HZ Books
华章教育

经 典 原 版 书 库

云计算与分布式系统

从并行处理到物联网

（美）Kai Hwang　Geoffrey C. Fox　Jack J. Dongarra 著

（英文版）

Distributed and Cloud Computing
From Parallel Processing to the Internet of Things

Distributed and Cloud Computing
From Parallel Processing to the Internet of Things

Kai Hwang · Geoffrey C. Fox · Jack J. Dongarra

机械工业出版社
China Machine Press

MK
MORGAN KAUFMANN

经典原版书库

云计算与分布式系统：从并行处理到物联网

（英文版）

Distributed and Cloud Computing: From Parallel Processing to the Internet of Things

Kai Hwang

（美） Geoffrey C. Fox 著

Jack J. Dongarra

机械工业出版社
China Machine Press

本书版权登记号：图字：01-2012-2643

图书在版编目（CIP）数据

云计算与分布式系统：从并行处理到物联网（英文版）/（美）黄铠，（美）福克斯（Fox, G. C.），（美）唐加拉（Dongarra, J. J.）著．　北京：机械工业出版社，2012.5

（经典原版书库）

书名原文：Distributed and Cloud Computing: From Parallel Processing to the Internet of Things

ISBN 978-7-111-38227-0

I. 云　II. ①黄　②福　③唐　III. ①计算机网络－英文　②分布式操作系统－英文　IV. ① TP393 ② TP316.4

中国版本图书馆 CIP 数据核字（2012）第 079997 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）

责任编辑：迟振春

印刷

2012 年 5 月第 1 版第 1 次印刷

186mm　240mm • 41.75 印张

标准书号：ISBN 978-7-111-38227-0

定价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

华章科技图书出版中心

This book is dedicated to our wives: Jennifer, Judy, and Sue; and to our children: Tony, Andrew, Katherine, Annie, Alexis, James, Heather, Pamela, Oliver, Nick, Ben, and Katie.

KH, GF, and JD

Foreword

Richard Feynman, in his wonderful autobiography *Surely You're Joking, Mr. Feynman*, recounts how at Los Alamos in 1944 he was responsible for supervising the human computers who performed the long and tedious calculations required by the Manhattan Project. Using the mechanical calculators that were then the state of the art, the best human computer could achieve only one addition or multiplication every few seconds. Feynman and his team thus developed methods for decomposing problems into smaller tasks that could be performed simultaneously by different people (they passed cards with intermediate results between people operating adders, multipliers, collators, and sorters); for running multiple computations at once in the same computing complex (they used different color cards); for prioritizing a more important computation (they eliminated cards of other colors); and for detecting and recovering efficiently from errors (relevant cards, and their descendants, were removed, and computations restarted).

Seventy years later, computer architects face similar challenges—and have adopted similar solutions. Individual computing devices are far faster, but physical constraints still limit their speed. Thus, today's computing landscape is characterized by pervasive parallelism. Individual processors incorporate pipelining, parallel instructions, speculative execution, and multithreading. Essentially every computer system, from the humblest desktop to the most powerful supercomputer, incorporates multiple processors. Designers of future exascale supercomputers, to be capable of 10^{18} operations per second, tell us that these computers will need to support 10^{7} concurrent operations.

Parallelism is fundamentally about communication and coordination, and those two activities have also been transformed over the past seventy years by dramatic technological change. Light is no faster, at 8 inches or 20 centimeters per nanosecond in fiber, than in Feynman's time; one can never expect to send a message in less than 50 milliseconds from Los Angeles to Auckland. But the rate at which data can be transmitted has changed dramatically, from a few characters per second in 1910 (early telegraphs) to thousands of characters per second in 1960 (ARPANET) to more than 10 billion characters per second over optical fibers in 2010.

Quasi-ubiquitous high-speed communications not only allows call centers to be relocated to India, it also allows computation to be moved to centralized facilities that achieve massive economies of scale, and permits enormous quantities of data to be collected and organized to support decision making by people worldwide. Thus, government agencies, research laboratories, and companies who need to simulate complex phenomena create and operate enormous supercomputers with hundreds of thousands of processors. Similarly, companies such as Google, Facebook, and Microsoft who need to process large quantities of data operate numerous massive "cloud" data centers that may each occupy tens of thousands of square feet and contain tens or hundreds of thousands of computers. Like Feynman's Los Alamos team, these computing complexes provide computing as a service for many people, and must juggle many computations performed for different purposes.

Massive parallelism, ultra-fast communication, and massive centralization are all fundamental to human decision making today. The computations that are used to forecast tomorrow's weather, index the web, recommend movies, suggest social connections, predict the future state of the stock market, or provide any one of a multitude of other desirable information products are typically distributed over thousands of processors and depend on data collected from sometimes millions of

sources worldwide. Indeed, little of the modern world could function as it does without parallel and distributed computing.

In this pervasively parallel and distributed world, an understanding of distributed computing is surely an essential part of any undergraduate education in computer science. (Indeed, I would argue, an understanding of these topics should be an essential part of *any* undergraduate education. But I leave that argument for another time.) The most complex computer systems today are no longer individual microprocessors, but entire data centers. The most complex computer programs written today are those that manage or run on data-center-scale systems. A student who graduates with a degree in computer science and does not understand how these systems and programs are constructed is profoundly unprepared to engage productively in the modern workforce.

Hwang, Fox, and Dongarra's text is thus especially timely. In its three sections, it covers progressively the hardware and software architectures that underpin modern massively parallel computer systems; the concepts and technologies that enable cloud and distributed computing; and advanced topics in distributed computing, including grid, peer-to-peer, and the Internet of Things. In each area, the text takes a systems approach, describing not only concepts but also representative technologies and realistic large-scale distributed computing deployments. Computing is as much an engineering discipline as a science, and these descriptions of real systems will both prepare students to use those systems and help them understand how other architects have maneuvered the constraints associated with large-scale distributed system design.

The text also addresses some of the more challenging issues facing computer science researchers today. To name just two, computers have emerged as a major consumer of electricity, accounting for several percent of all electricity used in the US. (In Japan, it is ironic that following the 2011 tsunami, the large supercomputers that may help prepare for future natural disasters must often be turned off to conserve power.) And, the fact that 98% of the roughly 10 billion processors sold each year are for embedded devices, and that these embedded devices are increasingly communication-enabled, introduces the opportunity and challenge of an "Internet of Things" that will be vastly larger, more complex, and more capable than today's Internet of People.

I hope that the appearance of this book will stimulate more teaching of distributed computing in universities and colleges—and not just as an optional topic, as is too often the case, but as a core element of the undergraduate curriculum. I hope also that others outside universities will take this opportunity to learn about distributed computing, and more broadly about what computing looks like on the cutting edge: sometimes messy; often complex; but above all tremendously exciting.

Ian Foster
Jackson Hole, Wyoming
August, 2011

Preface

ABOUT THE BOOK

Over the past three decades, parallel processing and distributed computing have emerged as a well-developed field in computer science and information technology. Many universities and colleges are now offering standard courses in this field. However, the instructors and students are still in search of a comprehensive textbook that integrates computing theories and information technologies with the design, programming, and application of distributed systems. This book is designed to meet these demands. The book can be also used as a major reference for professionals working in this field.

The book addresses the latest advances in hardware and software, system architecture, new programming paradigms, and ecosystems emphasizing both speed performance and energy efficiency. These latest developments explain how to create high-performance clusters, scalable networks, automated data centers, and high-throughput cloud/grid systems. We also cover programming, and the use of distributed or cloud systems in innovative Internet applications. The book aims to transform traditional multiprocessors and multi-computer clusters into web-scale grids, clouds, and P2P networks for ubiquitous use in the future Internet, including large-scale social networks and the Internet of things that are emerging rapidly in recent years.

A GLANCE AT THE CONTENTS

We have included many milestone developments in a single volume. We present the contributions not only from our own research groups but also from leading research peers in the U.S., China, and Australia. Collectively, this group of authors and contributors summarize the progress that has been made in recent years, ranging from parallel processing to distributed computing and the future Internet.

Starting with an overview of modern distributed models, the text exposes the design principles, system architectures and innovative applications of parallel, distributed, and cloud computing systems. This book attempts to integrate parallel processing technologies with the network-based distributed systems. The book emphasizes scalable physical systems and virtualized data centers and cloud systems for research, e-commerce, social networking, supercomputing, and more applications, using concrete examples from open-source and commercial vendors.

The nine chapters are divided into three Parts: Part 1 covers system models and enabling technologies, including clustering and virtualization. Part 2 presents data center design, cloud computing platforms, service-oriented architectures, and distributed programming paradigms and software support. In Part 3, we study computational/data grids, peer-to-peer networks, ubiquitous clouds, the Internet of Things, and social networks.

Cloud computing material is addressed in six chapters (1, 3, 4, 5, 6, 9). Cloud systems presented include the public clouds: Google AppEngine, Amazon Web Service, Facebook, SalesForce.com, and many others. These cloud systems play an increasingly important role in upgrading the web services and Internet applications. Computer architects, software engineers, and system designers may want to explore the cloud technology to build the future computers and Internet-based systems.

KEY FEATURES

- Coverage of modern distributed computing technology including computer clusters, virtualization, service-oriented architecture, massively parallel processors, peer-to-peer systems, cloud computing, social networks, and the Internet of Things.
- Major emphases of the book lie in exploiting the ubiquity, agility, efficiency, scalability, availability, and programmability of parallel, distributed, and cloud computing systems.
- Latest developments in Hardware, Networks, and System Architecture:
  - Multi-core CPUs and Many-Core GPUs (Intel, Nvidia, AMD)
  - Virtual Machines and Virtual Clusters (CoD, Violin, Amazon VPC)
  - Top-500 Architectures (Tianbhe-1A, Jaguar, Roadrunner, etc.)
  - Google AppEngine, Amazon AWS, Microsoft Azure, IBM BlueCloud
  - TeraGrid, DataGrid, ChinaGrid, BOINIC, Grid5000 and FutureGrid
  - Chord, Napster, BiTorrent, KaZaA, PPlive, JXTA, and .NET
  - RFID, Sensor Networks, GPS, CPS, and the Internet of Things
  - Facebook, Force.Com, Twitter, SGI Cylone, Nebula, and GoGrid
- Recent advances in paradigms, programming, software and ecosystems:
  - MapReduce, Dryad, Hadoop, MPI, Twister, BigTable, DISC, etc
  - Cloud Service and Trust Models (SaaS, IaaS, PaaS, and PowerTrust)
  - Programming Languages and Protocols (Python, SOAP, UDDI, Pig Latin)
  - Virtualization Software (Xen, KVM, VMware ESX, etc.)
  - Cloud OS and Meshups (Eucalyptus, Nimbus, OpenNebula, vShere/4, etc.)
  - Service-Oriented Architecture (REST, WS, Web 2.0, OGSA, etc.)
  - Distributed Operating Systems (DCE, Amoeba, and MOSIX)
  - Middleware and Software Libraries (LSF, Globus, Hadoop, Aneka)
- Over 100 examples are illustrated with 300 figures, designed to meet the need of students taking a distributed system course. Each chapter includes exercises and further reading.
- Included are case studies from the leading distributed computing vendors: Amazon, Google, Microsoft, IBM, HP, Sun, Silicon Graphics, Rackspace, SalesForce.com, netSuite, Enomaly, and many more.

READERSHIP AND SUGGESTIONS TO INSTRUCTORS/STUDENTS

The readership includes students taking a distributed systems or distributed computing class. Professional system designers and engineers may find this book useful as a reference to the latest distributed system technologies including clusters, grids, clouds, and the Internet of Things. The book gives a balanced coverage of all of these topics, looking into the future of Internet and IT evolutions.

The nine chapters are logically sequenced for use in an one-semester (45-hour lectures) course for seniors and graduate-level students. For use in a tri-semester system, Chapters 1, 2, 3, 4, 6, and 9 are suitable for a 10-week course (30-hour lectures). In addition to solving homework problems, the students are advised to conduct some parallel and distributed programming experiments on available clusters, grids, P2P, and cloud platforms. Sample projects and a solutions manual will be made available to proven instructors from Morgan Kaufmann, Publishers.

INVITED CONTRIBUTIONS

The book was jointly planned, authored, edited, and proofread by all three lead authors in four years (2007–2011). Over this period, we have invited and received partial contributions and technical assistance from the following scientists, researchers, instructors, and doctorial students from 10 top Universities in the U.S., China, and Australia.

Listed below are the invited contributors to this book. The authorship, contributed sections, and editorship of individual chapters are explicitly identified at the end of each chapter, separately. We want to thank their dedicated work and valuable contributions throughout the courses of repeated writing and revision process. The comments by the anonymous reviewers are also useful to improve the final contents.

Albert Zomaya, Nikzad Rivandi, Young-Choon Lee, Ali Boloori, Reza Moraveji, Javid Taheri, and **Chen Wang**, Sydney University, Australia

Rajkumar Buyya, University of Melbourne, Australia

Yongwei Wu, Weimin Zheng, and **Kang Chen**, Tsinghua University, China

Zhenyu Li, Ninghui Sun, Zhiwei Xu, and **Gaogang Xie**, Chinese Academy of Sciences

Zhibin Yu, Xiaofei Liao and **Hai Jin**, Huazhong University of Science and Technology

Judy Qiu, Shrideep Pallickara, Marlon Pierce, Suresh Marru, Gregor Laszewski, Javier Diaz, Archit Kulshrestha, and **Andrew J. Younge**, Indiana University

Michael McLennan, George Adams III, and **Gerhard Klimeck**, Purdue University

Zhongyuan Qin, Kaikun Dong, Vikram Dixit, Xiaosong Lou, Sameer Kulkarni, Ken Wu, Zhou Zhao, and **Lizhong Chen**, University of Southern California

Renato Figueiredo, University of Florida

Michael Wilde, University of Chicago.

PERMISSIONS AND ACKNOWLEDGEMENTS

The permission to use copyrighted illustrations is openly acknowledged in the respective figure captions. In particular, we would like to thank Bill Dally, John Hopecroft, Mendel Roseblum, Dennis Gannon, Jon Kleinberg, Rajkumar Buyya, Albert Zomaya, Randel Bryant, Kang Chen, and Judy Qiu for their generosity in allowing us to use their presentation slides, original figures, and illustrated examples in this book.

We want to thank Ian Foster who wrote the visionary Foreword to introduce this book to our readers. The sponsorship by Todd Green and the editorial work of Robyn Day from Morgan Kaufmann Publishers and the production work led by Dennis Troutman of diacriTech are greatly appreciated. Without the collective effort of all of the above individuals, this book might be still in preparation. We hope that our readers will enjoy reading this timely-produced book and give us feedback for amending omissions and future improvements.

Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra

About the Authors

Kai Hwang is a Professor of Electrical Engineering and Computer Science, University of Southern California. Presently, he also serves as an Intellectual Venture-endowed Visiting Chair Professor at Tsinghua University in Beijing. He earned his Ph.D. in EECS from University of California at Berkeley in 1972. He has taught at Purdue University for 12 years prior to joining USC in 1985. Hwang has served as the founding Editor-in-Chief of the *Journal of Parallel and Distributed Computing* for 26 years. He is a world-renowned scholar and educator in computer science and engineering. He has published 8 books and 220 original papers in computer architecture, digital arithmetic, parallel processing, distributed systems, Internet security, and cloud computing.

