yield increasingly poor performance when run as multiple MPI processes, thus makes little sense in those cases.

Table 7.1: Performance Loss over Communication Network (normalized)

Workload	Vs2	Vs4	Vd1Vd1	Vs1Vs1	Vd2Vd2	Vs2Vs2
abinit	0.04	-0.07	0.07	-0.16	0.11	-0.02
cm1	0.13	0.12	0.18	0.16	0.16	-0.10
espresso	0.05	-0.08	-0.09	0.36	-0.08	0.56
gromacs	0.02	-0.02	-	-	-	-
hpcc	-	-0.07	-	-	-0.68	-0.43
hpl	-	-0.15	-	-	0.02	0.24
lammps	0	0.08	0.12	0.73	0.12	0.75
milc	0.05	-0.08	-0.23	0.29	-0.03	0.54
mpiblast	-	0.06	-	-	0.02	0.06
mrbayes	0.02	-0.03	0.01	0.02	0.01	-0.04
nek5000	0.10	-0.07	-	-	-	-
octopus	0.38	0.39	0.07	-	-	-
openatom	0.09	-0.10	0.08	0.09	0.04	-0.02
tachyon	0.06	-0.16	0.03	0.01	0.03	-0.24
wombat	0.24	-	0.15	0.20	0.42	0.17
wrf	0	-0.03	0.02	0	0.01	0.08

As a summary of the discussion about the benchmark result, Table 7.1 lists the normalized performance loss reasoned by the communication network. The first two columns show the performance loss incurred purely by virtualization (intra node), corresponding to Figure 7.7(a). The other four columns correspond to Figure 7.7(b) and (c). A noticeable thing is the negative performance loss, which indicates that virtual machine outperforms physical machine. Though theoretically this is impossible, it seems not to be so uncommon when benchmarking a virtual machine. An assumption is that over-clocking of the physical processor could have been triggered. In these cases, the results may be distorted to a certain extent, and cannot be compared bindly. However, for completeness, they are still presented here.

For entries with comparable data, the performance loss of the network is estimated by subtracting the value of Vs2 from Vd1Vd1 for the case of two processes, and Vs4 from Vd2Vd2 for the case of four processes. A few examples are: `abinit`: 3%, `cm1`: 4% to 5%, `wrf`: 2%, and `lammps`: 4% to %12. Considering that the Gigabit Ethernet is used as the network interconnection, the result does not account for serious performance loss.

7.3 Future Work

7.3.1 Future Work for Memory Virtualization

Currently, all functional units of the dynamic paging method switching are implemented based on the KVM and named as DPMS. Although DPMS meets the basic requirements for design and exhibits its potential of improving the hypervisor's performance, there are still a few problems unsolved in the implementation. To get DPMS outgrown from an experimental research project, and taken serious as a practically useful hypervisor component, the following problems must still be solved:

Problem on AMD Platform

As an important variant of the x86-64 architecture, AMD has a similar hardware extension for virtualization support as Intel. However, several subtle differences also exist between the AMD NPT and Intel EPT solutions, which tend to pose problems for developing hypervisors and migrating virtual machines. Features making use of the hardware virtualization extension on x86-64 needs two variants. This is also the case for DPMS.

Problem on Intel Platform

The occasional crash on Intel platform is another issue bothering DPMS. Hinted by the initial analysis, the two types of errors are all related to MMIO region. A basic assumption is that the MMIO region in the guest memory has been corrupted due to the changing of page tables. This may be caused by a mistake, or inappropriate actions in the switching operation. However, to find out the reasons, deeper investigation of the implementation is still needed.

Improving on Decision-Making

A few thresholds used for *Decision Making* (see Section 5.3.3) are determined by analyzing the statistics collected by the PMCs, and the rules are also derived by observation. Although it works fine, it can be applied only for handling a small number of workloads and data size. An ideal approach would be machine-learning. The thresholds and rules can be generated from a large data set, which leads to more reliable and generic results.

Support for Multi-core

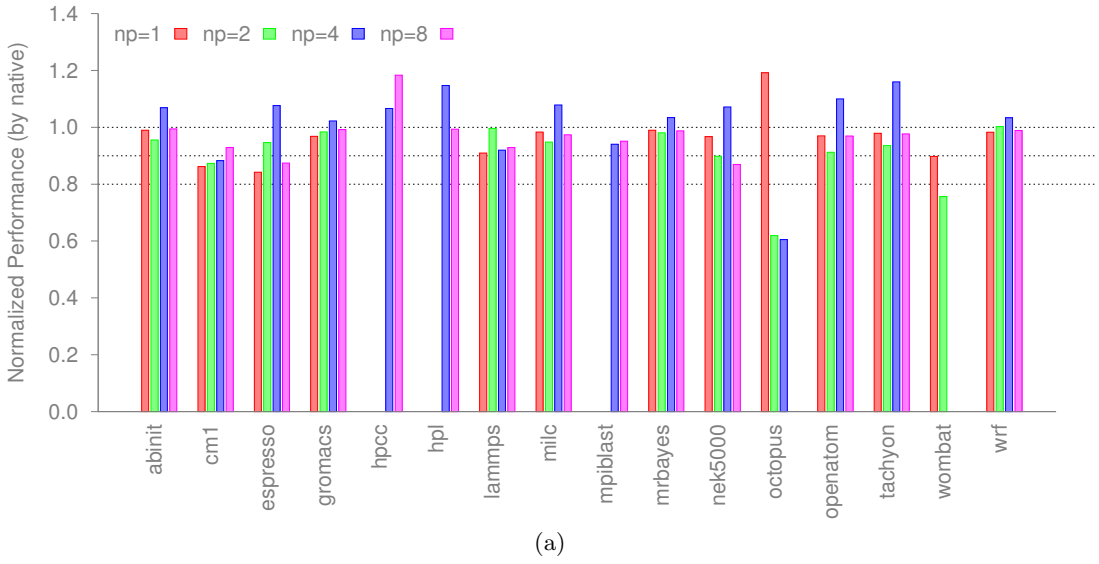
The current DPMS makes use of a single core. A challenge for supporting multi-core lies in the difficulty to control the VCPUs of a guest in an appropriate synchronized manner. However, this may harm the performance and offset the benefits gained by the dynamic switching.

The Impact of Cache and TLB on STDP's Performance

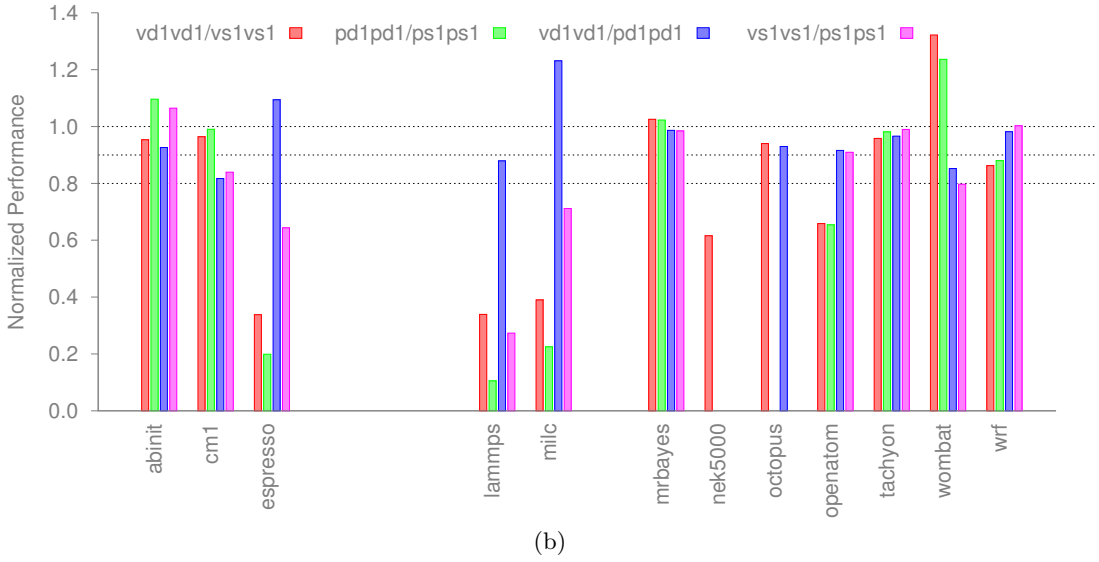
With the increasing of the TDP table size, the performance impact of the cache and TLB needs to be investigated in future.