Four of his published books: *Computer Arithmetic* (Wiley, 1978), *Computer Architecture and Parallel Processing* (McGraw-Hill, 1983), *Advanced Computer Architecture* (McGraw-Hill, 1993), and *Scalable Parallel Computing* (McGraw-Hill, 1998) have been translated into Spanish, Japanese, Chinese, and Korean from the English editions. By 2011, his published papers and books were cited over 9,000 times. Dr. Hwang has delivered three dozen keynote addresses in major IEEE/ACM conferences. In 1986, the IEEE Computer Society elevated him to an IEEE Fellow. He received the 2004 Outstanding Achievement Award from China Computer Federation. In May 2011, he received the IPDPS Founders' Award for his pioneering contributions in the field of parallel processing. For details, visit the web page: http://GridSec.usc.edu/hwang.html.

Geoffrey C. Fox is a Distinguished Professor of Informatics, Computing and Physics and Associate Dean of Graduate Studies and Research in the School of Informatics and Computing, Indiana University. He has taught and led many research groups at Caltech and Syracuse University, previously. He received his Ph.D. from Cambridge University, U.K. Fox is well known for his comprehensive work and extensive publications in parallel architecture, distributed programming, grid computing, web services, and Internet applications. His book on *Grid Computing* (coauthored with F. Berman and Tony Hey) is widely used by the research community. He has produced over 60 Ph.D. students in physics, computer science, and engineering over the years. Contact him via: gcf@indiana.edu.

Jack J. Dongarra is a University Distinguished Professor of Electrical Engineering and Computer Science, University of Tennessee, a Distinguished Research Staff, Oak Ridge National Laboratory, and a Turning Fellow at the University of Manchester. An ACM/IEEE/SIAM/AAAS Fellow, Dongarra pioneered the areas of supercomputer benchmarks, numerical analysis, PDE solvers, and high-performance computing and published extensively in these areas. He leads the Linpack benchmark evaluation of the Top-500 fastest computers over the years. Based on his high contributions in the supercomputing and high-performance areas, he was elected as a Member of the National Academy of Engineering in the U.S. Contact him at dongarra@eecs.utk.edu.

Contents

PART

1 Systems Modeling, Clustering, and Virtualization

The first three chapters cover systems models and review two enabling technologies. We model distributed systems and cloud platforms in Chapter 1. The clustering technology is presented in Chapter 2, and virtualization technology in Chapter 3. These two technologies enable distributed and cloud computing. The system models include computer clusters, computing grid, P2P networks, and cloud computing platform. System clustering is supported by hardware, software, and middleware advances. Virtualization creates virtual machines, virtualized clusters, automation of datacenters, and building of elastic cloud platforms.

CHAPTER 1: DISTRIBUTED SYSTEM MODELS AND ENABLING TECHNOLOGIES

This introductory chapter assesses the evolutional changes in computing and IT trends in the past 30 years. We study both high-performance computing (HPC) for scientific computing and high-throughput computing (HTC) systems for business computing. We examine clusters/MPP, grids, P2P networks, and Internet clouds. These systems are distinguished by their platform architectures, OS platforms, processing algorithms, communication protocols, security demands, and service models applied. The study emphasizes the scalability, performance, availability, security, energy-efficiency, workload outsourcing, data center protection, and so on.

This chapter is authored by Kai Hwang with partial contributions by Geoffrey Fox (Section 1.4.1) and Albert Zomaya (Section 1.5.4), and assisted by Nikzad Rivandi, Young-Choon Lee, Xiaosong Lou, and Lizhong Chen. The final manuscript was edited by Jack Dongarra.

CHAPTER 2: COMPUTER CLUSTERS FOR SCALABLE PARALLEL COMPUTING

Clustering enables the construction of scalable parallel and distributed systems for both HPC and HTC applications. Today's cluster nodes are built with either physical servers or virtual machines. In this chapter, we study clustered computing systems and massively parallel processors. We focus on design principles and assess the hardware, software, and middleware support needed. We study scalability, availability, programmability, single system image, job management, and fault tolerance. We study the clustering MPP architectures in three top supercomputers reported in recent years, namely China's Tiahe-1A, the Cray XT5 Jaguar, and IBM's RoadRunner.

This chapter is coauthored by Kai Hwang and Jack Dongarra with partial contributions by Rajkumar Buyya and Ninghui Sun. Special technical assistances are from Zhiwei Xu, Zhou Zhao, Xiaosong Lou, and Lizhong Chen.

CHAPTER 3: VIRTUAL MACHINES AND VIRTUALIZATION OF CLUSTERS AND DATA CENTERS

Virtualization technology was primarily designed for the sharing of expensive hardware resources by multiplexing virtual machines (VM) on the same set of hardware hosts. The surge of interest in installing virtual machines has widened the scope of system applications and upgraded computer performance and efficiency in recent years. We study VMs, live migration of VMs, virtual cluster construction, resource provisioning, virtual configuration adaptation, and the design of virtualized data centers for cloud computing applications. We emphasize the roles of using virtual clusters and virtualized resource management for building dynamic grids and cloud platforms.

This chapter is coauthored by Zhibin Yu and Kai Hwang with technical assistance from Hai Jin, Xiaofei Liao, Chongyuan Qin, Lizhong Chen, and Zhou Zhao.

CHAPTER

Distributed System Models and Enabling Technologies

1

CHAPTER OUTLINE

SUMMARY

This chapter presents the evolutionary changes that have occurred in parallel, distributed, and cloud computing over the past 30 years, driven by applications with variable workloads and large data sets. We study both high-performance and high-throughput computing systems in parallel computers appearing as computer clusters, service-oriented architecture, computational grids, peer-to-peer networks, Internet clouds, and the Internet of Things. These systems are distinguished by their hardware architectures, OS platforms, processing algorithms, communication protocols, and service models applied. We also introduce essential issues on the scalability, performance, availability, security, and energy efficiency in distributed systems.

1.1 SCALABLE COMPUTING OVER THE INTERNET

Over the past 60 years, computing technology has undergone a series of platform and environment changes. In this section, we assess evolutionary changes in machine architecture, operating system platform, network connectivity, and application workload. Instead of using a centralized computer to solve computational problems, a parallel and distributed computing system uses multiple computers to solve large-scale problems over the Internet. Thus, distributed computing becomes data-intensive and network-centric. This section identifies the applications of modern computer systems that practice parallel and distributed computing. These large-scale Internet applications have significantly enhanced the quality of life and information services in society today.

1.1.1 The Age of Internet Computing

Billions of people use the Internet every day. As a result, supercomputer sites and large data centers must provide high-performance computing services to huge numbers of Internet users concurrently. Because of this high demand, the Linpack Benchmark for *high-performance computing (HPC)* applications is no longer optimal for measuring system performance. The emergence of computing clouds instead demands *high-throughput computing (HTC)* systems built with parallel and distributed computing technologies [5,6,19,25]. We have to upgrade data centers using fast servers, storage systems, and high-bandwidth networks. The purpose is to advance network-based computing and web services with the emerging new technologies.

1.1.1.1 The Platform Evolution

Computer technology has gone through five generations of development, with each generation lasting from 10 to 20 years. Successive generations are overlapped in about 10 years. For instance, from 1950 to 1970, a handful of mainframes, including the IBM 360 and CDC 6400, were built to satisfy the demands of large businesses and government organizations. From 1960 to 1980, lower-cost minicomputers such as the DEC PDP 11 and VAX Series became popular among small businesses and on college campuses.

From 1970 to 1990, we saw widespread use of personal computers built with VLSI microprocessors. From 1980 to 2000, massive numbers of portable computers and pervasive devices appeared in both wired and wireless applications. Since 1990, the use of both HPC and HTC systems hidden in

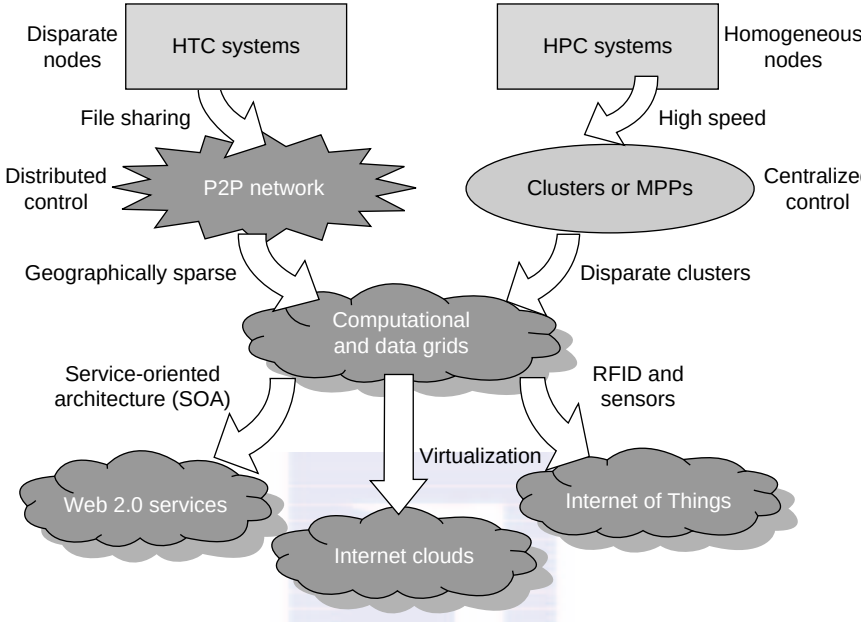


FIGURE 1.1

Evolutionary trend toward parallel, distributed, and cloud computing with clusters, MPPs, P2P networks, grids, clouds, web services, and the Internet of Things.

clusters, grids, or Internet clouds has proliferated. These systems are employed by both consumers and high-end web-scale computing and information services.

The general computing trend is to leverage shared web resources and massive amounts of data over the Internet. Figure 1.1 illustrates the evolution of HPC and HTC systems. On the HPC side, supercomputers (*massively parallel processors* or *MPPs*) are gradually replaced by clusters of cooperative computers out of a desire to share computing resources. The cluster is often a collection of homogeneous compute nodes that are physically connected in close range to one another. We will discuss clusters, MPPs, and grid systems in more detail in Chapters 2 and 7.

On the HTC side, *peer-to-peer (P2P)* networks are formed for distributed file sharing and content delivery applications. A P2P system is built over many client machines (a concept we will discuss further in Chapter 5). Peer machines are globally distributed in nature. P2P, cloud computing, and web service platforms are more focused on HTC applications than on HPC applications. Clustering and P2P technologies lead to the development of computational grids or data grids.

1.1.1.2 High-Performance Computing

For many years, HPC systems emphasize the raw speed performance. The speed of HPC systems has increased from Gflops in the early 1990s to now Pflops in 2010. This improvement was driven mainly by the demands from scientific, engineering, and manufacturing communities. For example,

the Top 500 most powerful computer systems in the world are measured by floating-point speed in Linpack benchmark results. However, the number of supercomputer users is limited to less than 10% of all computer users. Today, the majority of computer users are using desktop computers or large servers when they conduct Internet searches and market-driven computing tasks.

1.1.1.3 High-Throughput Computing

The development of market-oriented high-end computing systems is undergoing a strategic change from an HPC paradigm to an HTC paradigm. This HTC paradigm pays more attention to high-flux computing. The main application for high-flux computing is in Internet searches and web services by millions or more users simultaneously. The performance goal thus shifts to measure *high throughput* or the number of tasks completed per unit of time. HTC technology needs to not only improve in terms of batch processing speed, but also address the acute problems of cost, energy savings, security, and reliability at many data and enterprise computing centers. This book will address both HPC and HTC systems to meet the demands of all computer users.

1.1.1.4 Three New Computing Paradigms

As Figure 1.1 illustrates, with the introduction of SOA, Web 2.0 services become available. Advances in virtualization make it possible to see the growth of Internet clouds as a new computing paradigm. The maturity of *radio-frequency identification (RFID)*, *Global Positioning System (GPS)*, and sensor technologies has triggered the development of the *Internet of Things (IoT)*. These new paradigms are only briefly introduced here. We will study the details of SOA in Chapter 5; virtualization in Chapter 3; cloud computing in Chapters 4, 6, and 9; and the IoT along with cyber-physical systems (CPS) in Chapter 9.

When the Internet was introduced in 1969, Leonard Klienrock of UCLA declared: "As of now, computer networks are still in their infancy, but as they grow up and become sophisticated, we will probably see the spread of computer utilities, which like present electric and telephone utilities, will service individual homes and offices across the country." Many people have redefined the term "computer" since that time. In 1984, John Gage of Sun Microsystems created the slogan, "The network is the computer." In 2008, David Patterson of UC Berkeley said, "The data center is the computer. There are dramatic differences between developing software for millions to use as a service versus distributing software to run on their PCs." Recently, Rajkumar Buyya of Melbourne University simply said: "The cloud is the computer."

This book covers clusters, MPPs, P2P networks, grids, clouds, web services, social networks, and the IoT. In fact, the differences among clusters, grids, P2P systems, and clouds may blur in the future. Some people view clouds as grids or clusters with modest changes through virtualization. Others feel the changes could be major, since clouds are anticipated to process huge data sets generated by the traditional Internet, social networks, and the future IoT. In subsequent chapters, the distinctions and dependencies among all distributed and cloud systems models will become clearer and more transparent.

1.1.1.5 Computing Paradigm Distinctions

The high-technology community has argued for many years about the precise definitions of centralized computing, parallel computing, distributed computing, and cloud computing. In general, *distributed computing* is the opposite of *centralized computing*. The field of *parallel computing*

overlaps with distributed computing to a great extent, and *cloud computing* overlaps with distributed, centralized, and parallel computing. The following list defines these terms more clearly; their architectural and operational differences are discussed further in subsequent chapters.

- **Centralized computing** This is a computing paradigm by which all computer resources are centralized in one physical system. All resources (processors, memory, and storage) are fully shared and tightly coupled within one integrated OS. Many data centers and supercomputers are *centralized systems*, but they are used in parallel, distributed, and cloud computing applications [18,26].
- **Parallel computing** In parallel computing, all processors are either tightly coupled with centralized shared memory or loosely coupled with distributed memory. Some authors refer to this discipline as *parallel processing* [15,27]. Interprocessor communication is accomplished through shared memory or via message passing. A computer system capable of parallel computing is commonly known as a *parallel computer* [28]. Programs running in a parallel computer are called *parallel programs*. The process of writing *parallel programs* is often referred to as *parallel programming* [32].
- **Distributed computing** This is a field of computer science/engineering that studies distributed systems. A *distributed system* [8,13,37,46] consists of multiple autonomous computers, each having its own private memory, communicating through a computer network. Information exchange in a distributed system is accomplished through *message passing*. A computer program that runs in a distributed system is known as a *distributed program*. The process of writing distributed programs is referred to as *distributed programming*.
- **Cloud computing** An *Internet cloud* of resources can be either a centralized or a distributed computing system. The cloud applies parallel or distributed computing, or both. Clouds can be built with physical or virtualized resources over large data centers that are centralized or distributed. Some authors consider cloud computing to be a form of *utility computing* or *service computing* [11,19].

As an alternative to the preceding terms, some in the high-tech community prefer the term *concurrent computing* or *concurrent programming*. These terms typically refer to the union of parallel computing and distributing computing, although biased practitioners may interpret them differently. *Ubiquitous computing* refers to computing with pervasive devices at any place and time using wired or wireless communication. The *Internet of Things* (IoT) is a networked connection of everyday objects including computers, sensors, humans, etc. The IoT is supported by Internet clouds to achieve ubiquitous computing with any object at any place and time. Finally, the term *Internet computing* is even broader and covers all computing paradigms over the Internet. This book covers all the aforementioned computing paradigms, placing more emphasis on distributed and cloud computing and their working systems, including the clusters, grids, P2P, and cloud systems.

1.1.1.6 Distributed System Families

Since the mid-1990s, technologies for building P2P networks and *networks of clusters* have been consolidated into many national projects designed to establish wide area computing infrastructures, known as *computational grids* or *data grids*. Recently, we have witnessed a surge in interest in exploring Internet cloud resources for data-intensive applications. Internet clouds are the result of moving desktop computing to service-oriented computing using server clusters and huge databases

at data centers. This chapter introduces the basics of various parallel and distributed families. Grids and clouds are disparity systems that place great emphasis on resource sharing in hardware, software, and data sets.

Design theory, enabling technologies, and case studies of these massively distributed systems are also covered in this book. Massively distributed systems are intended to exploit a high degree of parallelism or concurrency among many machines. In October 2010, the highest performing cluster machine was built in China with 86016 CPU processor cores and 3,211,264 GPU cores in a Tianhe-1A system. The largest computational grid connects up to hundreds of server clusters. A typical P2P network may involve millions of client machines working simultaneously. Experimental cloud computing clusters have been built with thousands of processing nodes. We devote the material in Chapters 4 through 6 to cloud computing. Case studies of HTC systems will be examined in Chapters 4 and 9, including data centers, social networks, and virtualized cloud platforms

In the future, both HPC and HTC systems will demand multicore or many-core processors that can handle large numbers of computing threads per core. Both HPC and HTC systems emphasize parallelism and distributed computing. Future HPC and HTC systems must be able to satisfy this huge demand in computing power in terms of throughput, efficiency, scalability, and reliability. The system efficiency is decided by speed, programming, and energy factors (i.e., *throughput per watt* of energy consumed). Meeting these goals requires to yield the following design objectives:

- **Efficiency** measures the utilization rate of resources in an execution model by exploiting massive parallelism in HPC. For HTC, efficiency is more closely related to job throughput, data access, storage, and power efficiency.
- **Dependability** measures the reliability and self-management from the chip to the system and application levels. The purpose is to provide high-throughput service with Quality of Service (QoS) assurance, even under failure conditions.
- **Adaptation in the programming model** measures the ability to support billions of job requests over massive data sets and virtualized cloud resources under various workload and service models.
- **Flexibility in application deployment** measures the ability of distributed systems to run well in both HPC (science and engineering) and HTC (business) applications.

1.1.2 Scalable Computing Trends and New Paradigms

Several predictable trends in technology are known to drive computing applications. In fact, designers and programmers want to predict the technological capabilities of future systems. For instance, Jim Gray's paper, "Rules of Thumb in Data Engineering," is an excellent example of how technology affects applications and vice versa. In addition, Moore's law indicates that processor speed doubles every 18 months. Although Moore's law has been proven valid over the last 30 years, it is difficult to say whether it will continue to be true in the future.

Gilder's law indicates that network bandwidth has doubled each year in the past. Will that trend continue in the future? The tremendous price/performance ratio of commodity hardware was driven by the desktop, notebook, and tablet computing markets. This has also driven the adoption and use of commodity technologies in large-scale computing. We will discuss the future of these computing trends in more detail in subsequent chapters. For now, it's important to understand how distributed

systems emphasize both resource distribution and concurrency or high *degree of parallelism (DoP)*. Let's review the degrees of parallelism before we discuss the special requirements for distributed computing.

1.1.2.1 Degrees of Parallelism

Fifty years ago, when hardware was bulky and expensive, most computers were designed in a bit-serial fashion. In this scenario, *bit-level parallelism (BLP)* converts bit-serial processing to word-level processing gradually. Over the years, users graduated from 4-bit microprocessors to 8-, 16-, 32-, and 64-bit CPUs. This led us to the next wave of improvement, known as *instruction-level parallelism (ILP)*, in which the processor executes multiple instructions simultaneously rather than only one instruction at a time. For the past 30 years, we have practiced ILP through pipelining, superscalar computing, *VLIW (very long instruction word)* architectures, and multithreading. ILP requires branch prediction, dynamic scheduling, speculation, and compiler support to work efficiently.

Data-level parallelism (DLP) was made popular through *SIMD (single instruction, multiple data)* and vector machines using vector or array types of instructions. DLP requires even more hardware support and compiler assistance to work properly. Ever since the introduction of multicore processors and *chip multiprocessors (CMPs)*, we have been exploring *task-level parallelism (TLP)*. A modern processor explores all of the aforementioned parallelism types. In fact, BLP, ILP, and DLP are well supported by advances in hardware and compilers. However, TLP is far from being very successful due to difficulty in programming and compilation of code for efficient execution on multicore CMPs. As we move from parallel processing to distributed processing, we will see an increase in computing granularity to *job-level parallelism (JLP)*. It is fair to say that coarse-grain parallelism is built on top of fine-grain parallelism.

1.1.2.2 Innovative Applications

Both HPC and HTC systems desire transparency in many application aspects. For example, data access, resource allocation, process location, concurrency in execution, job replication, and failure recovery should be made transparent to both users and system management. Table 1.1 highlights a few key applications that have driven the development of parallel and distributed systems over the

Table 1.1 Applications of High-Performance and High-Throughput Systems

Domain	Specific Applications
Science and engineering	Scientific simulations, genomic analysis, etc. Earthquake prediction, global warming, weather forecasting, etc.
Business, education, services industry, and health care	Telecommunication, content delivery, e-commerce, etc. Banking, stock exchanges, transaction processing, etc. Air traffic control, electric power grids, distance education, etc. Health care, hospital automation, telemedicine, etc.
Internet and web services, and government applications	Internet search, data centers, decision-making systems, etc. Traffic monitoring, worm containment, cyber security, etc. Digital government, online tax return processing, social networking, etc.
Mission-critical applications	Military command and control, intelligent systems, crisis management, etc.

years. These applications spread across many important domains in science, engineering, business, education, health care, traffic control, Internet and web services, military, and government applications.

Almost all applications demand computing economics, web-scale data collection, system reliability, and scalable performance. For example, distributed transaction processing is often practiced in the banking and finance industry. Transactions represent 90 percent of the existing market for reliable banking systems. Users must deal with multiple database servers in distributed transactions. Maintaining the consistency of replicated transaction records is crucial in real-time banking services. Other complications include lack of software support, network saturation, and security threats in these applications. We will study applications and software support in more detail in subsequent chapters.

1.1.2.3 The Trend toward Utility Computing

Figure 1.2 identifies major computing paradigms to facilitate the study of distributed systems and their applications. These paradigms share some common characteristics. First, they are all ubiquitous in daily life. Reliability and scalability are two major design objectives in these computing models. Second, they are aimed at autonomic operations that can be self-organized to support dynamic discovery. Finally, these paradigms are composable with QoS and *SLAs (service-level agreements)*. These paradigms and their attributes realize the computer utility vision.

Utility computing focuses on a business model in which customers receive computing resources from a paid service provider. All grid/cloud platforms are regarded as utility service providers. However, cloud computing offers a broader concept than utility computing. Distributed cloud applications run on any available servers in some edge networks. Major technological challenges include all aspects of computer science and engineering. For example, users demand new network-efficient processors, scalable memory and storage schemes, distributed OSes, middleware for machine virtualization, new programming models, effective resource management, and application

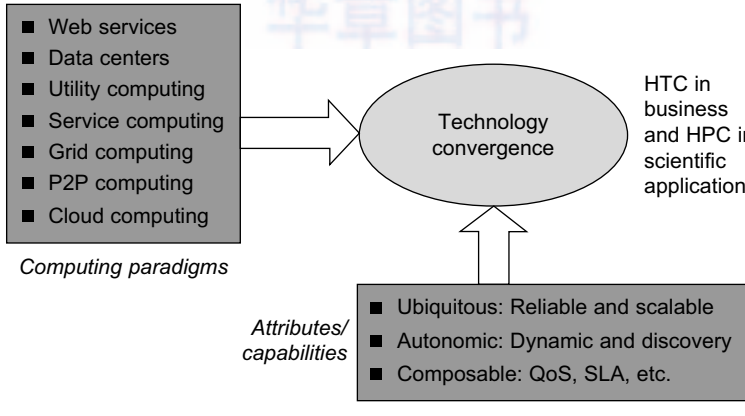


FIGURE 1.2

The vision of computer utilities in modern distributed computing systems.

(*Modified from presentation slide by Raj Buyya, 2010*)

program development. These hardware and software supports are necessary to build distributed systems that explore massive parallelism at all processing levels.

1.1.2.4 The Hype Cycle of New Technologies

Any new and emerging computing and information technology may go through a hype cycle, as illustrated in Figure 1.3. This cycle shows the expectations for the technology at five different stages. The expectations rise sharply from the trigger period to a high peak of inflated expectations. Through a short period of disillusionment, the expectation may drop to a valley and then increase steadily over a long enlightenment period to a plateau of productivity. The number of years for an emerging technology to reach a certain stage is marked by special symbols. The hollow circles indicate technologies that will reach mainstream adoption in two years. The gray circles represent technologies that will reach mainstream adoption in two to five years. The solid circles represent those that require five to 10 years to reach mainstream adoption, and the triangles denote those that require more than 10 years. The crossed circles represent technologies that will become obsolete before they reach the plateau.

The hype cycle in Figure 1.3 shows the technology status as of August 2010. For example, at that time *consumer-generated media* was at the disillusionment stage, and it was predicted to take less than two years to reach its plateau of adoption. *Internet micropayment systems* were forecast to take two to five years to move from the enlightenment stage to maturity. It was believed that *3D printing* would take five to 10 years to move from the rising expectation stage to mainstream adoption, and *mesh network sensors* were expected to take more than 10 years to move from the inflated expectation stage to a plateau of mainstream adoption.

Also as shown in Figure 1.3, the *cloud technology* had just crossed the peak of the expectation stage in 2010, and it was expected to take two to five more years to reach the productivity stage. However, *broadband over power line* technology was expected to become obsolete before leaving the valley of disillusionment stage in 2010. Many additional technologies (denoted by dark circles in Figure 1.3) were at their peak expectation stage in August 2010, and they were expected to take five to 10 years to reach their plateau of success. Once a technology begins to climb the slope of enlightenment, it may reach the productivity plateau within two to five years. Among these promising technologies are the clouds, biometric authentication, interactive TV, speech recognition, predictive analytics, and media tablets.

1.1.3 The Internet of Things and Cyber-Physical Systems

In this section, we will discuss two Internet development trends: the Internet of Things [48] and cyber-physical systems. These evolutionary trends emphasize the extension of the Internet to everyday objects. We will only cover the basics of these concepts here; we will discuss them in more detail in Chapter 9.

1.1.3.1 The Internet of Things

The traditional Internet connects machines to machines or web pages to web pages. The concept of the IoT was introduced in 1999 at MIT [40]. The IoT refers to the networked interconnection of everyday objects, tools, devices, or computers. One can view the IoT as a wireless network of sensors that interconnect all things in our daily life. These things can be large or small and they vary

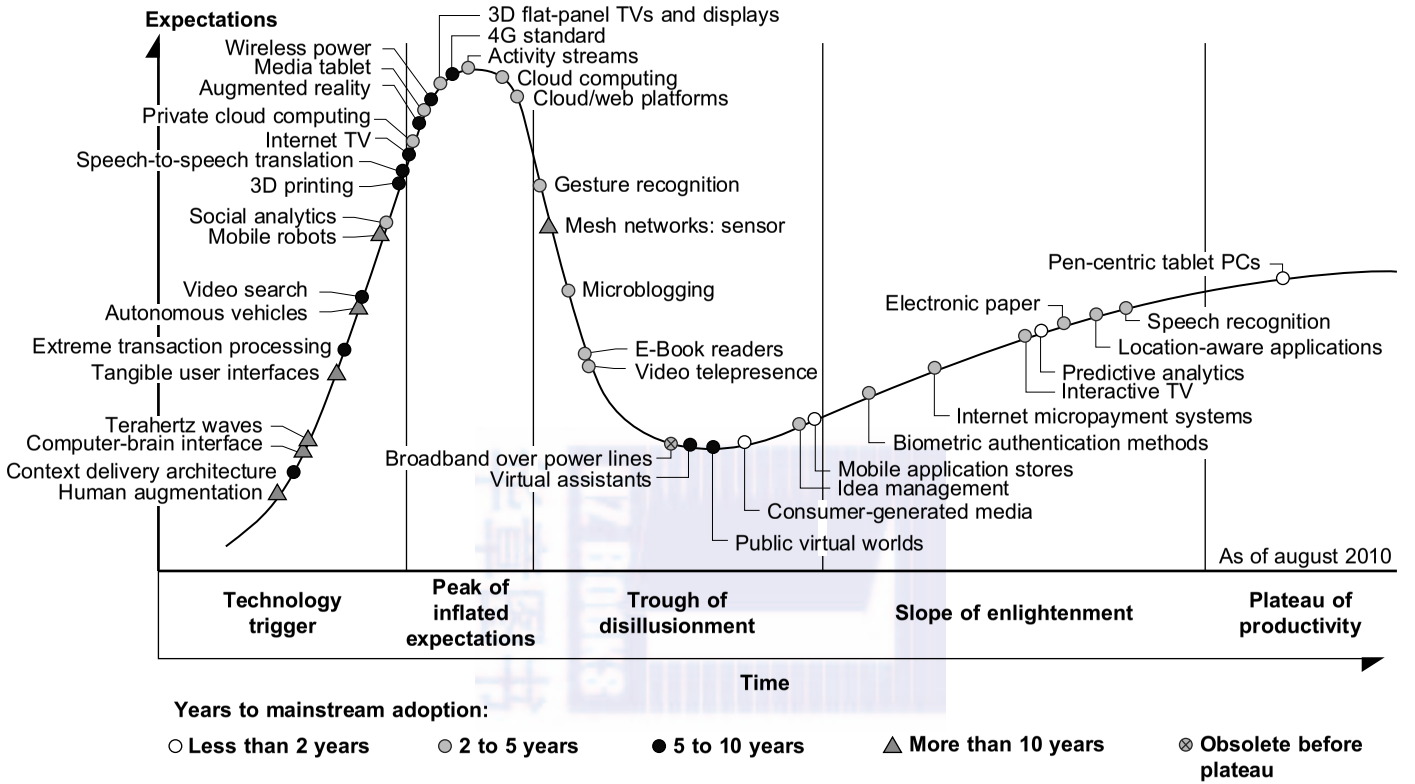


FIGURE 1.3

Hype cycle for Emerging Technologies, 2010.

Hype Cycle Disclaimer

The Hype Cycle is copyrighted 2010 by Gartner, Inc. and its affiliates and is reused with permission. Hype Cycles are graphical representations of the relative maturity of technologies, IT methodologies and management disciplines. They are intended solely as a research tool, and not as a specific guide to action. Gartner disclaims all warranties, express or implied, with respect to this research, including any warranties of merchantability or fitness for a particular purpose.

This Hype Cycle graphic was published by Gartner, Inc. as part of a larger research note and should be evaluated in the context of the entire report. The Gartner report is available at http://www.gartner.com/it/page.jsp?id=1447613.

(*Source: Gartner Press Release "Gartner's 2010 Hype Cycle Special Report Evaluates Maturity of 1,800 Technologies" 7 October 2010.*)

with respect to time and place. The idea is to tag every object using RFID or a related sensor or electronic technology such as GPS.