7.3.2 Future Work for I/O Virtualization

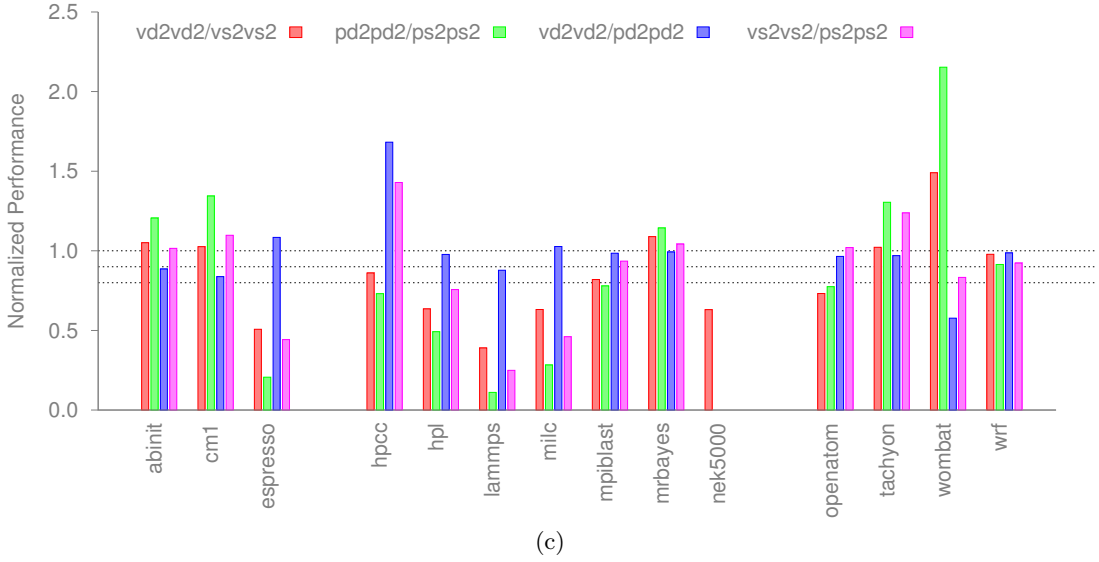
More efforts are needed to study the performance loss of network for the MPI-based workload, with high-end NIC such as Infiniband. This area attracts more focus to imporve the performance of virtualization for HPC workload.



(a)



(b)



(c)

Figure 7.7 Performance comparisons (a) without real network communication, (b) with physical network communication between 2 MPI processes, (c) with physical network communication among 4 MPI processes

Chapter 8 Conclusions

With the aim of improving the performance of HPC workloads in virtual execution environments, the dissertation initially made a comprehensive study of the performance by benchmarking the typical real-world HPC applications in virtual machines. The benchmark results indicate that MPI-based workloads in a cluster environment suffer less significantly than the multi-threading HPC workloads in a single computing node. Therefore, the efforts for improving the performance of the hypervisor is focused on the individual components of a single node. In this case, further benchmarks showed that the virtualization of the memory management system may pose as a serious source of performance loss for specific types of HPC workloads. This problem arises partially due to the respective strengths and weaknesses from the current approaches for virtualizing the memory management system.

For a given processor and I/O hardware, the performance can only be improved by dynamically exploiting the advantages of each paging approach. However, most of the current main-stream hypervisors apply static way to configure the paging method prior to the execution of guest systems, thus use the same paging method once for all to handle different types of workloads. Based on these observations, the thesis proposes the approaches - DPMS and STDP.

DPMS is practically a software approach, which adds a capability in the hypervisor to exploit the best qualities of the two standard approaches at present. By analyzing workload-specific information at run time, the hypervisor is able to adjust the paging method periodically to avoid certain large overhead. The main concepts realized in DPMS include:

1. **Effective use of the performance monitor counters:** The performance monitor counters are able to capture specific run-time statistics about the interaction between the workload and the execution environment. Useful information can be obtained by analyzing these data, which may in turn aid the hypervisor to be more adaptive for avoiding high performance loss.
2. **Reduction of misprediction and unnecessary switching:** A main problem of the predictive model is the misprediction. Although the benefits of DPMS is obvious, in the cases of misprediction, the performance cost is expensive. DPMS suffers from this problem, but solves it by adopting reasonable rules to determine the situation in which the paging method really needs a switching. These rules are deduced from the data sampled by the performance monitor counters. By applying these rules, the decision is less likely to be disturbed by random fluctuation of the related performance metrics. In other words, a switching is triggered for more deterministic reasons.
3. **Reconfiguration of the paging method:** The dynamic switching of paging method has advantages over the current static way to configure the paging method. The advantage lies in the flexibility to switch between the two page methods dynamically. The switching is performed by modifying not only the components of hypervisor, but also a few registers in the physical processor. The challenge is to determine the minimum set of operations that guarantee the proper function and minimize the impact on the original code. With the de-configuration of the current paging method, the page tables are partially or fully discarded. By configuring the new paging method, new page tables are incrementally prepared for subsequent operations.

4. **Balance between sensitivity and stability:** Sensitivity and stability are conflicting targets. In the design and implementation of DPMS, the balance between them boils down to the choice of a sampling frequency. Although theoretically the relation is obvious, it is more a matter of empiricism than of mathematical calculation. Due to insufficiency in the initial implementation, a certain degree of sensitivity had to be sacrificed by keeping the sampling frequency reasonablly lower, which ensures that the PM-sensitive workloads are handled by their nature. Meanwhile, the PM non-sensitive workloads is less likely affected due to switching.

By applying these concepts, the dynamic paging method switching is implemented based on KVM. In comparison to the current static approach, DPMS illustrated its flexibility to cope with workloads of different types. Based on the benchmark result, the conclusions are:

1. DPMS yields a mixed results of both positive and negative.
2. DPMS can speed up some specific workloads by a few percent.
3. DPMS can outperform the NPT/EPT, but may not exceed the SPT in these cases.

STDP is an approach depending on both the software and hardware, with the aim to reduce the paging overhead by applying fewer paging levels in the second dimensional page tables for workloads which are not good at taking the advantage of TLB. Theoretically, 40% of the paging cost can be reduced for the translation from the guest virtual to host physical address without considering the TLB effect. The main concepts realized by STDP include:

1. **Reducing the paging overhead by restructuring the page tables:** The starting point is that the overhead for n-level paging with the second dimensional page tables has a complexity of $\mathrm{O}(n^2)$, and if the overhead in the second dimension decreases by $\mathrm{O}(n)$, the total overhead will decrease by $\mathrm{O}(n^2)$. If the 2-level page table is adopted in the second dimension, ten times of memory accessing (10/24 of the total cost) can be saved. In STDP, multiple 4-KB TDP tables are concentrated into a single 2-MB table.
2. **Separation of the page table meta data from page table content:** This leads to the necessity of reorganizing the page tables and the control information. A particular challenge lies in the maintenance of the mapping relation between a page table and its meta data. In the implementation of STDP, an array is used for this purpose. The index is calculated from the address to be translated.
3. **Hardware for parsing the restructured page tables:** To correctly parse the restructured page tables, the hardware, more precisely the MMU also needs to be modified. The physical MMU must be adaptive to traverse the traditional 4-level guest page tables with four steps, but the restructured 2-level TDP tables with two steps.

The software part of STDP has been implemented. With these concepts, the nested page tables are concentrated in the 2-MB pages, yielding a decreased walking length. The page tables can be allocated and managed as expected.

Bibliography

[1] B. Parhami. *Computer Architecture - From Microprocessors to Supercomputers*. pp. 468, pp. 59, Oxford University Press, Inc, 2005.

[2] J. Dongarra, P. Luszczek, and A. Petitet. *The LINPACK Benchmark: past, present and future*. Concurrency and Computation: Practice and Experience, pp. 803-820, 2003.

[3] R. Nambiar and M. Poess. *Transaction Performance vs. Moore's Law: A Trend Analysis*. TPCTC 2010, LNCS 6417, pp. 110-120, Springer-Verlag Berlin Heidelberg, 2011.

[4] *A History of Microprocessor Transistor Count*. Wikipedia, Aug. 29, 2013.

[5] M. Feldman. *Moore's Law Is Breaking Down, But That's OK*. https://www.top500.org/news/moores-law-is-breaking-down-but-thats-ok Mar. 2017.

[6] *Top 500, The List*. https://www.top500.org May. 2017.

[7] *Three Gorges Dam Project, Yangtze River, China*. p140, P. H. Gleick, Nov, 2008.

[8] *The Green 500*. https://www.top500.org/green500/lists/2016/11 May. 2017.

[9] A. Radonic and F. Meyer. *XEN3*. p67, Franzis Verlag GmbH, Poing, 2006.

[10] K. Hwang, J. Dongarra, and G. Fox. *Distributed and Cloud Computing - From Parallel Processing to the Internet of Things*. 1st Edition, Morgan Kaufmann, pp. 4, Oct. 2011.

[11] G. J. Popek and R. P. Goldberg. *Formal Requirements for Virtualizable Third Generation Architectures*. Communications of the ACM, 17(7): pp. 412-421, Jul. 1974.

[12] *Virtualization*. https://en.wikipedia.org/wiki/Virtualization Wikipedia, Apr. 2017.

[13] P. Shenoy. *Introduction to Virtualization*. Lecture slides, University of Massachusetts, Sep. 2007.

[14] J. E. Smith and R. Nair. *Virtual Machines - Versatile Platforms for Systems and Processes*. pp. 416, 32, 35-38, 369, 405, Morgan Kaufmann, Elsevier, San Francisco, 2005.

[15] M. Rosenblum. *The Reincarnation of Virtual Machines*. ACM QUEUE (07/08). Jul. 2004.

[16] J. Mears. *The 8 key challenges of virtualizing your data center*. http://www.networkworld.com/article/2295569/data-center. Feb. 2007.

[17] F5 Virtualization Solutions. *7 Virtualization Challenges: Building a Virtualization-ready Application and Storage Networking Infrastrcture*. Sep. 2008.

[18] J. Metzler. *Virtualization: Benefits, Challenges and Solutions*. Riverbed Technology White Paper. Dec. 2011.

[19] *Citrix XenServer ® 7.3 Virtual Machine User's Guide*. Citrix Systems, Inc. Dec. 2017.

[20] *KVM: Kernel-based Virtualization Driver*. Qumranet white paper. Nov. 2006.

[21] A. Finn, M. Luescher, P. Lownds, and D. Flynn. *Windows Server 2012 ® Hyper-V Installation and Configuration Guide*. Sybex, John Wiley & Sons, Inc. 2013.

[22] P. Bright. *Will VMware's new licensing scheme open the door for Microsoft? - The new pricing scheme that VMware has announced for vSphere 5 has many ....* http://arstechnica.com/business/2011/07/what-will-the-vmware-vsphere-5-licensing-changes-mean-for-you. Jul. 2011.

[23] S. Hanselman, *Here's a Great List of Tips on Optimzing Performance with Virtual PC (VPC)*, http://www.hanselman.com/blog, Feb. 20, 2004.

[24] W. Huang, J. Liu, B. Abali, and D. K. Panda. *A Case for High Performance Computing with Virtual Machines*. Proceedings of the 20th ACM Annual International Conference

on Supercomputing, pp. 125-134. 2006.
[25] Q. Ali, V. Kiriansky, J. Simons, and P. Zaroo. *Performance Evaluation of HPC Benchmarks on VMware's ESXi Server*. 5th Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2011), Aug. 2011.
[26] A. Reuther, P. Michaleas, A. Prout, and J. Kepner. *HPC-VMs: Virtual Machines in High Performance Computing Systems*. IEEE-HPEC 2012, Sep. 2012.
[27] *Virtualizing HPC and Technical Computing with VMware vSphere*. Technical white paper of John Hopkins University Applied Physics Laboratory, Feb. 2016.
[28] A. Nanos, N. Nikoleris, S. Psomadakis, E. Kozyri, and N. Koziris. *A Smart HPC interconnect for clusters of Virtual Machines*. Euro-Par 2011: Parallel Processing Workshops. Lecture Notes in Computer Science, vol 7156. Springer, Berlin, Heidelberg, 2012.
[29] N. Huber, M. von Quast, F. Brosig, and S. Kounev. *Analysis of the Performance-Influencing Factors of Virtualization Platforms*. On the Move to Meaningful Internet Systems, OTM 2010, Proceedings, Part II, pp. 811-828. Oct. 2010.
[30] P. Shenoy. *Server Design, Code Migration*. Lecture slides, University of Massachusetts. Feb. 2008.
[31] V. Moya del Barrio. *Study of the Techniques for Emulation Programming*. pp. 15, Computer Science Engineering - FIB UPC, Jun. 2001.
[32] Y. N. Cui, J. M. Pang, Z. Shan, and F. Yue. *Register one-to-one Mapping Strategy in Danymic Binary Translation*. Network Security and Communication Engineering: Procceedings of the 2014, pp. 201-204, 2014.
[33] D. Ung and C. Cifuentes. *Machine-Adaptable Dynamic Binary Translation*. Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, Vol. 35.7, pp. 41-51, Jan. 2000.
[34] C. May. *MIMIC : A FAST SYSTEM/370 SIMULATOR*. Proceedings of the ACM SIGPLAN'87 Symposium Interpreters and Interpretive Techniques, pp. 1-13, Jun. 1987.
[35] K. Adams and O. Agesen. *A Comparison of Software and Hardware Techniques for x86 Virtualization*. ASPLOS'06, Oct. 2006.
[36] I. Pratt, K. Fraser, S. Hand, C. Limpach, and A. Warfield. *Xen 3.0 and the Art of Virtualization*. Linux Symposium, 2006.
[37] *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*. VMware White Paper, Oct. 2007.
[38] J. Rushby. *Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance*. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, Oct. 1999.
[39] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber. *A Comparison of Partitioning Operating Systems for Integrated Systems*. Computer Safety, Reliability, and Security (2007): pp. 342-355., 2007.
[40] A. S. Tanenbaum, A. Woodhull, and A. S. Woodhull. *Operating Systems: Design and Implementation*. 2nd Edition, pp. 319, Prentice Hall, Upper Saddle River, NJ, 1997.
[41] S. Dwarkadas. *Address Translation for Virtual Machines*. Lecture slides, CS456, University of Rochester. Dec. 2013.
[42] D. A. Godse and A. P. Godse. *Microprocessors & Microcontrollers*. Third Revised Edition, pp. 4-28, Technical Publications Pune, Jan. 2008.
[43] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol.3a, 2-15, System Programming Guide, Part 1, Intel Corporation, Sep. 2016.
[44] Adavanced Micro Devices, Inc, *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. pp. 146, pp. 273, Adavanced Micro Devices, Mar. 2017.
[45] *Intel 64 and IA-32 Architectures Software Developer's Manual*.

Vol.1, 11-24, Basic Architecture, Intel Corporation, Dec. 2015.
[46] Adavanced Micro Devices, Inc, *AMD-V Nested Paging*. White Paper, Adavanced Micro Devices, Jul. 2008.
[47] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol.3c, 28-2, System Programming Guide, Part 3, Intel Corporation, Sep. 2016.
[48] J. F. Kloster, J. Kristensen, and A. Mejlholm. *A Paravirtualized Approach to Content-Based Page Sharing*. Department of Computer Science, Aalborg University, Jun. 2007.
[49] B. D. Payne, M. D. P. de A. Carbone, and W. Lee. *Secure and Flexible Monitoring of Virtual Machines*. Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual. IEEE, 2007.
[50] F. Baiardi and D. Sgandurra. *An Obfuscation-Based Approach against Injection Attacks*. Availability Reliability and Security 2011 Sixth International Conference,p51-58, 2011.
[51] Wikipedia *X86 Paravirtualised Memory Management*. http://wiki.xen.org/wiki/X86_Paravirtualised_Memory_Management, Mar. 2013.
[52] *Intel Virtualization Technology for Directed I/O*. Architecture Specification, Revision 1.1, Intel Corporation, Sep. 2007.
[53] M. Mahalingam. *I/O Architectures for Virtualization*. Slides, VMWorld, Nov. 2006.
[54] *Planning Guide Virtualization and Cloud Computing - Steps in the Evolution from Virtualization to Private Cloud Infrastructure as a Service*. Intel IT Center, Aug. 2013.
[55] M. Vaughn. *Key differences between virtualization and cloud computing - Is virtualization cloud computing?*. White Paper, SearchServerVirtualization.com, Jun. 2011.
[56] S. P. Ahuja. *Virtualization for Cloud Computing*. Slides, School of Computing, University of North Florida, Jan. 2015.
[57] J. Kremer. *Cloud Computing and Virtualization*. White Paper, Jan. Kremer Consulting Service, Sep. 2010.
[58] *Cisco Global Cloud Index: Forecast and Methodology, 2013-2018, 2014-2019*. Cisco White Paper, Cisco Inc., Nov. 2014, Apr. 2016.
[59] S. K. Prasad, A. Gupta, A. L. Rosenberg, A. Sussman, and C. C. Weems. *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses*. Morgen Kaufmann, pp. 125, 2015.
[60] D. Walker and O. Rann. *The Use of Java in High Performance Computing: A Data Mining Example*. Lecture Notes in Computer Science 1593, pp. 861-872, Springer Verlag, 1999.
[61] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis. *Virtualization for High-Performance Computing*. ACM SIGOPS Operating Systems Review 40.2 (2006), pp. 8-11. 2006.
[62] *Big Data Meets High Performance Computing*. White Paper, Intel Enterprise Edition for Lustre Software High Performance Data Division, Jul. 2014.
[63] *The Value of High-Performance Computing for Simulation*. White Paper, ANSYS, 2015.
[64] J. Madeira. *Brute-Force Algorithms*. Lecture slides, University of Aveiro, Sep. 2015.
[65] A. P. Black. *Exhaustive Search Algorithms*. Lecture slides, Portland State University, Oct. 2015.
[66] P. Nyberg. *The Critical Role of Supercomputers in Weather Forecasting*. http://www.cray.com/blog/the-critical-role-of-supercomputers-in-weather-forecasting. Jul. 24, 2013.
[67] M. Chabowski. *It's Raining HPC - How Supercomputing is Shaping Weather Forecasts*. http://insidehpc.com/2014/07/raining-hpc-next-gen-supercomputing-will-shape-weather-forecasts. Jul. 31, 2014.
[68] M. A. Stadtherr. *Large-Scale Process Simulation and Optimization in a High Performance Computing Environment*. AspenWorld, 1997.
[69] S. J. Ezell and R. D. Atkinson. *The Vital Importance of High-Performance Computing to*

U.S. Competitiveness. Information Technology & Innovation Foundation, Apr. 2016.

[70] *Pushing the Boundaries of High-Performance Computing*. U.S. Department of Energy, Sep. 2014.

[71] *A Tutorial on High Performance Computing applied to Cryptanalysis*. Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Berlin, Heidelberg, 2012.

[72] F. Borges. *HPC for Security, Privacy, Cryptography, and Trust*. Laboratório Nacional de Computação Científica (LNCC), Coordenação de Sistemas e Redes (CSR), Sep. 2015.