With the introduction of the IPv6 protocol, 2^{128} IP addresses are available to distinguish all the objects on Earth, including all computers and pervasive devices. The IoT researchers have estimated that every human being will be surrounded by 1,000 to 5,000 objects. The IoT needs to be designed to track 100 trillion static or moving objects simultaneously. The IoT demands universal addressability of all of the objects or things. To reduce the complexity of identification, search, and storage, one can set the threshold to filter out fine-grain objects. The IoT obviously extends the Internet and is more heavily developed in Asia and European countries.

In the IoT era, all objects and devices are instrumented, interconnected, and interacted with each other intelligently. This communication can be made between people and things or among the things themselves. Three communication patterns co-exist: namely H2H (human-to-human), H2T (human-to-thing), and T2T (thing-to-thing). Here things include machines such as PCs and mobile phones. The idea here is to connect things (including human and machine objects) at any time and any place intelligently with low cost. Any place connections include at the PC, indoor (away from PC), outdoors, and on the move. Any time connections include daytime, night, outdoors and indoors, and on the move as well.

The dynamic connections will grow exponentially into a new dynamic network of networks, called the *Internet of Things* (IoT). The IoT is still in its infancy stage of development. Many prototype IoTs with restricted areas of coverage are under experimentation at the time of this writing. Cloud computing researchers expect to use the cloud and future Internet technologies to support fast, efficient, and intelligent interactions among humans, machines, and any objects on Earth. A smart Earth should have intelligent cities, clean water, efficient power, convenient transportation, good food supplies, responsible banks, fast telecommunications, green IT, better schools, good health care, abundant resources, and so on. This dream living environment may take some time to reach fruition at different parts of the world.

1.1.3.2 Cyber-Physical Systems

A *cyber-physical system* (CPS) is the result of interaction between computational processes and the physical world. A CPS integrates "cyber" (heterogeneous, asynchronous) with "physical" (concurrent and information-dense) objects. A CPS merges the "3C" technologies of *computation, communication*, and *control* into an intelligent closed feedback system between the physical world and the information world, a concept which is actively explored in the United States. The IoT emphasizes various networking connections among physical objects, while the CPS emphasizes exploration of *virtual reality* (VR) applications in the physical world. We may transform how we interact with the physical world just like the Internet transformed how we interact with the virtual world. We will study IoT, CPS, and their relationship to cloud computing in Chapter 9.

1.2 TECHNOLOGIES FOR NETWORK-BASED SYSTEMS

With the concept of scalable computing under our belt, it's time to explore hardware, software, and network technologies for distributed computing system design and applications. In particular, we will focus on viable approaches to building distributed operating systems for handling massive parallelism in a distributed environment.

1.2.1 Multicore CPUs and Multithreading Technologies

Consider the growth of component and network technologies over the past 30 years. They are crucial to the development of HPC and HTC systems. In Figure 1.4, processor speed is measured in *millions of instructions per second* (MIPS) and network bandwidth is measured in *megabits per second (Mbps)* or *gigabits per second* (Gbps). The unit *GE* refers to 1 Gbps Ethernet bandwidth.

1.2.1.1 Advances in CPU Processors

Today, advanced CPUs or microprocessor chips assume a multicore architecture with dual, quad, six, or more processing cores. These processors exploit parallelism at ILP and TLP levels. Processor speed growth is plotted in the upper curve in Figure 1.4 across generations of microprocessors or CMPs. We see growth from 1 MIPS for the VAX 780 in 1978 to 1,800 MIPS for the Intel Pentium 4 in 2002, up to a 22,000 MIPS peak for the Sun Niagara 2 in 2008. As the figure shows, Moore's law has proven to be pretty accurate in this case. The clock rate for these processors increased from 10 MHz for the Intel 286 to 4 GHz for the Pentium 4 in 30 years.

However, the clock rate reached its limit on CMOS-based chips due to power limitations. At the time of this writing, very few CPU chips run with a clock rate exceeding 5 GHz. In other words, clock rate will not continue to improve unless chip technology matures. This limitation is attributed primarily to excessive heat generation with high frequency or high voltages. The ILP is highly exploited in modern CPU processors. ILP mechanisms include multiple-issue superscalar architecture, dynamic branch prediction, and speculative execution, among others. These ILP techniques demand hardware and compiler support. In addition, DLP and TLP are highly explored in *graphics processing units* (GPUs) that adopt a many-core architecture with hundreds to thousands of simple cores.

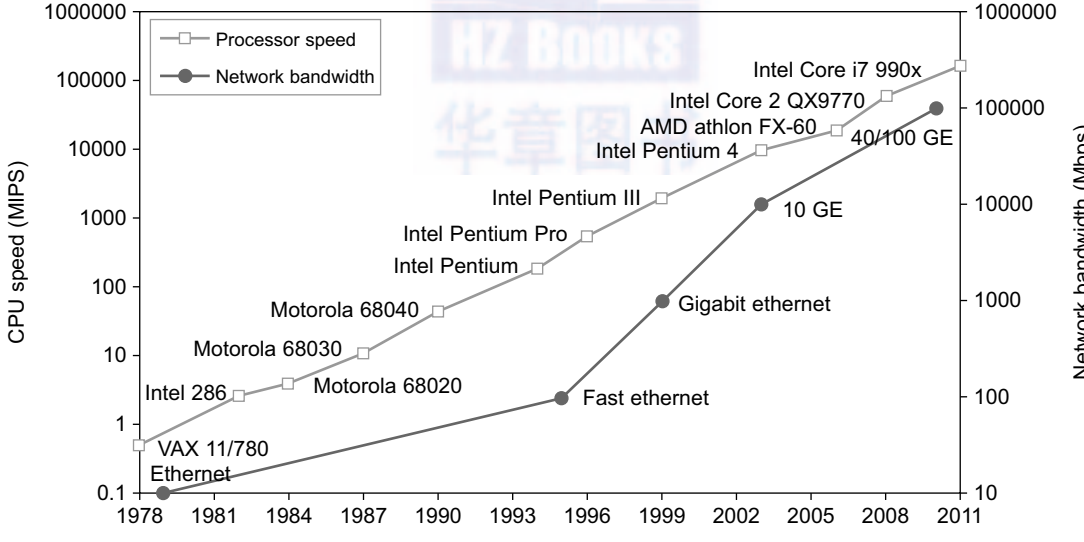


FIGURE 1.4

Improvement in processor and network technologies over 33 years.

(*Courtesy of Xiaosong Lou and Lizhong Chen of University of Southern California, 2011*)

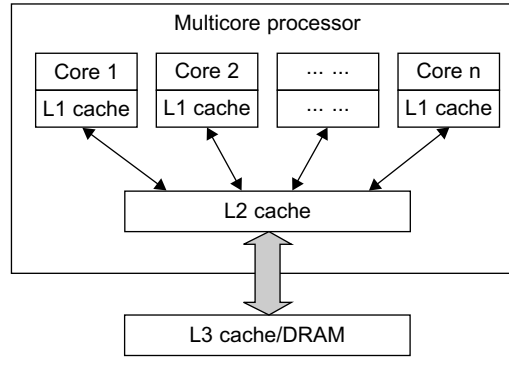


FIGURE 1.5

Schematic of a modern multicore CPU chip using a hierarchy of caches, where L1 cache is private to each core, on-chip L2 cache is shared and L3 cache or DRAM Is off the chip.

Both multi-core CPU and many-core GPU processors can handle multiple instruction threads at different magnitudes today. Figure 1.5 shows the architecture of a typical multicore processor. Each core is essentially a processor with its own private cache (L1 cache). Multiple cores are housed in the same chip with an L2 cache that is shared by all cores. In the future, multiple CMPs could be built on the same CPU chip with even the L3 cache on the chip. Multicore and multithreaded CPUs are equipped with many high-end processors, including the Intel i7, Xeon, AMD Opteron, Sun Niagara, IBM Power 6, and X cell processors. Each core could be also multithreaded. For example, the Niagara II is built with eight cores with eight threads handled by each core. This implies that the maximum ILP and TLP that can be exploited in Niagara is 64 ($8 \times 8 = 64$). In 2011, the Intel Core i7 990x has reported 159,000 MIPS execution rate as shown in the uppermost square in Figure 1.4.

1.2.1.2 Multicore CPU and Many-Core GPU Architectures

Multicore CPUs may increase from the tens of cores to hundreds or more in the future. But the CPU has reached its limit in terms of exploiting massive DLP due to the aforementioned memory wall problem. This has triggered the development of many-core GPUs with hundreds or more thin cores. Both IA-32 and IA-64 instruction set architectures are built into commercial CPUs. Now, x-86 processors have been extended to serve HPC and HTC systems in some high-end server processors.

Many RISC processors have been replaced with multicore x-86 processors and many-core GPUs in the Top 500 systems. This trend indicates that x-86 upgrades will dominate in data centers and supercomputers. The GPU also has been applied in large clusters to build supercomputers in MPPs. In the future, the processor industry is also keen to develop asymmetric or heterogeneous chip multiprocessors that can house both fat CPU cores and thin GPU cores on the same chip.

1.2.1.3 Multithreading Technology

Consider in Figure 1.6 the dispatch of five independent threads of instructions to four pipelined data paths (functional units) in each of the following five processor categories, from left to right: a

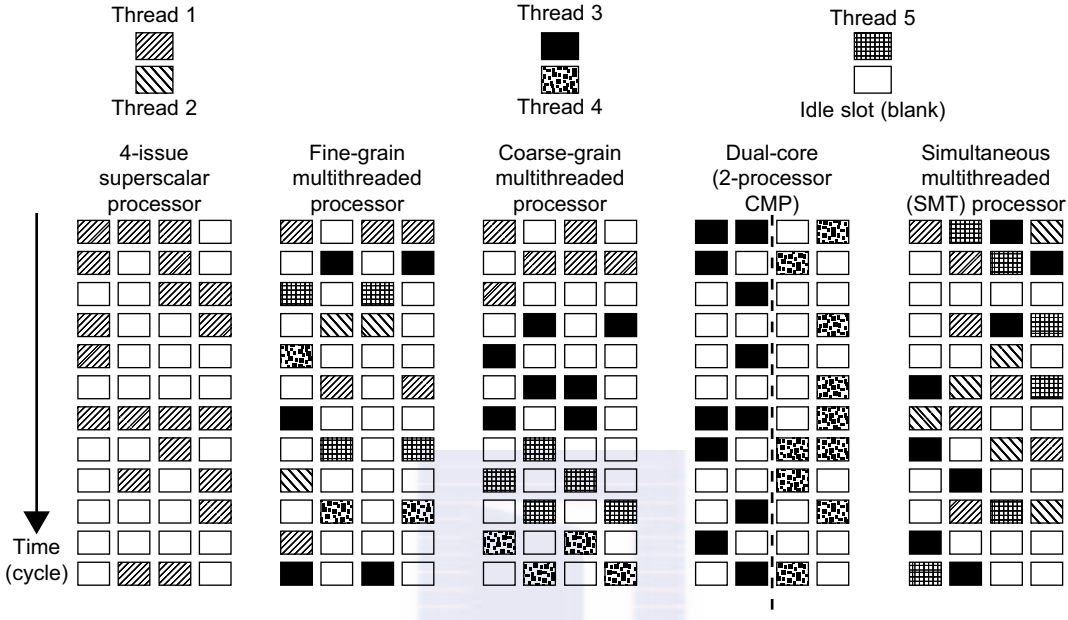


FIGURE 1.6

Five micro-architectures in modern CPU processors, that exploit ILP and TLP supported by multicore and multithreading technologies.

four-issue superscalar processor, a fine-grain multithreaded processor, a coarse-grain multithreaded processor, a two-core CMP, *and a simultaneous multithreaded* (SMT) *processor*. The superscalar processor is single-threaded with four functional units. Each of the three multithreaded processors is four-way multithreaded over four functional data paths. In the dual-core processor, assume two processing cores, each a single-threaded two-way superscalar processor.

Instructions from different threads are distinguished by specific shading patterns for instructions from five independent threads. Typical instruction scheduling patterns are shown here. Only instructions from the same thread are executed in a superscalar processor. Fine-grain multithreading switches the execution of instructions from different threads per cycle. Course-grain multithreading executes many instructions from the same thread for quite a few cycles before switching to another thread. The multicore CMP executes instructions from different threads completely. The SMT allows simultaneous scheduling of instructions from different threads in the same cycle.

These execution patterns closely mimic an ordinary program. The blank squares correspond to no available instructions for an instruction data path at a particular processor cycle. More blank cells imply lower scheduling efficiency. The maximum ILP or maximum TLP is difficult to achieve at each processor cycle. The point here is to demonstrate your understanding of typical instruction scheduling patterns in these five different micro-architectures in modern processors.

1.2.2 GPU Computing to Exascale and Beyond

A GPU is a graphics coprocessor or accelerator mounted on a computer's graphics card or video card. A GPU offloads the CPU from tedious graphics tasks in video editing applications. The world's first GPU, the GeForce 256, was marketed by NVIDIA in 1999. These GPU chips can process a minimum of 10 million polygons per second, and are used in nearly every computer on the market today. Some GPU features were also integrated into certain CPUs. Traditional CPUs are structured with only a few cores. For example, the Xeon X5670 CPU has six cores. However, a modern GPU chip can be built with hundreds of processing cores.

Unlike CPUs, GPUs have a throughput architecture that exploits massive parallelism by executing many concurrent threads slowly, instead of executing a single long thread in a conventional microprocessor very quickly. Lately, parallel GPUs or GPU clusters have been garnering a lot of attention against the use of CPUs with limited parallelism. *General-purpose computing on GPUs*, known as GPGPUs, have appeared in the HPC field. NVIDIA's CUDA model was for HPC using GPGPUs. Chapter 2 will discuss GPU clusters for massively parallel computing in more detail [15,32].

1.2.2.1 How GPUs Work

Early GPUs functioned as coprocessors attached to the CPU. Today, the NVIDIA GPU has been upgraded to 128 cores on a single chip. Furthermore, each core on a GPU can handle eight threads of instructions. This translates to having up to 1,024 threads executed concurrently on a single GPU. This is true massive parallelism, compared to only a few threads that can be handled by a conventional CPU. The CPU is optimized for latency caches, while the GPU is optimized to deliver much higher throughput with explicit management of on-chip memory.

Modern GPUs are not restricted to accelerated graphics or video coding. They are used in HPC systems to power supercomputers with massive parallelism at multicore and multithreading levels. GPUs are designed to handle large numbers of floating-point operations in parallel. In a way, the GPU offloads the CPU from all data-intensive calculations, not just those that are related to video processing. Conventional GPUs are widely used in mobile phones, game consoles, embedded systems, PCs, and servers. The NVIDIA CUDA Tesla or Fermi is used in GPU clusters or in HPC systems for parallel processing of massive floating-pointing data.

1.2.2.2 GPU Programming Model

Figure 1.7 shows the interaction between a CPU and GPU in performing parallel execution of floating-point operations concurrently. The CPU is the conventional multicore processor with limited parallelism to exploit. The GPU has a many-core architecture that has hundreds of simple processing cores organized as multiprocessors. Each core can have one or more threads. Essentially, the CPU's floating-point kernel computation role is largely offloaded to the many-core GPU. The CPU instructs the GPU to perform massive data processing. The bandwidth must be matched between the on-board main memory and the on-chip GPU memory. This process is carried out in NVIDIA's CUDA programming using the GeForce 8800 or Tesla and Fermi GPUs. We will study the use of CUDA GPUs in large-scale cluster computing in Chapter 2.

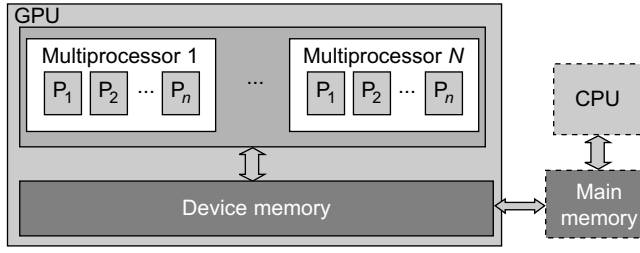


FIGURE 1.7

The use of a GPU along with a CPU for massively parallel execution in hundreds or thousands of processing cores.

(*Courtesy of B. He, et al., PACT'08 [23]*)

Example 1.1 The NVIDIA Fermi GPU Chip with 512 CUDA Cores

In November 2010, three of the five fastest supercomputers in the world (the Tianhe-1a, Nebulae, and Tsubame) used large numbers of GPU chips to accelerate floating-point computations. Figure 1.8 shows the architecture of the Fermi GPU, a next-generation GPU from NVIDIA. This is a *streaming multiprocessor (SM)* module. Multiple SMs can be built on a single GPU chip. The Fermi chip has 16 SMs implemented with 3 billion transistors. Each SM comprises up to 512 *streaming processors (SPs)*, known as *CUDA cores*. The Tesla GPUs used in the Tianhe-1a have a similar architecture, with 448 CUDA cores.

The Fermi GPU is a newer generation of GPU, first appearing in 2011. The Tesla or Fermi GPU can be used in desktop workstations to accelerate floating-point calculations or for building large-scale data centers. The architecture shown is based on a 2009 white paper by NVIDIA [36]. There are 32 CUDA cores per SM. Only one SM is shown in Figure 1.8. Each CUDA core has a simple pipelined integer ALU and an FPU that can be used in parallel. Each SM has 16 load/store units allowing source and destination addresses to be calculated for 16 threads per clock. There are four *special function units (SFUs)* for executing transcendental instructions.

All functional units and CUDA cores are interconnected by an *NoC (network on chip)* to a large number of SRAM banks (L2 caches). Each SM has a 64 KB L1 cache. The 768 KB unified L2 cache is shared by all SMs and serves all load, store, and texture operations. *Memory controllers* are used to connect to 6 GB of off-chip DRAMs. The SM schedules threads in groups of 32 parallel threads called *warps*. In total, 256/512 *FMA (fused multiply and add)* operations can be done in parallel to produce 32/64-bit floating-point results. The 512 CUDA cores in an SM can work in parallel to deliver up to 515 Gflops of double-precision results, if fully utilized. With 16 SMs, a single GPU has a peak speed of 82.4 Tflops. Only 12 Fermi GPUs have the potential to reach the Pflops performance.

In the future, thousand-core GPUs may appear in Exascale (Eflops or 10^{18} flops) systems. This reflects a trend toward building future MPPs with hybrid architectures of both types of processing chips. In a DARPA report published in September 2008, four challenges are identified for exascale computing: (1) energy and power, (2) memory and storage, (3) concurrency and locality, and (4) system resiliency. Here, we see the progress of GPUs along with CPU advances in power

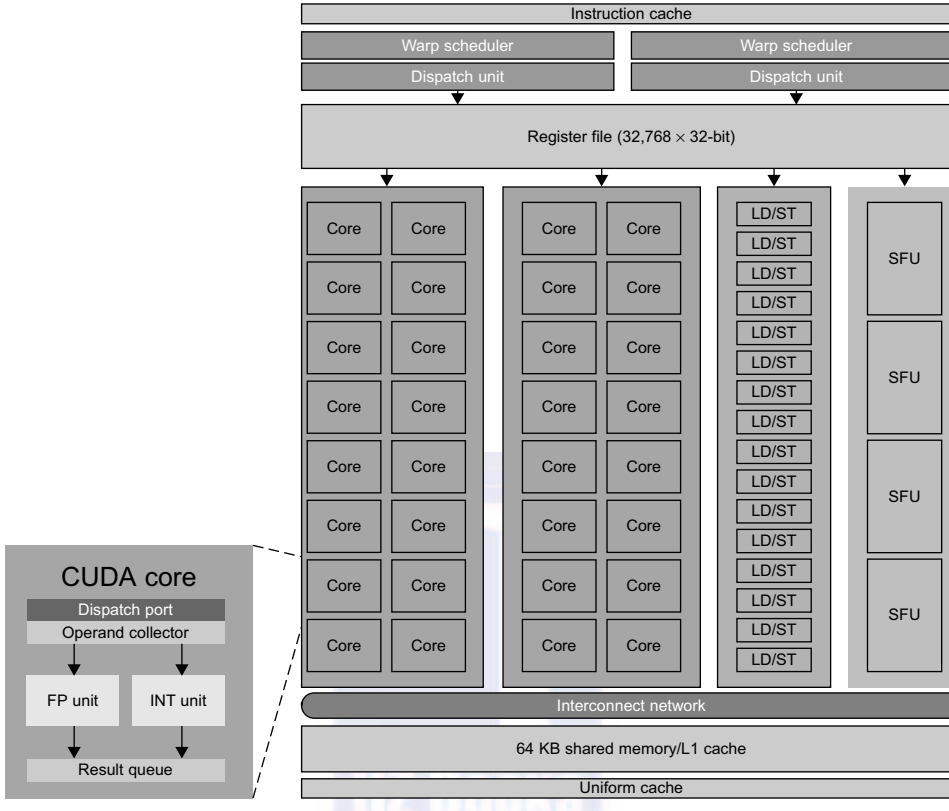


FIGURE 1.8

NVIDIA Fermi GPU built with 16 streaming multiprocessors (SMs) of 32 CUDA cores each; only one SM Is shown. More details can be found also in [49].

(*Courtesy of NVIDIA, 2009 [36] 2011*)

efficiency, performance, and programmability [16]. In Chapter 2, we will discuss the use of GPUs to build large clusters.

1.2.2.3 Power Efficiency of the GPU

Bill Dally of Stanford University considers power and massive parallelism as the major benefits of GPUs over CPUs for the future. By extrapolating current technology and computer architecture, it was estimated that 60 Gflops/watt per core is needed to run an exaflops system (see Figure 1.10). Power constrains what we can put in a CPU or GPU chip. Dally has estimated that the CPU chip consumes about 2 nJ/instruction, while the GPU chip requires 200 pJ/instruction, which is 1/10 less than that of the CPU. The CPU is optimized for latency in caches and memory, while the GPU is optimized for throughput with explicit management of on-chip memory.

Figure 1.9 compares the CPU and GPU in their performance/power ratio measured in Gflops/watt per core. In 2010, the GPU had a value of 5 Gflops/watt at the core level, compared with less

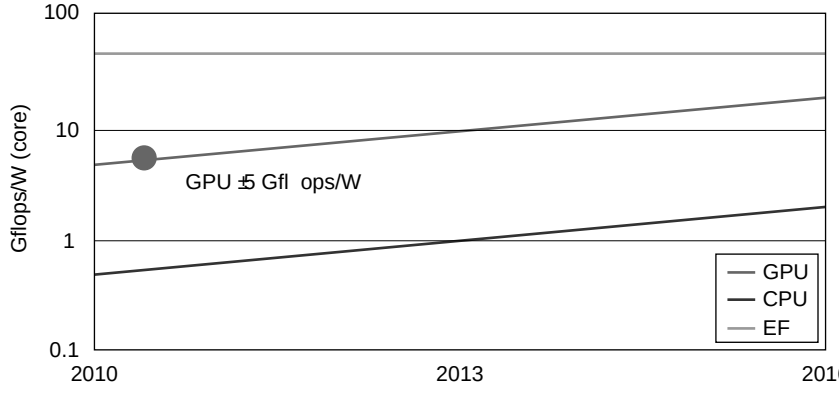


FIGURE 1.9

The GPU performance (middle line, measured 5 Gflops/W/core in 2011), compared with the lower CPU performance (lower line measured 0.8 Gflops/W/core in 2011) and the estimated 60 Gflops/W/core performance in 2011 for the Exascale (EF in upper curve) in the future.

(*Courtesy of Bill Dally [15]*)

than 1 Gflop/watt per CPU core. This may limit the scaling of future supercomputers. However, the GPUs may close the gap with the CPUs. Data movement dominates power consumption. One needs to optimize the storage hierarchy and tailor the memory to the applications. We need to promote self-aware OS and runtime support and build locality-aware compilers and auto-tuners for GPU-based MPPs. This implies that both power and software are the real challenges in future parallel and distributed computing systems.

1.2.3 Memory, Storage, and Wide-Area Networking

1.2.3.1 Memory Technology

The upper curve in Figure 1.10 plots the growth of DRAM chip capacity from 16 KB in 1976 to 64 GB in 2011. This shows that memory chips have experienced a 4x increase in capacity every three years. Memory access time did not improve much in the past. In fact, the memory wall problem is getting worse as the processor gets faster. For hard drives, capacity increased from 260 MB in 1981 to 250 GB in 2004. The Seagate Barracuda XT hard drive reached 3 TB in 2011. This represents an approximately 10x increase in capacity every eight years. The capacity increase of disk arrays will be even greater in the years to come. Faster processor speed and larger memory capacity result in a wider gap between processors and memory. The memory wall may become even worse a problem limiting the CPU performance in the future.

1.2.3.2 Disks and Storage Technology

Beyond 2011, disks or disk arrays have exceeded 3 TB in capacity. The lower curve in Figure 1.10 shows the disk storage growth in 7 orders of magnitude in 33 years. The rapid growth of flash memory and *solid-state drives* (SSDs) also impacts the future of HPC and HTC systems. The mortality rate of SSD is not bad at all. A typical SSD can handle 300,000 to 1 million write cycles per

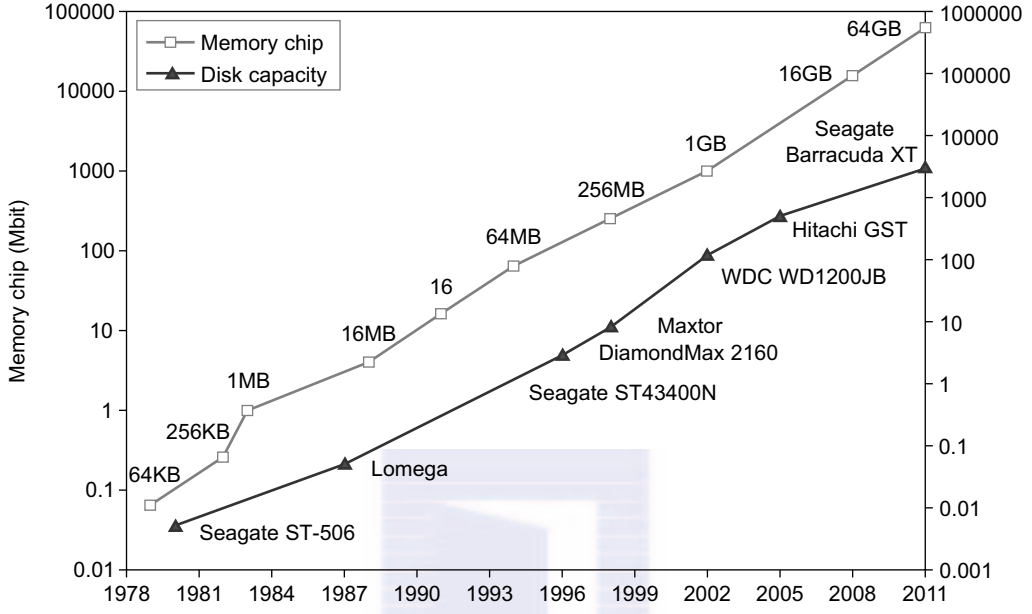


FIGURE 1.10

Improvement in memory and disk technologies over 33 years. The Seagate Barracuda XT disk has a capacity of 3 TB in 2011.

(*Courtesy of Xiaosong Lou and Lizhong Chen of University of Southern California, 2011*)

block. So the SSD can last for several years, even under conditions of heavy write usage. Flash and SSD will demonstrate impressive speedups in many applications.

Eventually, power consumption, cooling, and packaging will limit large system development. Power increases linearly with respect to clock frequency and quadratic ally with respect to voltage applied on chips. Clock rate cannot be increased indefinitely. Lowered voltage supplies are very much in demand. Jim Gray once said in an invited talk at the University of Southern California, "*Tape units are dead, disks are tape units, flashes are disks, and memory are caches now.*" This clearly paints the future for disk and storage technology. In 2011, the SSDs are still too expensive to replace stable disk arrays in the storage market.

1.2.3.3 System-Area Interconnects

The nodes in small clusters are mostly interconnected by an Ethernet switch or a *local area network* (LAN). As Figure 1.11 shows, a LAN typically is used to connect client hosts to big servers. A *storage area network* (SAN) connects servers to network storage such as disk arrays. *Network attached storage* (NAS) connects client hosts directly to the disk arrays. All three types of networks often appear in a large cluster built with commercial network components. If no large distributed storage is shared, a small cluster could be built with a multiport Gigabit Ethernet switch plus copper cables to link the end machines. All three types of networks are commercially available.

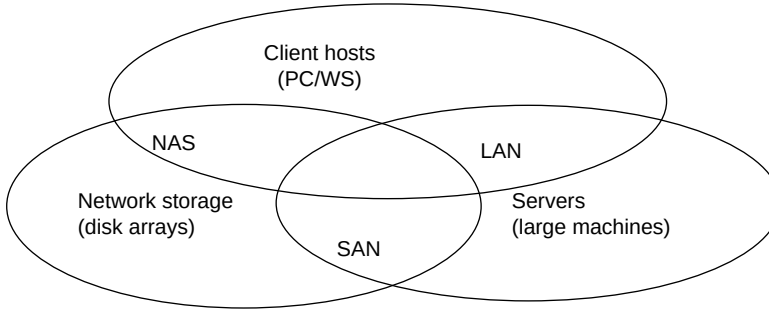


FIGURE 1.11

Three interconnection networks for connecting servers, client hosts, and storage devices; the LAN connects client hosts and servers, the SAN connects servers with disk arrays, and the NAS connects clients with large storage systems in the network environment.

1.2.3.4 Wide-Area Networking

The lower curve in Figure 1.10 plots the rapid growth of Ethernet bandwidth from 10 Mbps in 1979 to 1 Gbps in 1999, and 40 ~ 100 GE in 2011. It has been speculated that 1 Tbps network links will become available by 2013. According to Berman, Fox, and Hey [6], network links with 1,000, 1,000, 100, 10, and 1 Gbps bandwidths were reported, respectively, for international, national, organization, optical desktop, and copper desktop connections in 2006.

An increase factor of two per year on network performance was reported, which is faster than Moore's law on CPU speed doubling every 18 months. The implication is that more computers will be used concurrently in the future. High-bandwidth networking increases the capability of building massively distributed systems. The IDC 2010 report predicted that both InfiniBand and Ethernet will be the two major interconnect choices in the HPC arena. Most data centers are using Gigabit Ethernet as the interconnect in their server clusters.

1.2.4 Virtual Machines and Virtualization Middleware

A conventional computer has a single OS image. This offers a rigid architecture that tightly couples application software to a specific hardware platform. Some software running well on one machine may not be executable on another platform with a different instruction set under a fixed OS. *Virtual machines* (VMs) offer novel solutions to underutilized resources, application inflexibility, software manageability, and security concerns in existing physical machines.