[73] B. Goudey, M. Abedini, J. L. Hopper, M. Inouye, E. Makalic, D. F. Schmidt, J. Wagner, Z. Zhou, J. Zobel, and M. Reumann. *High performance computing enabling exhaustive analysis of higher order single nucleotide polymorphism interaction in Genome Wide Association Studies*. Health Information Science and Systems 2015, S3, Oct. 2015.

[74] S. Anthony. *What Can You Do with a Supercomputer*. http://www.extremetech.com/extreme/122159-what-can-you-do-with-a-supercomputer. Mar. 15, 2012.

[75] *FutureGrid - an XSEDE resource provider*. http://archive.futuregrid.org.

[76] C. Metz. *Amazon Builds World's Fastest Nonexistent Supercomputer*. http://www.wired.com/2011/12/nonexistent-supercomputer/all. Dec. 23, 2011.

[77] R. Brueckner. *Virtual Supercomputer Service Enters Beta*. http://insidehpc.com/2014/12/virtual-supercomputer-hpc-service-enters-beta. Dec. 2014.

[78] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. *Palacios and Kitten: New High Performance Operating Systems For Scalable Virtualized and Native Supercomputing*. Parallel & Distributed Processing, 2010 IEEE International Symposium, Apr. 2010.

[79] J. R. Lange and P. A. Dinda. *SymCall: Symbiotic Virtualization Through VMM-to-Guest Upcalls*. Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'11), Mar. 2011.

[80] A. Kudryavtsev, V. Koshelev, B. Pavlovic, and A. Avetisyan. *Virtualizing HPC Applications using Modern Hypervisors*. Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit. ACM, 2012.

[81] Y. Zhang, R. Oertel, and W. Rehm. *Performance Loss on Virtual Machines*. TUCSIS 2012, Studentensymposium Informatik Chemnitz 2012 Tagungsband zum 1. Studentensymposium Chemnitz vom 4. Jul. 2012.

[82] Y. Zhang, R. Oertel, and W. Rehm. *Performance Impact of Futex on Virtual Machines*. EMS - IEEE European Modelling Symposium 2013, Manchester, UK, Nov. 2013.

[83] D. Bailey, E. Barszcz, and J. Barton et al. *The NAS Parallel Benchmarks*. RNR Technical Report RNR-94-007, Mar. 1994.

[84] N. Regola and J. C. Ducom. *Recommendations for Virtualization Technologies in High Performance Computing*. Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science, CLOUDCOM'10, pp. 409-416, 2010.

[85] A. O. Kudryavtsev, V. K. Koshelev, and A. I. Avetisyan. *Prospects for Virtualization of High Performance x64 Systems*. Programming and Computer Software, 39(6), pp. 285-294, 2013.

[86] K. Bala, M. F. Kaashoek, and W. E. Weihl. *Software Prefetching and Caching for Translation Lookaside Buffers*. Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation. USENIX Association, 1994.

[87] M. Wu and W. Zwaenepoel. *Improving TLB Miss Handling with Page Table Pointer Caches*. SOSP '95 Proceedings of the 15th ACM Symposium on Operating Systems Principles, Technical report, Rice University, Dec. 1997.

[88] A. Bhattacharjee and M. Martonosi. *Characterizing the TLB Behavior of Emerging*

Parallel Workloads on Chip Multiprocessors. Parallel Architectures and Compilation Techniques, PACT'09. 18th International Conference. IEEE, 2009.

[89] A. Bhattacharjee and M. Martonosi. *Inter-core Cooperative TLB for Chip Multiprocessors*. Proceedings of the 15th edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, pp. 359-370. 2010

[90] A. Bhattacharjee, D. Lustig, and M. Martonosi. *Shared Last-level TLBs for Chip Multiprocessors*. 2011 IEEE 17th International Symposium on High Performance Computer Architecture, 2011.

[91] S. Srikantaiah and M. Kandemir. *Synergistic TLBs for High Performance Address Translation in Chip Multiprocessor*. Microarchitecture (MICRO'10), 43rd Annual IEEE/ACM International Symposium, 2010.

[92] G. B. Kandiraju and A. Sivasubramaniam. *Going the Distance for TLB Prefetching: an Application-driven Study*. ISCA'02 Proceedings of the 29th Annual International Symposium on Computer Architecture, pp. 195-206, May. 2002.

[93] B. L. Jacob and T. N. Mudge. *A Look at Several Memory Management Units, TLB-refill Mechanisms, and Page Table Organizations*. Technical Report, 2009.

[94] M. Talluri, M. D. Hill, and Y. A. Khalidi. *A New Page Table for 64-bit Address Spaces*. SOSP '95 Proceedings of the 15th ACM Symposium on Operating Systems Principles, pp. 184-200 Vol. 29. No. 5. ACM, Dec. 1995.

[95] J. Liedtke. *Address Space Sparsity and Fine Granularity*. ACM SIGOPS Operating Systems Review 29.1 (1995): pp. 87-90, 1995.

[96] R. Bhargave, B. Serebin, F. Spadini, and S. Manne. *Accelerating Two-Dimensional Page Walks for Virtualized Systems*. Advanced Micro Devices. Mar. 2008.

[97] T. W. Barr, A. L. Cox, and S. Rixner. *Translation Caching: Skip, Don't Walk (the Page Table)*. Houston, TX. Jun. 2010.

[98] J. Ahn, S. Jin, and J. Huh. *Revisiting Hardware-Assisted Page Walks for Virtualized Systems*. 39th International Symposium on Computer Architecture (ISCA'12), 2012.

[99] A. Arcangeli and A. Kivity. *Using Linux as Hypervisor with KVM*. Qumranet Inc., CERN, Geneve, Sep. 2008.

[100] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li. *Selective Hardware/Software Memory Virtualization*. ACM VEE'11. Mar. 2011.

[101] C. S. Bae, J. R. Lange, and P. A. Dinda. *Enhancing Virtualized Application Performance Through Dynamic Adaptive Paging Mode Selection*. ACM ICAC'11. Jun. 2011.

[102] Y. Zhang, R. Oertel, and W. Rehm. *Paging Method Switching for QEMU-KVM Guest Machine*, BigDataScience'14 Proceedings of the 2014 International Conference on Big Data Science and Computing Article No. 22, Aug. 2014.

[103] J. Gandhi, M. D. Hill, and M. M. Swift. *Agile Paging: Exceeding the Best of Nested and Shadow Paging*, ISCA'16 Proceedings of the 43rd International Symposium on Computer Architecture, pp. 707-718, Jun. 2016.

[104] G. Hoang, C. Bae, J. Lange, L. Zhang, P. Dinda, and R. Joseph. *A Case for Alternative Nested Paging Models for Virtualized Systems*. Computer Architecture Letters, 9, pp. 17-20, University of Michigan, Jun. 2010.

[105] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift. *Efficient Memory Virtualization*. University of Wisconsin-Madison and AMD Research, Oct. 2014.

[106] G. Shainer, P. Lui, T. Liu, T. Wilde, and J. Layton. *The Impact of Inter-node Latency versus Intra-node Latency on HPC Applications*. The 23rd IASTED International Conference on PDCS 2011, Dec. 2011.

[107] H. Subramoni, M. Koop, and D. K. Panda. *Designing Next Generation Clusters: Evaluation of InfiniBand DDR/QDR on Intel Computing Platforms*. High Performance Interconnects (HOTI'09), Sep. 2009.

[108] G. Shainer. *Offloading vs. Onloading: The Case of CPU Utilization*.

https://www.hpcwire.com/2016/06/18/offloading-vs-onloading-case-cpu-utilization. Jun. 18, 2016.

[109] J. Tao, W. Karl, and C. Trinitis. *Implementing an OpenMP Execution Environment on InfiniBand Clusters*. OpenMP Shared Memory Parallel Programming'08: pp. 65-77, 2008.

[110] M. L. Li, R. Sasanka, S. V. Adve, Y. K. Chen, and E. Debes. *The ALPBench Benchmark Suite for Multimedia Applications*. UIUC CS Technical Report UIUCDCS-R-2005-2603, Jul. 2005.

[111] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C. W. Tseng, and D. Yeung. *BioBench: A Benchmark Suite of Bioinformatics Applications*. IEEE International Symposium on Performance Analysis of Systems and Software, 2005.

[112] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. *MediaBench II Video: Expediting the Next Generation of Video Systems Research*. Microprocessors and Microsystems 33.4 (2009): pp. 301-318, 2009.

[113] *PARSEC Overview*. http://parsec.cs.princeton.edu/overview.htm#Motivation. 2009.

[114] J. Han, J. Ahn, C. Kim, Y. Kwon, Y. Choi, and J. Huh. *The Effect of Multi-core on HPC Applications in Virtualized Systems*. ARCS'2012, LNCS 7179, pp. 123-134, Springer-Verlag Berlin Heidelberg, 2012.

[115] J. Held, J. Bautsta, and S. Koehl. *From a Few Cores to Many - A Tera-scale Computing Research Overview*. White Paper, Research at Intel. Apr. 13, 2012.

[116] O. Wechser. *Developing the HPC Compute Cores of Tomorrow*. Intel Fellow, Visual and Parallel Group. Feb. 10, 2014.