Today, to build large clusters, grids, and clouds, we need to access large amounts of computing, storage, and networking resources in a virtualized manner. We need to aggregate those resources, and hopefully, offer a single system image. In particular, a cloud of provisioned resources must rely on virtualization of processors, memory, and I/O facilities dynamically. We will cover virtualization in Chapter 3. However, the basic concepts of virtualized resources, such as VMs, virtual storage, and virtual networking and their virtualization software or middleware, need to be introduced first. Figure 1.12 illustrates the architectures of three VM configurations.

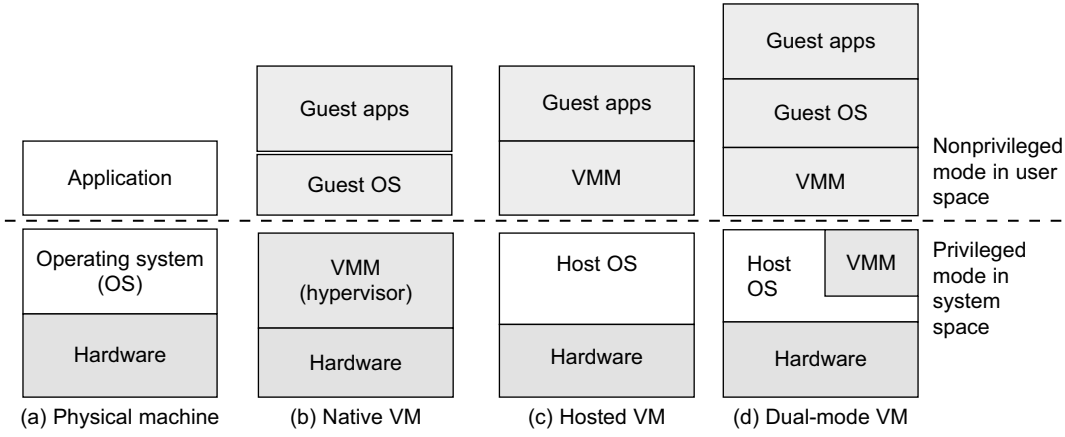


FIGURE 1.12

Three VM architectures in (b), (c), and (d), compared with the traditional physical machine shown in (a).

(*Courtesy of M. Abde-Majeed and S. Kulkarni, 2009 USC*)

1.2.4.1 Virtual Machines

In Figure 1.12, the host machine is equipped with the physical hardware, as shown at the bottom of the figure. An example is an x-86 architecture desktop running its installed Windows OS, as shown in part (a) of the figure. The VM can be provisioned for any hardware system. The VM is built with virtual resources managed by a guest OS to run a specific application. Between the VMs and the host platform, one needs to deploy a middleware layer called a *virtual machine monitor (VMM)*. Figure 1.12(b) shows a native VM installed with the use of a VMM called a *hypervisor* in privileged mode. For example, the hardware has x-86 architecture running the Windows system.

The guest OS could be a Linux system and the hypervisor is the XEN system developed at Cambridge University. This hypervisor approach is also called *bare-metal VM*, because the hypervisor handles the bare hardware (CPU, memory, and I/O) directly. Another architecture is the host VM shown in Figure 1.12(c). Here the VMM runs in nonprivileged mode. The host OS need not be modified. The VM can also be implemented with a dual mode, as shown in Figure 1.12(d). Part of the VMM runs at the user level and another part runs at the supervisor level. In this case, the host OS may have to be modified to some extent. Multiple VMs can be ported to a given hardware system to support the virtualization process. The VM approach offers hardware independence of the OS and applications. The user application running on its dedicated OS could be bundled together as a *virtual appliance* that can be ported to any hardware platform. The VM could run on an OS different from that of the host computer.

1.2.4.2 VM Primitive Operations

The VMM provides the VM abstraction to the guest OS. With full virtualization, the VMM exports a VM abstraction identical to the physical machine so that a standard OS such as Windows 2000 or Linux can run just as it would on the physical hardware. Low-level VMM operations are indicated by Mendel Rosenblum [41] and illustrated in Figure 1.13.

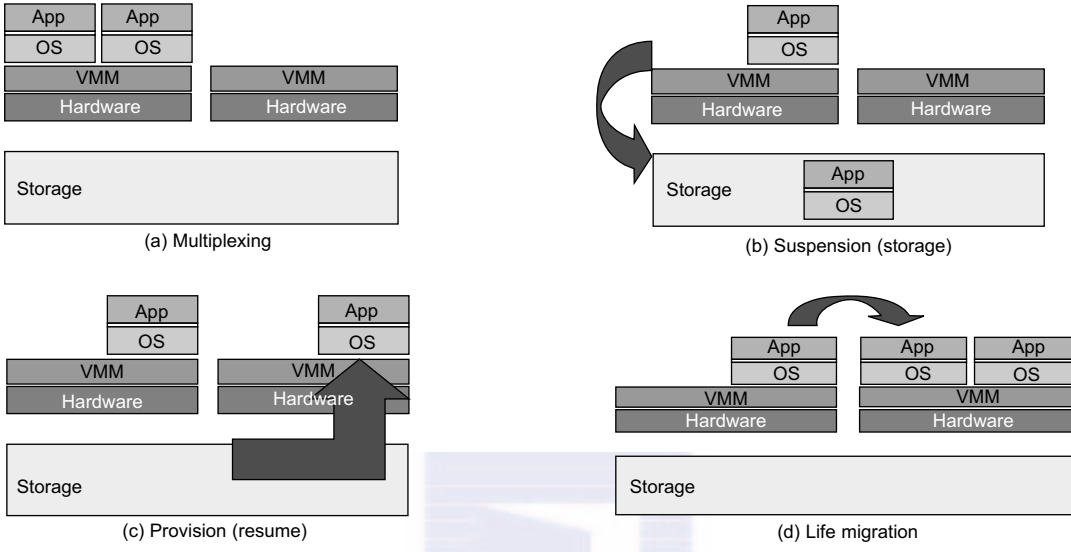


FIGURE 1.13

VM multiplexing, suspension, provision, and migration in a distributed computing environment.
(*Courtesy of M. Rosenblum, Keynote address, ACM ASPLOS 2006 [41]*)

- First, the VMs can be multiplexed between hardware machines, as shown in Figure 1.13(a).
- Second, a VM can be suspended and stored in stable storage, as shown in Figure 1.13(b).
- Third, a suspended VM can be resumed or provisioned to a new hardware platform, as shown in Figure 1.13(c).
- Finally, a VM can be migrated from one hardware platform to another, as shown in Figure 1.13(d).

These VM operations enable a VM to be provisioned to any available hardware platform. They also enable flexibility in porting distributed application executions. Furthermore, the VM approach will significantly enhance the utilization of server resources. Multiple server functions can be consolidated on the same hardware platform to achieve higher system efficiency. This will eliminate server sprawl via deployment of systems as VMs, which move transparency to the shared hardware. With this approach, VMware claimed that server utilization could be increased from its current 5–15 percent to 60–80 percent.

1.2.4.3 Virtual Infrastructures

Physical resources for compute, storage, and networking at the bottom of Figure 1.14 are mapped to the needy applications embedded in various VMs at the top. Hardware and software are then separated. Virtual infrastructure is what connects resources to distributed applications. It is a dynamic mapping of system resources to specific applications. The result is decreased costs and increased efficiency and responsiveness. Virtualization for server consolidation and containment is a good example of this. We will discuss VMs and virtualization support in Chapter 3. Virtualization support for clusters, clouds, and grids is covered in Chapters 3, 4, and 7, respectively.

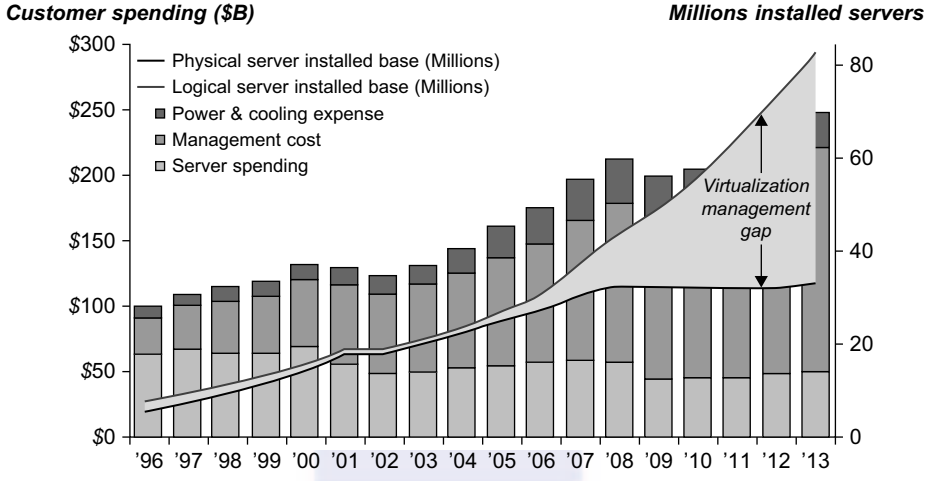


FIGURE 1.14

Growth and cost breakdown of data centers over the years.

(*Source: IDC Report, 2009*)

1.2.5 Data Center Virtualization for Cloud Computing

In this section, we discuss basic architecture and design considerations of data centers. Cloud architecture is built with commodity hardware and network devices. Almost all cloud platforms choose the popular x86 processors. Low-cost terabyte disks and Gigabit Ethernet are used to build data centers. Data center design emphasizes the performance/price ratio over speed performance alone. In other words, storage and energy efficiency are more important than shear speed performance. Figure 1.13 shows the server growth and cost breakdown of data centers over the past 15 years. Worldwide, about 43 million servers are in use as of 2010. The cost of utilities exceeds the cost of hardware after three years.

1.2.5.1 Data Center Growth and Cost Breakdown

A large data center may be built with thousands of servers. Smaller data centers are typically built with hundreds of servers. The cost to build and maintain data center servers has increased over the years. According to a 2009 IDC report (see Figure 1.14), typically only 30 percent of data center costs goes toward purchasing IT equipment (such as servers and disks), 33 percent is attributed to the chiller, 18 percent to the *uninterruptible power supply (UPS)*, 9 percent to *computer room air conditioning (CRAC)*, and the remaining 7 percent to power distribution, lighting, and transformer costs. Thus, about 60 percent of the cost to run a data center is allocated to management and maintenance. The server purchase cost did not increase much with time. The cost of electricity and cooling did increase from 5 percent to 14 percent in 15 years.

1.2.5.2 Low-Cost Design Philosophy

High-end switches or routers may be too cost-prohibitive for building data centers. Thus, using high-bandwidth networks may not fit the economics of cloud computing. Given a fixed budget,

commodity switches and networks are more desirable in data centers. Similarly, using commodity x86 servers is more desired over expensive mainframes. The software layer handles network traffic balancing, fault tolerance, and expandability. Currently, nearly all cloud computing data centers use Ethernet as their fundamental network technology.

1.2.5.3 Convergence of Technologies

Essentially, cloud computing is enabled by the convergence of technologies in four areas: (1) hardware virtualization and multi-core chips, (2) utility and grid computing, (3) SOA, Web 2.0, and WS mashups, and (4) atonomic computing and data center automation. Hardware virtualization and multicore chips enable the existence of dynamic configurations in the cloud. Utility and grid computing technologies lay the necessary foundation for computing clouds. Recent advances in SOA, Web 2.0, and mashups of platforms are pushing the cloud another step forward. Finally, achievements in autonomic computing and automated data center operations contribute to the rise of cloud computing.

Jim Gray once posted the following question: "*Science faces a data deluge. How to manage and analyze information?*" This implies that science and our society face the same challenge of data deluge. Data comes from sensors, lab experiments, simulations, individual archives, and the web in all scales and formats. Preservation, movement, and access of massive data sets require generic tools supporting high-performance, scalable file systems, databases, algorithms, workflows, and visualization. With science becoming data-centric, a new paradigm of scientific discovery is becoming based on data-intensive technologies.

On January 11, 2007, the *Computer Science and Telecommunication Board (CSTB)* recommended fostering tools for data capture, data creation, and data analysis. A cycle of interaction exists among four technical areas. First, cloud technology is driven by a surge of interest in data deluge. Also, cloud computing impacts e-science greatly, which explores multicore and parallel computing technologies. These two hot areas enable the buildup of data deluge. To support data-intensive computing, one needs to address workflows, databases, algorithms, and virtualization issues.

By linking computer science and technologies with scientists, a spectrum of e-science or e-research applications in biology, chemistry, physics, the social sciences, and the humanities has generated new insights from interdisciplinary activities. Cloud computing is a transformative approach as it promises much more than a data center model. It fundamentally changes how we interact with information. The cloud provides services on demand at the infrastructure, platform, or software level. At the platform level, MapReduce offers a new programming model that transparently handles data parallelism with natural fault tolerance capability. We will discuss MapReduce in more detail in Chapter 6.

Iterative MapReduce extends MapReduce to support a broader range of data mining algorithms commonly used in scientific applications. The cloud runs on an extremely large cluster of commodity computers. Internal to each cluster node, multithreading is practiced with a large number of cores in many-core GPU clusters. Data-intensive science, cloud computing, and multicore computing are converging and revolutionizing the next generation of computing in architectural design and programming challenges. They enable the pipeline: Data becomes information and knowledge, and in turn becomes machine wisdom as desired in SOA.

1.3 SYSTEM MODELS FOR DISTRIBUTED AND CLOUD COMPUTING

Distributed and cloud computing systems are built over a large number of autonomous computer nodes. These node machines are interconnected by SANs, LANs, or WANs in a hierarchical manner. With today's networking technology, a few LAN switches can easily connect hundreds of machines as a working cluster. A WAN can connect many local clusters to form a very large cluster of clusters. In this sense, one can build a massive system with millions of computers connected to edge networks.

Massive systems are considered highly scalable, and can reach web-scale connectivity, either physically or logically. In Table 1.2, massive systems are classified into four groups: *clusters, P2P networks, computing grids*, and *Internet clouds* over huge data centers. In terms of node number, these four system classes may involve hundreds, thousands, or even millions of computers as participating nodes. These machines work collectively, cooperatively, or collaboratively at various levels. The table entries characterize these four system classes in various technical and application aspects.

Table 1.2 Classification of Parallel and Distributed Computing Systems

Functionality, Applications	Computer Clusters [10,28,38]	Peer-to-Peer Networks [34,46]	Data/ Computational Grids [6,18,51]	Cloud Platforms [1,9,11,12,30]
Architecture, Network Connectivity, and Size	Network of compute nodes interconnected by SAN, LAN, or WAN hierarchically	Flexible network of client machines logically connected by an overlay network	Heterogeneous clusters interconnected by high-speed network links over selected resource sites	Virtualized cluster of servers over data centers via SLA
Control and Resources Management	Homogeneous nodes with distributed control, running UNIX or Linux	Autonomous client nodes, free in and out, with self-organization	Centralized control, server-oriented with authenticated security	Dynamic resource provisioning of servers, storage, and networks
Applications and Network-centric Services	High-performance computing, search engines, and web services, etc.	Most appealing to business file sharing, content delivery, and social networking	Distributed supercomputing, global problem solving, and data center services	Upgraded web search, utility computing, and outsourced computing services
Representative Operational Systems	Google search engine, SunBlade, IBM Road Runner, Cray XT4, etc.	Gnutella, eMule, BitTorrent, Napster, KaZaA, Skype, JXTA	TeraGrid, GriPhyN, UK EGEE, D-Grid, ChinaGrid, etc.	Google App Engine, IBM Bluecloud, AWS, and Microsoft Azure

From the application perspective, clusters are most popular in supercomputing applications. In 2009, 417 of the Top 500 supercomputers were built with cluster architecture. It is fair to say that clusters have laid the necessary foundation for building large-scale grids and clouds. P2P networks appeal most to business applications. However, the content industry was reluctant to accept P2P technology for lack of copyright protection in ad hoc networks. Many national grids built in the past decade were underutilized for lack of reliable middleware or well-coded applications. Potential advantages of cloud computing include its low cost and simplicity for both providers and users.

1.3.1 Clusters of Cooperative Computers

A computing cluster consists of interconnected stand-alone computers which work cooperatively as a single integrated computing resource. In the past, clustered computer systems have demonstrated impressive results in handling heavy workloads with large data sets.

1.3.1.1 Cluster Architecture

Figure 1.15 shows the architecture of a typical server cluster built around a low-latency, high-bandwidth interconnection network. This network can be as simple as a SAN (e.g., Myrinet) or a LAN (e.g., Ethernet). To build a larger cluster with more nodes, the interconnection network can be built with multiple levels of Gigabit Ethernet, Myrinet, or InfiniBand switches. Through hierarchical construction using a SAN, LAN, or WAN, one can build scalable clusters with an increasing number of nodes. The cluster is connected to the Internet via a virtual private network (VPN) gateway. The gateway IP address locates the cluster. The system image of a computer is decided by the way the OS manages the shared cluster resources. Most clusters have loosely coupled node computers. All resources of a server node are managed by their own OS. Thus, most clusters have multiple system images as a result of having many autonomous nodes under different OS control.

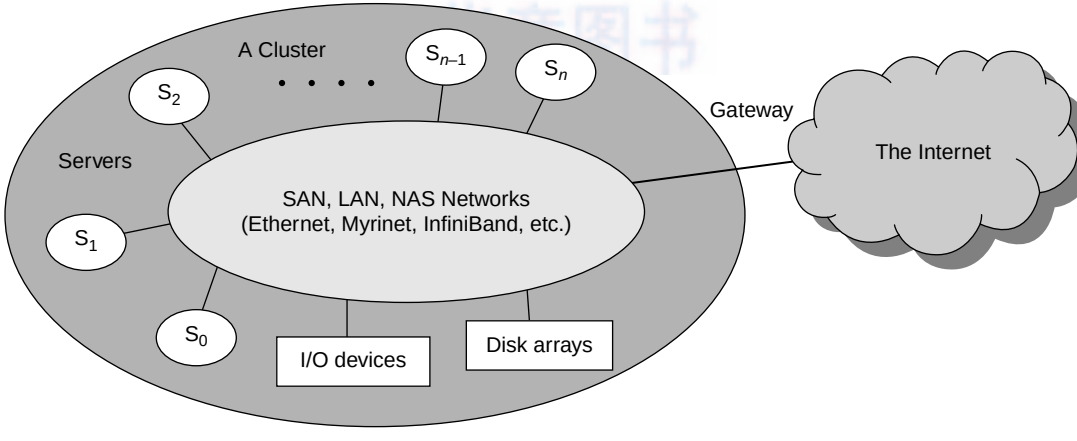


FIGURE 1.15

A cluster of servers interconnected by a high-bandwidth SAN or LAN with shared I/O devices and disk arrays; the cluster acts as a single computer attached to the Internet.

1.3.1.2 Single-System Image

Greg Pfister [38] has indicated that an ideal cluster should merge multiple system images into a *single-system image (SSI)*. Cluster designers desire a *cluster operating system* or some middleware to support SSI at various levels, including the sharing of CPUs, memory, and I/O across all cluster nodes. An SSI is an illusion created by software or hardware that presents a collection of resources as one integrated, powerful resource. SSI makes the cluster appear like a single machine to the user. A cluster with multiple system images is nothing but a collection of independent computers.

1.3.1.3 Hardware, Software, and Middleware Support

In Chapter 2, we will discuss cluster design principles for both small and large clusters. Clusters exploring massive parallelism are commonly known as MPPs. Almost all HPC clusters in the Top 500 list are also MPPs. The building blocks are computer nodes (PCs, workstations, servers, or SMP), special communication software such as PVM or MPI, and a network interface card in each computer node. Most clusters run under the Linux OS. The computer nodes are interconnected by a high-bandwidth network (such as Gigabit Ethernet, Myrinet, InfiniBand, etc.).

Special cluster middleware supports are needed to create SSI or *high availability (HA)*. Both sequential and parallel applications can run on the cluster, and special parallel environments are needed to facilitate use of the cluster resources. For example, distributed memory has multiple images. Users may want all distributed memory to be shared by all servers by forming *distributed shared memory (DSM)*. Many SSI features are expensive or difficult to achieve at various cluster operational levels. Instead of achieving SSI, many clusters are loosely coupled machines. Using virtualization, one can build many virtual clusters dynamically, upon user demand. We will discuss virtual clusters in Chapter 3 and the use of virtual clusters for cloud computing in Chapters 4, 5, 6, and 9.

1.3.1.4 Major Cluster Design Issues

Unfortunately, a cluster-wide OS for complete resource sharing is not available yet. Middleware or OS extensions were developed at the user space to achieve SSI at selected functional levels. Without this middleware, cluster nodes cannot work together effectively to achieve cooperative computing. The software environments and applications must rely on the middleware to achieve high performance. The cluster benefits come from scalable performance, efficient message passing, high system availability, seamless fault tolerance, and cluster-wide job management, as summarized in Table 1.3. We will address these issues in Chapter 2.

1.3.2 Grid Computing Infrastructures

In the past 30 years, users have experienced a natural growth path from Internet to web and grid computing services. Internet services such as the *Telnet* command enables a local computer to connect to a remote computer. A web service such as HTTP enables remote access of remote web pages. Grid computing is envisioned to allow close interaction among applications running on distant computers simultaneously. *Forbes Magazine* has projected the global growth of the IT-based economy from $1 trillion in 2001 to $20 trillion by 2015. The evolution from Internet to web and grid services is certainly playing a major role in this growth.

Table 1.3 Critical Cluster Design Issues and Feasible Implementations

Features	Functional Characterization	Feasible Implementations
Availability and Support	Hardware and software support for sustained HA in cluster	Failover, failback, check pointing, rollback recovery, nonstop OS, etc.
Hardware Fault Tolerance	Automated failure management to eliminate all single points of failure	Component redundancy, hot swapping, RAID, multiple power supplies, etc.
Single System Image (SSI)	Achieving SSI at functional level with hardware and software support, middleware, or OS extensions	Hardware mechanisms or middleware support to achieve DSM at coherent cache level
Efficient Communications	To reduce message-passing system overhead and hide latencies	Fast message passing, active messages, enhanced MPI library, etc.
Cluster-wide Job Management	Using a global job management system with better scheduling and monitoring	Application of single-job management systems such as LSF, Codine, etc.
Dynamic Load Balancing	Balancing the workload of all processing nodes along with failure recovery	Workload monitoring, process migration, job replication and gang scheduling, etc.
Scalability and Programmability	Adding more servers to a cluster or adding more clusters to a grid as the workload or data set increases	Use of scalable interconnect, performance monitoring, distributed execution environment, and better software tools

1.3.2.1 Computational Grids

Like an electric utility power grid, a *computing grid* offers an infrastructure that couples computers, software/middleware, special instruments, and people and sensors together. The grid is often constructed across LAN, WAN, or Internet backbone networks at a regional, national, or global scale. Enterprises or organizations present grids as integrated computing resources. They can also be viewed as *virtual platforms* to support *virtual organizations*. The computers used in a grid are primarily workstations, servers, clusters, and supercomputers. Personal computers, laptops, and PDAs can be used as access devices to a grid system.

Figure 1.16 shows an example computational grid built over multiple resource sites owned by different organizations. The resource sites offer complementary computing resources, including workstations, large servers, a mesh of processors, and Linux clusters to satisfy a chain of computational needs. The grid is built across various IP broadband networks including LANs and WANs already used by enterprises or organizations over the Internet. The grid is presented to users as an integrated resource pool as shown in the upper half of the figure.

Special instruments may be involved such as using the radio telescope in SETI@Home search of life in the galaxy and the austrophysics@Swineburne for pulsars. At the server end, the grid is a network. At the client end, we see wired or wireless terminal devices. The grid integrates the computing, communication, contents, and transactions as rented services. Enterprises and consumers form the user base, which then defines the usage trends and service characteristics. Many national and international grids will be reported in Chapter 7, the NSF

TeraGrid in US, EGEE in Europe, and ChinaGrid in China for various distributed scientific grid applications.

1.3.2.2 Grid Families

Grid technology demands new distributed computing models, software/middleware support, network protocols, and hardware infrastructures. National grid projects are followed by industrial grid platform development by IBM, Microsoft, Sun, HP, Dell, Cisco, EMC, Platform Computing, and others. New *grid service providers* (GSPs) and new grid applications have emerged rapidly, similar to the growth of Internet and web services in the past two decades. In Table 1.4, grid systems are classified in essentially two categories: *computational or data grids* and *P2P grids*. Computing or data grids are built primarily at the national level. In Chapter 7, we will cover grid applications and lessons learned.

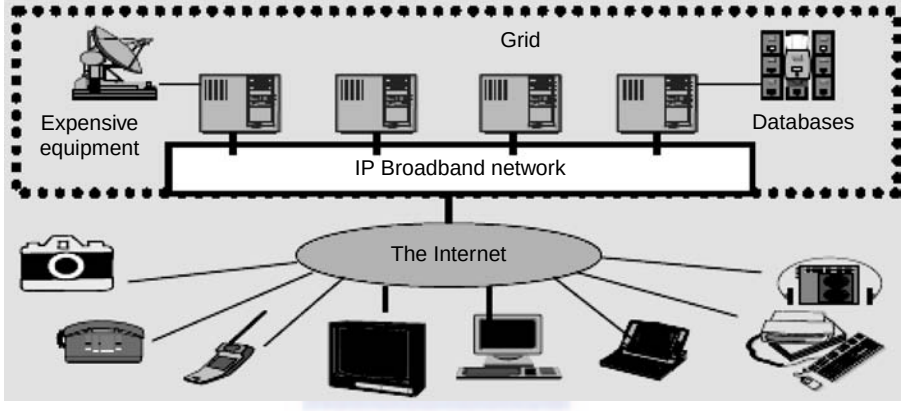


FIGURE 1.16

Computational grid or data grid providing computing utility, data, and information services through resource sharing and cooperation among participating organizations.

(*Courtesy of Z. Xu, Chinese Academy of Science, 2004*)

Table 1.4 Two Grid Computing Infrastructures and Representative Systems

Design Issues	Computational and Data Grids	P2P Grids
Grid Applications Reported	Distributed supercomputing, National Grid initiatives, etc.	Open grid with P2P flexibility, all resources from client machines
Representative Systems	TeraGrid built in US, ChinaGrid in China, and the e-Science grid built in UK	JXTA, FightAid@home, SETI@home
Development Lessons Learned	Restricted user groups, middleware bugs, protocols to acquire resources	Unreliable user-contributed resources, limited to a few apps

1.3.3 Peer-to-Peer Network Families

An example of a well-established distributed system is the *client-server architecture*. In this scenario, client machines (PCs and workstations) are connected to a central server for compute, e-mail, file access, and database applications. The *P2P architecture* offers a distributed model of networked systems. First, a P2P network is client-oriented instead of server-oriented. In this section, P2P systems are introduced at the physical level and overlay networks at the logical level.

1.3.3.1 P2P Systems

In a P2P system, every node acts as both a client and a server, providing part of the system resources. Peer machines are simply client computers connected to the Internet. All client machines act autonomously to join or leave the system freely. This implies that no master-slave relationship exists among the peers. No central coordination or central database is needed. In other words, no peer machine has a global view of the entire P2P system. The system is self-organizing with distributed control.

Figure 1.17 shows the architecture of a P2P network at two abstraction levels. Initially, the peers are totally unrelated. Each peer machine joins or leaves the P2P network voluntarily. Only the participating peers form the *physical network* at any time. Unlike the cluster or grid, a P2P network does not use a dedicated interconnection network. The physical network is simply an ad hoc network formed at various Internet domains randomly using the TCP/IP and NAI protocols. Thus, the physical network varies in size and topology dynamically due to the free membership in the P2P network.

1.3.3.2 Overlay Networks

Data items or files are distributed in the participating peers. Based on communication or file-sharing needs, the peer IDs form an *overlay network* at the logical level. This overlay is a virtual network

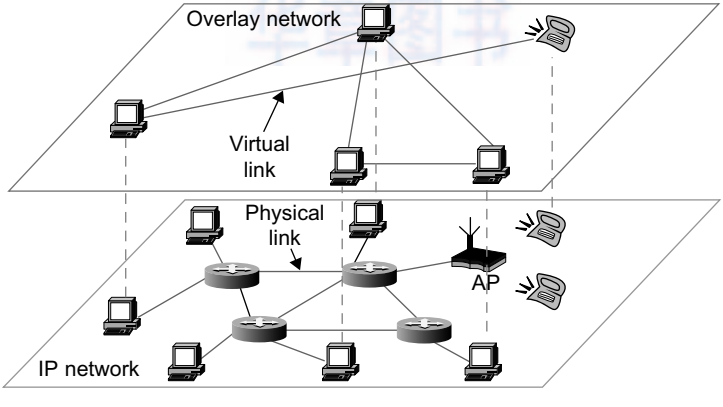


FIGURE 1.17

The structure of a P2P system by mapping a physical IP network to an overlay network built with virtual links.

(*Courtesy of Zhenyu Li, Institute of Computing Technology, Chinese Academy of Sciences, 2010*)

formed by mapping each physical machine with its ID, logically, through a virtual mapping as shown in Figure 1.17. When a new peer joins the system, its peer ID is added as a node in the overlay network. When an existing peer leaves the system, its peer ID is removed from the overlay network automatically. Therefore, it is the P2P overlay network that characterizes the logical connectivity among the peers.

There are two types of overlay networks: *unstructured* and *structured*. An *unstructured overlay network* is characterized by a random graph. There is no fixed route to send messages or files among the nodes. Often, flooding is applied to send a query to all nodes in an unstructured overlay, thus resulting in heavy network traffic and nondeterministic search results. *Structured overlay networks* follow certain connectivity topology and rules for inserting and removing nodes (peer IDs) from the overlay graph. Routing mechanisms are developed to take advantage of the structured overlays.

1.3.3.3 P2P Application Families

Based on application, P2P networks are classified into four groups, as shown in Table 1.5. The first family is for distributed file sharing of digital contents (music, videos, etc.) on the P2P network. This includes many popular P2P networks such as Gnutella, Napster, and BitTorrent, among others. Collaboration P2P networks include MSN or Skype chatting, instant messaging, and collaborative design, among others. The third family is for distributed P2P computing in specific applications. For example, SETI@home provides 25 Tflops of distributed computing power, collectively, over 3 million Internet host machines. Other P2P platforms, such as JXTA, .NET, and FightingAID@home, support naming, discovery, communication, security, and resource aggregation in some P2P applications. We will discuss these topics in more detail in Chapters 8 and 9.