[117] P. Dubey. *Recognition, Mining and Synthesis Moves Computers to the Era of Tera*. Technology at Intel Magazine, Feb. 2005.

[118] C. Bienia, S. Kumar, J. P. Singh, and K. Li. *The PARSEC Benchmark Suite: Characterization and Architectural Implications*. PACT'08, pp. 72-81. 2008.

[119] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. *The SPLASH-2 Programs: Characterization and Methodological Considerations*. Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 24-36, Jun. 1995.

[120] C. Bienia, S. Kumar, and K. Li. *PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors*. IEEE International Symposium on Workload Characterization (IISWC 2008), Sep. 2008.

[121] T. Reeves. *What's the Difference between AMD64 and Intel EM64T*. https://jetteroheller.wordpress.com/2007/03/09/whats-the-difference-between-amd64-and-intel-em64t. Mar. 2007.

[122] *Intel Core i7-2960XM vs i7-6700K*. http://www.cpu-world.com/Compare. CPU World, Feb. 17, 2017.

[123] *Intel Xeon E5-1620 v2 specifications*. http://www.cpu-world.com/CPUs/Xeon. CPU World, Feb. 16, 2017.

[124] *Intel Xeon E5-1620 vs E5-2603 v2*. http://www.cpu-world.com/Compare. CPU World, Nov. 23, 2016.

[125] *AMD FX-8150 Specifications*. http://www.cpu-world.com/CPUs/Bulldozer. CPU World, Feb. 12, 2017.

[126] *AMD FX-8150 - Bulldozer im ausführlichen Test*. http://ht4u.net/reviews/2011/amd_bulldozer_fx_prozessoren. Oct. 12, 2011.

[127] A. Graf. *KVM on PowerPC*. Presentation slides, Aug. 2010.

[128] W. Mauerer. *Professional Linux Kernel Architecture*. pp. 155, 189, Wiley Publishing Inc., 2008.

[129] B. R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. B. R. Preiss, P. Eng, 1999.

[130] J. Corbet. *Four-level Page Tables Merged*. https://lwn.net/Articles/117749. Jan. 2005.

[131] K. A. Shutemov. *x86: 5-level paging enabling for v4.12*. https://lwn.net/Articles/716916.

Mar. 2017.
[132] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A quantitative Approach (Fifth Edition)*. pp. 11-13, 497, 108, Morgan Kaufmann, 2013.
[133] X. Song, J. Shi, H. Chen, and B. Zang. *Schedule Processes, not VCPUs*. APSys'13, ACM, Jul. 2013.
[134] G. C. Buttazzo. *Hypervisor*. Lecture slides of Component-Based Software Design, University of Pisa and Scuola Superiore Sant'Anna of Pisa, May. 2015.
[135] D. Faggioli. *Scheduling in The Age of Virtualization*. FOSDEM (Free and Open Source Software Developers' European Meeting) 2016, Bruxelles, Jan. 2016.
[136] T. Friebel and S. Biemueller. *How to Deal with Lock Holder Preemption*. GI OS Workshop, Oct. 2010.
[137] S. Kashyap, C. Min, and T. Kim. *Opportunistic Spinlocks: Achieving Virtual Machine Scalability in the Clouds*. APSys'15, ACM, Jul. 2015.
[138] J. W. Ousterhout. *Scheduling Techniques for Concurrent Systems*. Proc. 3rd International Conference on Distributed Computing systems. Oct. 1982.
[139] W. Lee, M.Frank, V. Lee, K. Mackenzie, and L. Rudolph. *Implications of I/O for Gang Scheduled Workloads*. Job Scheduling Strategies for Parallel Processing, pp. 215-237, 1997.
[140] VMware. *The CPU Scheduler in VMware vSphere 5.1 Performance Study*. VMware Technical Whitepaper, Feb. 2013.
[141] N. A. Dadhania. http://lwn.net/Articles/472797 Gang scheduling in CFS, Dec. 19, 2011.
[142] L. R. S. Chivukula and T. P. Bhattacharjee. *Evaluation and Analysis of Gang Scheduling in Linux Kernel*. Dec. 2012.
[143] O. Sukwong and H. S. Kim, *Is Co-scheduling Too Expensive for SMP VMs?* EuroSys'11, April 10 - 13, 2011, Salzburg, Austria, Dec. 2012.
[144] T. Miao and H. Chen. *FlexCore: Dynamic Virtual Machine Scheduling Using VCPU Ballooning*. Tsinghua Science and technology, pp. 7-16, Volume 20, Number 1, Feb. 2015.
[145] M. T. Jones. *Virtio: An I/O virtualization framework for Linux*. http://www.ibm.com/developerworks/library/l-virtio. IBM developerWorks, Jan. 2010.
[146] R. Rusell. *virtio: Towards a De-Facto Standard For Virtual I/O Devices*. May. 2008.
[147] N. Har'El, A. Gordon, and A. Landau. *Efficient and Scalable Paravirtual I/O System*. 2013 USENIX Annual Technical Confrence (USENIX ATC'13), Jun. 2013.
[148] L. Rizzo. *netmap: A Novel Framework for Fast Packet I/O*. 2012 USENIX Annual Technical Confrence (USENIX ATC'12), Jun. 2012.
[149] L. Rizzo, M. Carbone, and G. Catalli. *Transparent acceleration of software packet forwarding using netmap*. INFOCOM, 2012 Proceedings IEEE, Mar. 2012.
[150] L. Rizzo. *Revisiting Network I/O APIs: The netmap Framework*. ACMQUEUE Networks, Volume 10, Issue 1, Jan. 2012.
[151] L. Rizzo and G. Lettieri. *VALE, a switched ethernet for virtual machines*. CoNEXT'12, 2012, Nice, France. Jun. 2012.
[152] L. Rizzo and G. Lettieri. *Virtual Device Passthrough for High Speed VM Networking*. ANCS'15 Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 99-110, Mar. 2015.
[153] L. Rizzo, G. Lettieri, and V. Maffione. *Speeding up packet I/O in virtual machines*. ANCS'13 Proceedings of the ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 47-58, Oct. 2015.
[154] A. Gordon, N. Amit, and N. Har'El. *ELI: Bare-Metal Performance for I/O Virtualization*. ASPLOS'12, UK. Mar. 2012.
[155] F. Diakhaté, M. Perache, R. Namyst, and H. Jourdren. *Efficient Shared Memory Message Passing for Inter-VM Communications*. Euro-Par 2008 Workshops - Parallel Processing,

Volume 5415 of the series Lecture Notes in Computer Science pp. 53-62. 2008.
[156] B. P. Haley. *ABINIT: First Time User Guide*. Presentation slides, Network for Computational Nanotechnology (NCN), Purdue University, Jun. 2014.
[157] *CM1 Homepage*. http://www2.mmm.ucar.edu/people/bryan/cm. Aug. 2017.
[158] G. Bryan. *Parallelization in CM1*. Presentation slides, NCAR, Dec. 2009.
[159] P. Giannozzi, S. Baroni, N. Bonini et al. QUANTUM ESPRESSO*: a modular and open-source software project for quantum simulations of materials*. Journal of Physicals: Condensed Matter 21 (2009) 395502, 2009.
[160] *User's Guide for* QUANTUM ESPRESSO *(v.6.1)*. Mar. 2017.
[161] E. Lindahl, B. Hess, and D. van der Spoel. *GROMACS 3.0: a package for molecular simulation and trajectory analysis*. Journal of Molecular Model, pp. 306-317, 2001.
[162] *HPCC Performance Benchmark and Profiling*. Presentation slides, HPC Advisory Council, Dec. 2015.
[163] J. Dongarra. *The HPC Challenge Benchmark*. Presentation slides, May. 2004.
[164] *HPL Benchmark*. http://icl.eecs.utk.edu/hpl. Sep. 14, 2017.
[165] B. Wilson. *Short Tutorial/Starter For LAMMPS - LJ17 Simulation*. Apr. 2014.
[166] *Notes on SU(3) and the Quark Model*. Dec. 2009.
[167] A. Bazavov, C. Bernard et al. *The MILC Code (version 7.7.11)*. Mar. 2014.
[168] *MILC Performance Benchmark and Profiling*. Presentation slides, HPC Advisory Council, Aug. 2012.
[169] A. E. Darling, L. Carey, and W. C. Feng. *The Design, Implementation, and Evaluation of mpiBLAST*. Proceedings of ClusterWorld 2003, 2003.
[170] C. S. Oehmen, and J. Nieplocha. *ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis*. IEEE Transactions on Parallel and Distributed Systems, 17(8): pp. 740-749, 2006.
[171] A. Kalyanaraman. *Introduction to BLAST*. Presentation slides, May. 2010.
[172] *mpiBLAST: Open-Source Parallel BLAST*. http://www.mpiblast.org. Nov. 2012.
[173] *Phylogeny Definition*. https://biologydictionary.net/phylogeny. Sep. 2017.
[174] F. Ronquist and J. P. Huelsenbeck. *MrBayes 3: Bayesian phylogenetic inference under mixed models*. Bioinformatics Application Note Vol. 19 no. 2003(12), pp. 1572-1574, 2003.
[175] A. Sayma. *Computational Fluid Dynamics*. BookBoon 2009 Publisher, 2009.
[176] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. Third Edition, Mit University Press Group Ltd., 2014.
[177] *NEK5000: A fast and scalable open-source spectral element solver for CFD*. https://nek5000.mcs.anl.gov. Sep. 2017.
[178] K. Capelle. *A bird's-eye view of density-functional theory*. Brazilian Journal of Physics, Nov. 2006.
[179] X. Andrade, J. Alberdi-Rodriguez, D. A. Strubbe, M. J. T. Oliveira, F. Nogueira, A. Castro, and J. Muguerza. *Time-Dependent Density-Functional Theory in Massively Parallel Computer Architectures: The Octopus Project*. Journal of Physics: Condensed Matter 24.23 (2012): 233202, 2012.
[180] A. Castro, H. Appel1, M. Oliveira, C. A. Rozzi, X. Andrade, F. Lorenzen, M. A. L. Marques, E. K. U. Gross, and A. Rubio. *Octopus: A Tool for the Application of Time-dependent Density Functional Theory*. Physica Status Solidi (b), 243(11), pp. 2465-2488, 2006.
[181] D. A. Strubbe et al. *Introduction to Octopus: a real-space (TD)DFT code*. Presentation slides, TDDFT 2012, Jan. 2012.
[182] *Octopus Performance Benchmark and Profiling*. Presentation slides, HPC Advisory Council, May. 2011.
[183] *Octopus: Main Page*. http://octopus-code.org/wiki. Jun. 2017.
[184] G. J. Martyna, E. J. Bohm, R. Venkataraman, A. Arya, L. V. Kale, and A. Bhatele.

OpenAtom: Ab initio Molecular Dynamics for Petascale Platforms. LLNL-BOOK-608553, Dec. 2012.

[185] N. Jain, E. Bohm, E. Mikida, S. Mandal, M. Kim, P. Jindal, Q. Li , S. Ismail-Beigi, G. J. Martyna, and L. V. Kale. *OpenAtom: Scalable Ab-Initio Molecular Dynamics with Diverse Capabilities.* ISC High Performance Computing pp. 139-158, Jun. 2016.

[186] *OpenAtom Performance Benchmark and Profiling.* Presentation slides, HPC Advisory Council, Feb. 2010.

[187] *An Introduction to Ray Tracing.* Technical report, TR87-038, Dec. 1987.

[188] J. E. Stone. *An Efficient Library for Parallel Ray Tracing and Animation.* Feb. 2001.

[189] J. E. Stone. *Tachyon Parallel / Multiprocessor Ray Tracing System.* http://jedi.ks.uiuc.edu/~johns/raytracer. Sep. 2017.

[190] K. Meyer. *WOMBAT-A tool for mixed model analyses in quantitative genetics by restricted maximum likelihood (REML).* Journal of Zhejiang University Science B 8(11): pp. 815-821, Nov. 2007.

[191] K. Meyer. *WOMBAT - A Program for Mixed Model Analyses by Restricted Maximum Likelihood.* User Notes, Aug. 2012.

[192] K. Meyer. *WOMBAT - Digging Deep for Quantitative Genetic Analyses by Restricted Maximum Likelihood.* 8th World Congress on Genetics Applied to Livestock Production, Aug. 2006.

[193] *ARW - Version 3 Modeling System User's Guide.* Mesoscale & Microscale Meteorology Laboratory, National Center for Atmospheric Research, Jun. 2017.

[194] *WRF Model Users' Page.* http://www2.mmm.ucar.edu/wrf/users. Sep. 2017.

[195] *Intel 64 and IA-32 Architectures Software Developer's Manual.* Vol.3C 28-9, Intel Corporation, Oct. 2016.

[196] Adavanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Volume 2: System Programming.* p80, Adavanced Micro Devices, May. 2015.

[197] A. Färber. *Modern QEMU Devices - A Hands-On Approach.* Presentation Slides, SUSE Linux Product GmbH, Oct. 21, 2013.

[198] A. Liguori. *Features/QDevCleanup.* http://wiki.qemu.org/Features/QDevCleanup. Aug. 31, 2010.

[199] P. Bonzini. *QOM exegesis and apocalypse.* Presentation, Red Hat. KVM Forum, 2014.

[200] A. Liguori. *Features/QOM.* http://wiki.qemu.org/Features/QOM. Sep. 17, 2011.

[201] Archlinux. *QEMU.* https://wiki.archlinux.org/index.php/QEMU. June 14, 2015.

[202] A. Liguori. *RFC - QEMU Object Model.* https://lists.nongnu.org. Jul. 19, 2011.

[203] *Intel 64 and IA-32 Architectures Software Developer's Manual - Documentation Changes.* Vol.3B 18-3, Intel Corporation, Jun. 2016.

[204] E. Dong, Y. Zhao, and X. Li. *KVM tuning and testing, and SMP enhancement.* Presentation Slides, Intel Corporation, Sep. 2007.

[205] M. Tim Jones. *Kernel APIs, Part 3: Timers and lists in the 2.6 kernel - Efficient processing with work-deferral APIs.* IBM developerWorks, Mar. 2010.

[206] *Linux kernel Documentation about MMU in KVM.* https://www.kernel.org/doc/Documentation/virtual/kvm/mmu.txt.

[207] M. K. Johnson. *Memory Allocation.* Linux Journal, Issue 16, http://www.linuxjournal.com/article/1133. Aug. 1995.

[208] *Wind River Simics.* Product note, Virtutech, Oct. 2015.

[209] J. Engblom. *Virtutech Simics.* Presentation slides, Virtutech, Sep. 2005.

[210] P. Magnusson and B. Werner. *Efficient Memory Simulation in SimICS.* Swedish Institute and Bengt Werner, Dec. 2003.

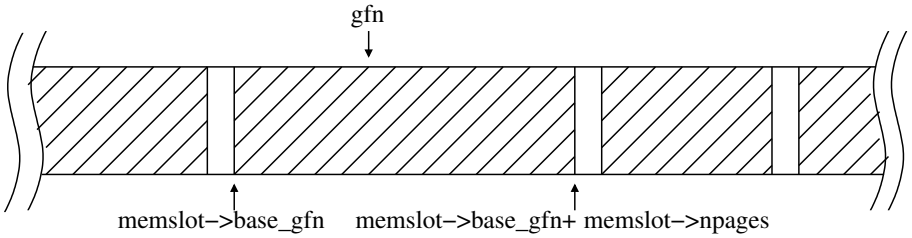
[211] G. J. Myers. *The Art of Software Testing.* p177, Wiley, New York, 1979.

[212] W. Hetzel and W. C. Hetzel. *The Complete Guide to Software Testing.* 2nd ed, pp. 280, QED Information Sciences, Inc. Wellesley, MA, USA, 1988.

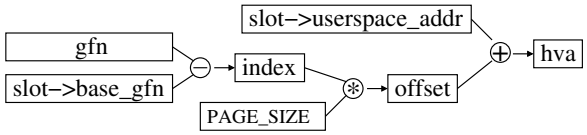
Appendix

Appendix A Some Data Structures in KVM

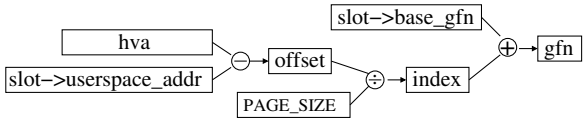
A.1 Searching of `kvm_mem_slot`



A.2 Translation between GFN and HVA

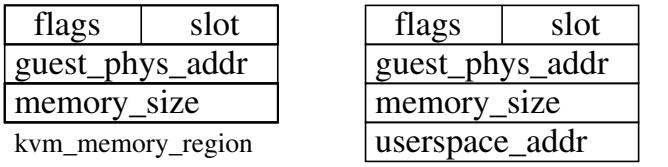


(a) GFN to HVA



(b) HVA to GFN

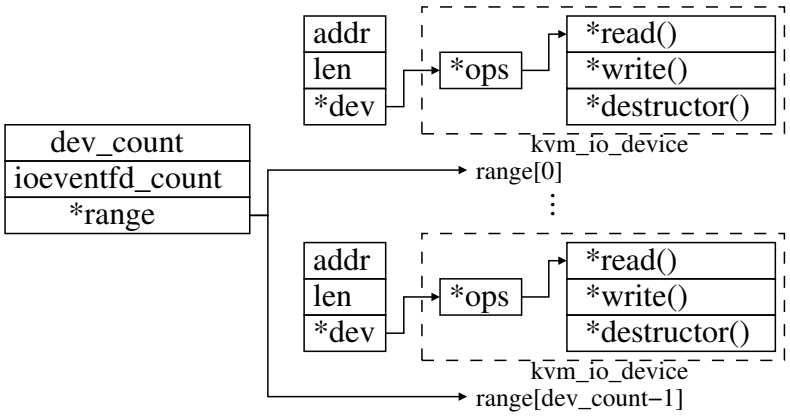
A.3 `kvm_memory_region` and `kvm_userspace_memory_region`



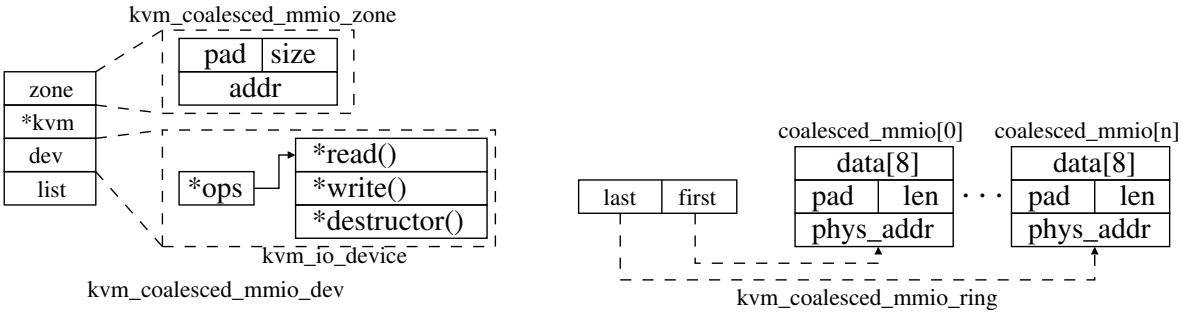
kvm_memory_region

kvm_userspace_memory_region

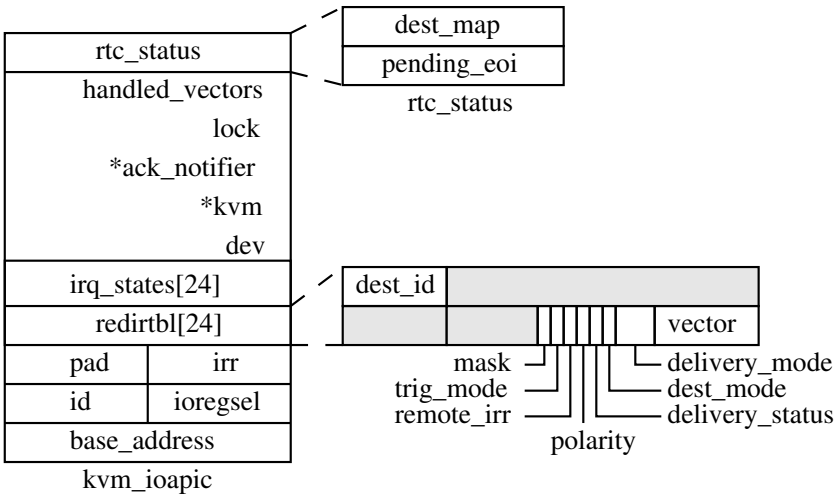
A.4 kvm_io_bus



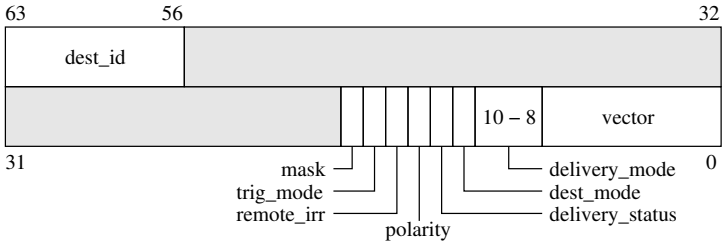
A.5 kvm_coalesced_mmio_dev and kvm_coalesced_mmio_ring



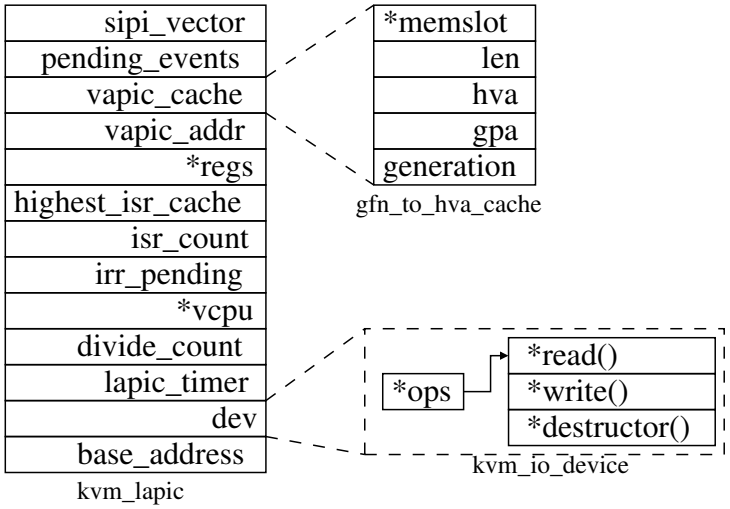
A.6 kvm_ioapic



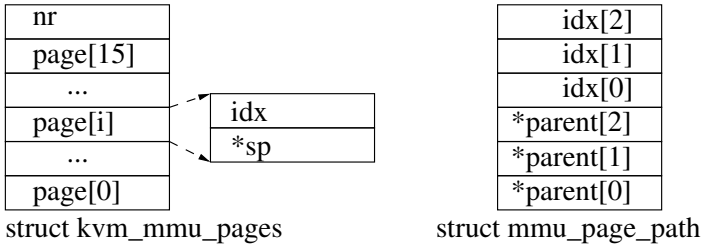
A.7 interrupt message



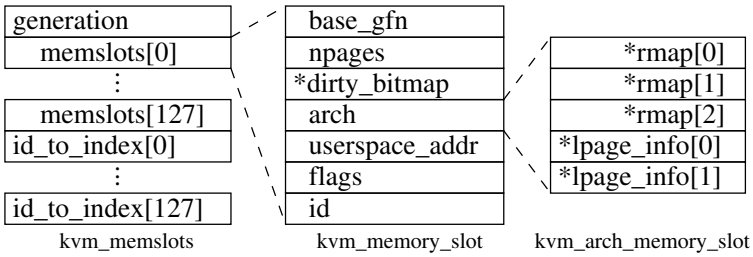
A.8 kvm_lapic



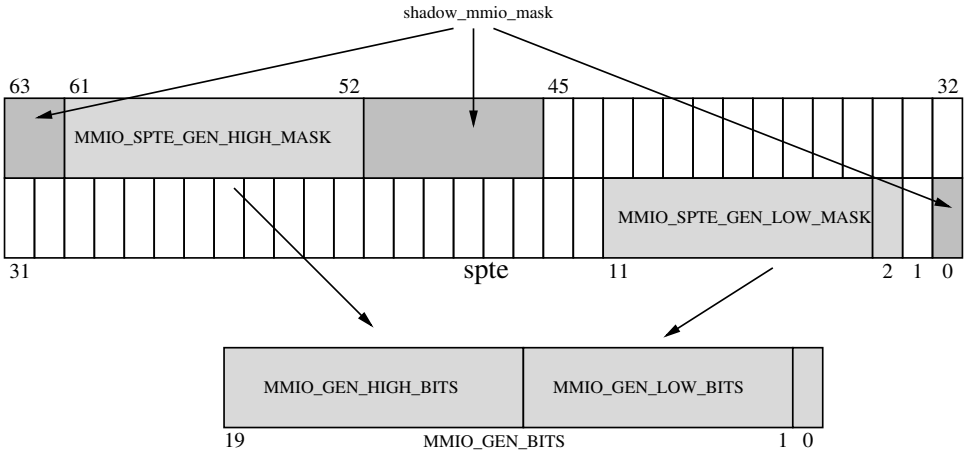
A.9 iterator



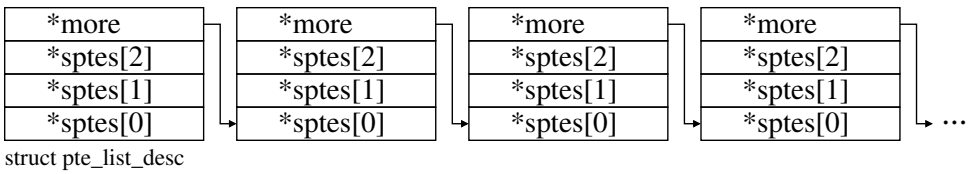
A.10 kvm_mmuslot



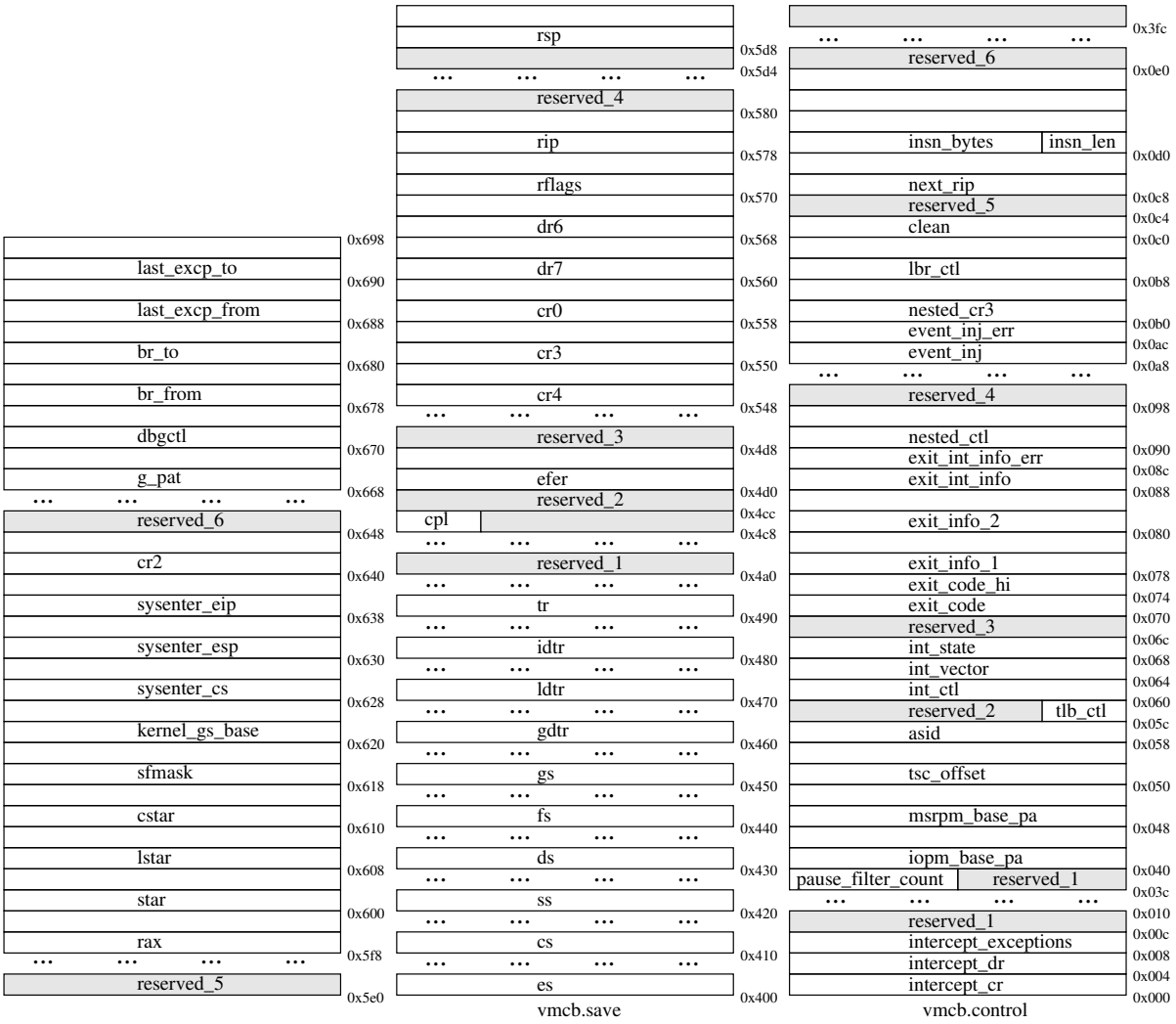
A.11 GEN_MMIO



A.12 pte_list_desc



A.13 VMCB layout in AMD NPT



Appendix B Part of the DPMS Code

B.1 Paging Method Switching Operation in KVM

```
static int ept_to_spt(struct kvm_vcpu *vcpu)
{
        struct kvm *kvm = vcpu->kvm;
        struct kvm_vcpu *v;
        unsigned int i;
//      DEFINE_WAIT(wait);

        if (vcpu->vcpu_id)
                return 0;
        if (!vcpu->arch.mmu.direct_map)
                return 0;
//      kvm_guest_suspend(kvm, wait); /* suspend this guest */
        enable_ept = 0;
        kvm_disable_tdp();
        kvm_for_each_vcpu(i, v, kvm) {
                kvm->arch.pmc.root_hpa[v->vcpu_id] = v->arch.mmu.root_hpa;
                v->arch.mmu.root_hpa = INVALID_PAGE;
                vmcs_update(v);
                kvm_mmu_setup(v);
                kvm_mmu_load(vcpu);
        }
//      kvm_guest_restore(kvm, wait); /* restore this guest */
        return 0;
}

static int spt_to_ept(struct kvm_vcpu *vcpu)
{
        struct kvm *kvm = vcpu->kvm;
        struct kvm_vcpu *v;
        unsigned int i;
//      DEFINE_WAIT(wait);

        if (vcpu->vcpu_id)
                return 0;
        if (vcpu->arch.mmu.direct_map)
                return 0;
//      kvm_guest_suspend(kvm, wait); /* suspend this guest */
        enable_ept = 1;
        kvm_enable_tdp();
        kvm_for_each_vcpu(i, v, kvm) {
                kvm_mmu_unload(v);
                vmcs_update(v);
                kvm_mmu_setup(v);
                if (kvm->arch.pmc.first_time) {
                        kvm->arch.pmc.first_time = 0;
                        kvm_mmu_load(vcpu);
                }
                else {
                        v->arch.mmu.root_hpa = kvm->arch.pmc.root_hpa[v->vcpu_id];
                        vmx_set_cr3(vcpu, v->arch.mmu.root_hpa);
                }
        }
//      kvm_guest_restore(kvm, wait); /* restore this guest */
        return 0;
}
```

B.2 VMCS Updating Operation in KVM

```
static void vmcs_update(struct kvm_vcpu *vcpu)
{
        struct vcpu_vmx *vmx = to_vmx(vcpu);
        u32 exec_control, sec_exec_ctl;
        u32 eb;
        u64 mask;
        int maxphyaddr = boot_cpu_data.x86_phys_bits;
        eb = vmcs_read32(EXCEPTION_BITMAP);

        if (enable_ept)
                eb &= ~(1u << PF_VECTOR);
        else
                eb |= (1u << PF_VECTOR);

        vmcs_write32(EXCEPTION_BITMAP, eb);
        mask = rsvd_bits(maxphyaddr, 51);
        mask |= 0x3ull << 62;
        mask |= 1ull;

#ifdef CONFIG_X86_64
        if (maxphyaddr == 52)
        mask &= ~1ull;
#endif

        kvm_mmu_set_mmio_spte_mask(mask);

        kvm_mmu_set_mask_ptes(PT_USER_MASK, PT_ACCESSED_MASK,
                        PT_DIRTY_MASK, PT64_NX_MASK, 0);
        if (enable_ept) {
                if (!vcpu->kvm->arch.ept_identity_map_addr)
                        vcpu->kvm->arch.ept_identity_map_addr =
                        VMX_EPT_IDENTITY_PAGETABLE_ADDR;
                init_rmode_identity_map(vcpu->kvm);

                kvm_mmu_set_mask_ptes(0ull,
                        (enable_ept_ad_bits) ? VMX_EPT_ACCESS_BIT : 0ull,
                        (enable_ept_ad_bits) ? VMX_EPT_DIRTY_BIT : 0ull,
                        0ull, VMX_EPT_EXECUTABLE_MASK);
                ept_set_mmio_spte_mask();
        }

        exec_control = vmx_exec_control(vmx);
        if (enable_ept)
                exec_control &= ~(CPU_BASED_CR3_STORE_EXITING |
                                  CPU_BASED_CR3_LOAD_EXITING  |
                                  CPU_BASED_INVLPG_EXITING);
        vmcs_write32(CPU_BASED_VM_EXEC_CONTROL, exec_control);

        if (!enable_ept || !enable_ept_ad_bits || !cpu_has_vmx_pml())
                enable_pml = 0;
        if (!enable_pml) {
                kvm_x86_ops->slot_enable_log_dirty = NULL;
                kvm_x86_ops->slot_disable_log_dirty = NULL;
                kvm_x86_ops->flush_log_dirty = NULL;
                kvm_x86_ops->enable_log_dirty_pt_masked = NULL;
        }
        else {
                vmx_create_pml_buffer(vmx);
                kvm_x86_ops->slot_enable_log_dirty = vmx_slot_enable_log_dirty;
                kvm_x86_ops->slot_disable_log_dirty = vmx_slot_disable_log_dirty;
                kvm_x86_ops->flush_log_dirty = vmx_flush_log_dirty;
                kvm_x86_ops->enable_log_dirty_pt_masked = vmx_enable_log_dirty_pt_masked;
        }

        setup_msrs(vmx); // newly added
        sec_exec_ctl = vmx_secondary_exec_control(vmx);
        vmcs_write32(SECONDARY_VM_EXEC_CONTROL, sec_exec_ctl);
    }
}
```

Publication

1. Yu Zhang, Peter Tröger, Matthias Werner, *Dynamic Paging Method Switching - an Implementation for KVM*, Virtualization for High Performance Cloud Computing Workshop, ISC VHPC 2017, Frankfurt (am Main), Germany, June, 2017.

2. Yu Zhang, *A Simplified TDP with Large Page Table*, Virtualization for High Performance Cloud Computing Workshop, Euro-Par VHPC 2015, Vienna, Austria, August, 2015.

3. Yu Zhang, René Oertel, Wolfgang Rehm, *Paging Method Switching for QEMU-KVM Guest Machine*, ACM BigDataScience 2014, Beijing, China, August 2014.

4. Yu Zhang, René Oertel, Wolfgang Rehm, *Performance Impact of Futex on Virtual Machines*, EMS 2013, IEEE European Modelling Symposium 2013, Manchester, UK, November 2013.

5. Yu Zhang, René Oertel, Wolfgang Rehm, *Performance Loss on Virtual Machines*, TUCSIS 2012, Studentensymposium Informatik Chemnitz 2012 Tagungsband zum 1. Studentensymposium Chemnitz vom 4. Juli 2012.