1.3.3.4 P2P Computing Challenges

P2P computing faces three types of heterogeneity problems in hardware, software, and network requirements. There are too many hardware models and architectures to select from; incompatibility exists between software and the OS; and different network connections and protocols

Table 1.5 Major Categories of P2P Network Families [46]

System Features	Distributed File Sharing	Collaborative Platform	Distributed P2P Computing	P2P Platform
Attractive Applications	Content distribution of MP3 music, video, open software, etc.	Instant messaging, collaborative design and gaming	Scientific exploration and social networking	Open networks for public resources
Operational Problems	Loose security and serious online copyright violations	Lack of trust, disturbed by spam, privacy, and peer collusion	Security holes, selfish partners, and peer collusion	Lack of standards or protection protocols
Example Systems	Gnutella, Napster, eMule, BitTorrent, Aimster, KaZaA, etc.	ICQ, AIM, Groove, Magi, Multiplayer Games, Skype, etc.	SETI@home, Geonome@home, etc.	JXTA, .NET, FightingAid@home, etc.

make it too complex to apply in real applications. We need system scalability as the workload increases. System scaling is directly related to performance and bandwidth. P2P networks do have these properties. Data location is also important to affect collective performance. Data locality, network proximity, and interoperability are three design objectives in distributed P2P applications.

P2P performance is affected by routing efficiency and self-organization by participating peers. Fault tolerance, failure management, and load balancing are other important issues in using overlay networks. Lack of trust among peers poses another problem. Peers are strangers to one another. Security, privacy, and copyright violations are major worries by those in the industry in terms of applying P2P technology in business applications [35]. In a P2P network, all clients provide resources including computing power, storage space, and I/O bandwidth. The distributed nature of P2P networks also increases robustness, because limited peer failures do not form a single point of failure.

By replicating data in multiple peers, one can easily lose data in failed nodes. On the other hand, disadvantages of P2P networks do exist. Because the system is not centralized, managing it is difficult. In addition, the system lacks security. Anyone can log on to the system and cause damage or abuse. Further, all client computers connected to a P2P network cannot be considered reliable or virus-free. In summary, P2P networks are reliable for a small number of peer nodes. They are only useful for applications that require a low level of security and have no concern for data sensitivity. We will discuss P2P networks in Chapter 8, and extending P2P technology to social networking in Chapter 9.

1.3.4 Cloud Computing over the Internet

Gordon Bell, Jim Gray, and Alex Szalay [5] have advocated: "Computational science is changing to be data-intensive. Supercomputers must be balanced systems, not just CPU farms but also petascale I/O and networking arrays." In the future, working with large data sets will typically mean sending the computations (programs) to the data, rather than copying the data to the workstations. This reflects the trend in IT of moving computing and data from desktops to large data centers, where there is on-demand provision of software, hardware, and data as a service. This data explosion has promoted the idea of cloud computing.

Cloud computing has been defined differently by many users and designers. For example, IBM, a major player in cloud computing, has defined it as follows: "*A cloud is a pool of virtualized computer resources. A cloud can host a variety of different workloads, including batch-style backend jobs and interactive and user-facing applications*." Based on this definition, a cloud allows workloads to be deployed and scaled out quickly through rapid provisioning of virtual or physical machines. The cloud supports redundant, self-recovering, highly scalable programming models that allow workloads to recover from many unavoidable hardware/software failures. Finally, the cloud system should be able to monitor resource use in real time to enable rebalancing of allocations when needed.

1.3.4.1 Internet Clouds

Cloud computing applies a virtualized platform with elastic resources on demand by provisioning hardware, software, and data sets dynamically (see Figure 1.18). The idea is to move desktop computing to a service-oriented platform using server clusters and huge databases at data centers. Cloud computing leverages its low cost and simplicity to benefit both users and providers. Machine virtualization has enabled such cost-effectiveness. Cloud computing intends to satisfy many user

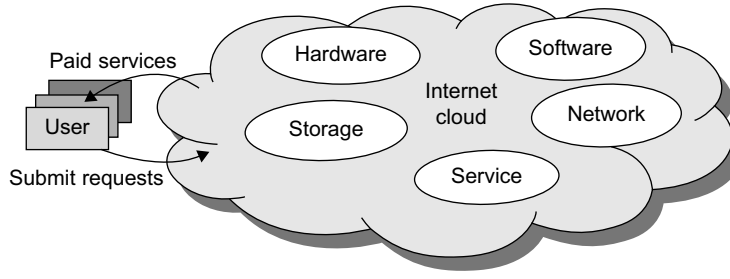


FIGURE 1.18

Virtualized resources from data centers to form an Internet cloud, provisioned with hardware, software, storage, network, and services for paid users to run their applications.

applications simultaneously. The cloud ecosystem must be designed to be secure, trustworthy, and dependable. Some computer users think of the cloud as a centralized resource pool. Others consider the cloud to be a server cluster which practices distributed computing over all the servers used.

1.3.4.2 The Cloud Landscape

Traditionally, a distributed computing system tends to be owned and operated by an autonomous administrative domain (e.g., a research laboratory or company) for on-premises computing needs. However, these traditional systems have encountered several performance bottlenecks: constant system maintenance, poor utilization, and increasing costs associated with hardware/software upgrades. Cloud computing as an on-demand computing paradigm resolves or relieves us from these problems. Figure 1.19 depicts the cloud landscape and major cloud players, based on three cloud service models. Chapters 4, 6, and 9 provide details regarding these cloud service offerings. Chapter 3 covers the relevant virtualization tools.

- **Infrastructure as a Service (IaaS)** This model puts together infrastructures demanded by users—namely servers, storage, networks, and the data center fabric. The user can deploy and run on multiple VMs running guest OSes on specific applications. The user does not manage or control the underlying cloud infrastructure, but can specify when to request and release the needed resources.
- **Platform as a Service (PaaS)** This model enables the user to deploy user-built applications onto a virtualized cloud platform. PaaS includes middleware, databases, development tools, and some runtime support such as Web 2.0 and Java. The platform includes both hardware and software integrated with specific programming interfaces. The provider supplies the API and software tools (e.g., Java, Python, Web 2.0, .NET). The user is freed from managing the cloud infrastructure.
- **Software as a Service (SaaS)** This refers to browser-initiated application software over thousands of paid cloud customers. The SaaS model applies to business processes, industry applications, *consumer relationship management (CRM), enterprise resources planning (ERP), human resources (HR)*, and collaborative applications. On the customer side, there is no upfront investment in servers or software licensing. On the provider side, costs are rather low, compared with conventional hosting of user applications.

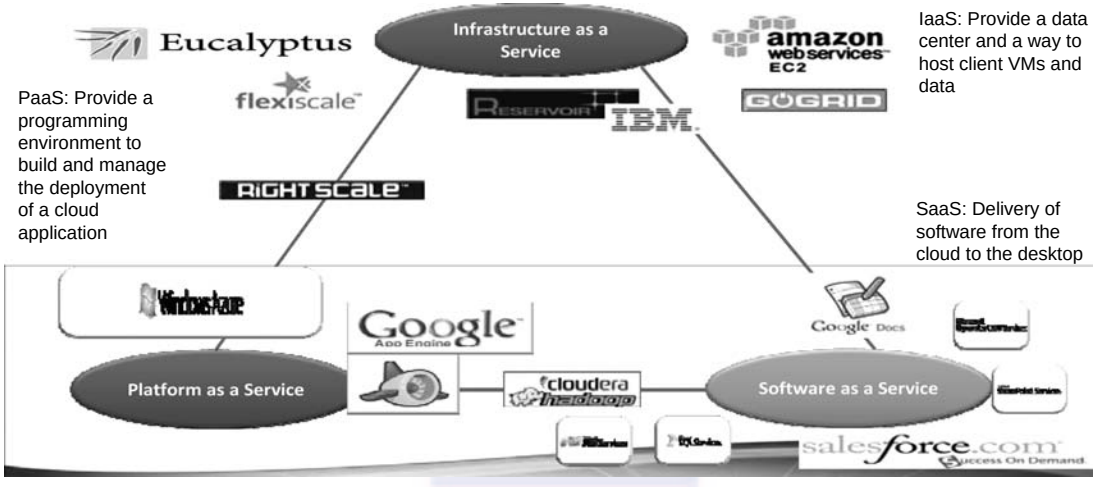


FIGURE 1.19

Three cloud service models in a cloud landscape of major providers.

(*Courtesy of Dennis Gannon, keynote address at Cloudcom2010 [19]*)

Internet clouds offer four deployment modes: *private, public, managed*, and *hybrid* [11]. These modes demand different levels of security implications. The different SLAs imply that the security responsibility is shared among all the cloud providers, the cloud resource consumers, and the third-party cloud-enabled software providers. Advantages of cloud computing have been advocated by many IT experts, industry leaders, and computer science researchers.

In Chapter 4, we will describe major cloud platforms that have been built and various cloud services offerings. The following list highlights eight reasons to adapt the cloud for upgraded Internet applications and web services:

1. Desired location in areas with protected space and higher energy efficiency
2. Sharing of peak-load capacity among a large pool of users, improving overall utilization
3. Separation of infrastructure maintenance duties from domain-specific application development
4. Significant reduction in cloud computing cost, compared with traditional computing paradigms
5. Cloud computing programming and application development
6. Service and data discovery and content/service distribution
7. Privacy, security, copyright, and reliability issues
8. Service agreements, business models, and pricing policies

1.4 SOFTWARE ENVIRONMENTS FOR DISTRIBUTED SYSTEMS AND CLOUDS

This section introduces popular software environments for using distributed and cloud computing systems. Chapters 5 and 6 discuss this subject in more depth.

1.4.1 Service-Oriented Architecture (SOA)

In grids/web services, Java, and CORBA, an entity is, respectively, a service, a Java object, and a CORBA distributed object in a variety of languages. These architectures build on the traditional seven Open Systems Interconnection (OSI) layers that provide the base networking abstractions. On top of this we have a base software environment, which would be .NET or Apache Axis for web services, the Java Virtual Machine for Java, and a broker network for CORBA. On top of this base environment one would build a higher level environment reflecting the special features of the distributed computing environment. This starts with entity interfaces and inter-entity communication, which rebuild the top four OSI layers but at the entity and not the bit level. Figure 1.20 shows the layered architecture for distributed entities used in web services and grid systems.

1.4.1.1 Layered Architecture for Web Services and Grids

The entity interfaces correspond to the *Web Services Description Language* (WSDL), Java method, and CORBA *interface definition language* (IDL) specifications in these example distributed systems. These interfaces are linked with customized, high-level communication systems: SOAP, RMI, and IIOP in the three examples. These communication systems support features including particular message patterns (such as *Remote Procedure Call* or RPC), fault recovery, and specialized routing. Often, these communication systems are built on message-oriented middleware (enterprise bus) infrastructure such as WebSphere MQ or *Java Message Service* (JMS) which provide rich functionality and support virtualization of routing, senders, and recipients.

In the case of fault tolerance, the features in the *Web Services Reliable Messaging (WSRM)* framework mimic the OSI layer capability (as in TCP fault tolerance) modified to match the different abstractions (such as messages versus packets, virtualized addressing) at the entity levels. Security is a critical capability that either uses or reimplements the capabilities seen in concepts such as *Internet Protocol Security* (IPsec) and secure sockets in the OSI layers. Entity communication is supported by higher level services for registries, metadata, and management of the entities discussed in Section 5.4.

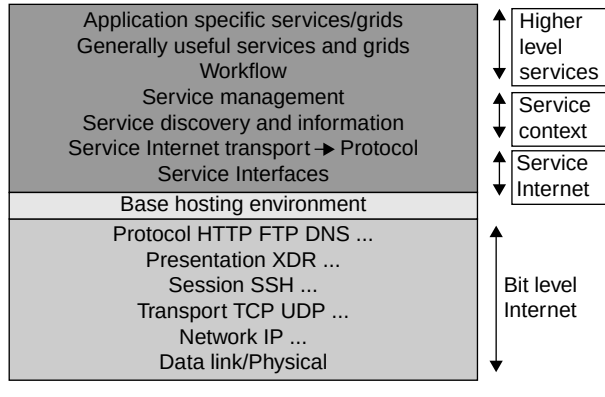


FIGURE 1.20

Layered achitecture for web services and the grids.

Here, one might get several models with, for example, JNDI (*Jini and Java Naming and Directory Interface*) illustrating different approaches within the Java distributed object model. The CORBA Trading Service, UDDI (*Universal Description, Discovery, and Integration*), LDAP (*Lightweight Directory Access Protocol*), and ebXML (*Electronic Business using eXtensible Markup Language*) are other examples of discovery and information services described in Section 5.4. Management services include service state and lifetime support; examples include the CORBA Life Cycle and Persistent states, the different Enterprise JavaBeans models, Jini's lifetime model, and a suite of web services specifications in Chapter 5. The above language or interface terms form a collection of entity-level capabilities.

The latter can have performance advantages and offers a "shared memory" model allowing more convenient exchange of information. However, the distributed model has two critical advantages: namely, higher performance (from multiple CPUs when communication is unimportant) and a cleaner separation of software functions with clear software reuse and maintenance advantages. The distributed model is expected to gain popularity as the default approach to software systems. In the earlier years, CORBA and Java approaches were used in distributed systems rather than today's SOAP, XML, or REST (*Representational State Transfer*).

1.4.1.2 Web Services and Tools

Loose coupling and support of heterogeneous implementations make services more attractive than distributed objects. Figure 1.20 corresponds to two choices of service architecture: web services or REST systems (these are further discussed in Chapter 5). Both web services and REST systems have very distinct approaches to building reliable interoperable systems. In web services, one aims to fully specify all aspects of the service and its environment. This specification is carried with communicated messages using Simple Object Access Protocol (SOAP). The hosting environment then becomes a universal distributed operating system with fully distributed capability carried by SOAP messages. This approach has mixed success as it has been hard to agree on key parts of the protocol and even harder to efficiently implement the protocol by software such as Apache Axis.

In the REST approach, one adopts simplicity as the universal principle and delegates most of the difficult problems to application (implementation-specific) software. In a web services language, REST has minimal information in the header, and the message body (that is opaque to generic message processing) carries all the needed information. REST architectures are clearly more appropriate for rapid technology environments. However, the ideas in web services are important and probably will be required in mature systems at a different level in the stack (as part of the application). Note that REST can use XML schemas but not those that are part of SOAP; "XML over HTTP" is a popular design choice in this regard. Above the communication and management layers, we have the ability to compose new entities or distributed programs by integrating several entities together.

In CORBA and Java, the distributed entities are linked with RPCs, and the simplest way to build composite applications is to view the entities as objects and use the traditional ways of linking them together. For Java, this could be as simple as writing a Java program with method calls replaced by Remote Method Invocation (RMI), while CORBA supports a similar model with a syntax reflecting the C++ style of its entity (object) interfaces. Allowing the term "grid" to refer to a single service or to represent a collection of services, here sensors represent entities that output data (as messages), and grids and clouds represent collections of services that have multiple message-based inputs and outputs.

1.4.1.3 The Evolution of SOA

As shown in Figure 1.21, *service-oriented architecture (SOA)* has evolved over the years. SOA applies to building grids, clouds, grids of clouds, clouds of grids, clouds of clouds (also known as interclouds), and systems of systems in general. A large number of sensors provide data-collection services, denoted in the figure as *SS (sensor service)*. A sensor can be a ZigBee device, a Bluetooth device, a WiFi access point, a personal computer, a GPA, or a wireless phone, among other things. Raw data is collected by sensor services. All the SS devices interact with large or small computers, many forms of grids, databases, the compute cloud, the storage cloud, the filter cloud, the discovery cloud, and so on. *Filter services* (*fs* in the figure) are used to eliminate unwanted raw data, in order to respond to specific requests from the web, the grid, or web services.

A collection of filter services forms a filter cloud. We will cover various clouds for compute, storage, filter, and discovery in Chapters 4, 5, and 6, and various grids, P2P networks, and the IoT in Chapters 7, 8, and 9. SOA aims to search for, or sort out, the useful data from the massive

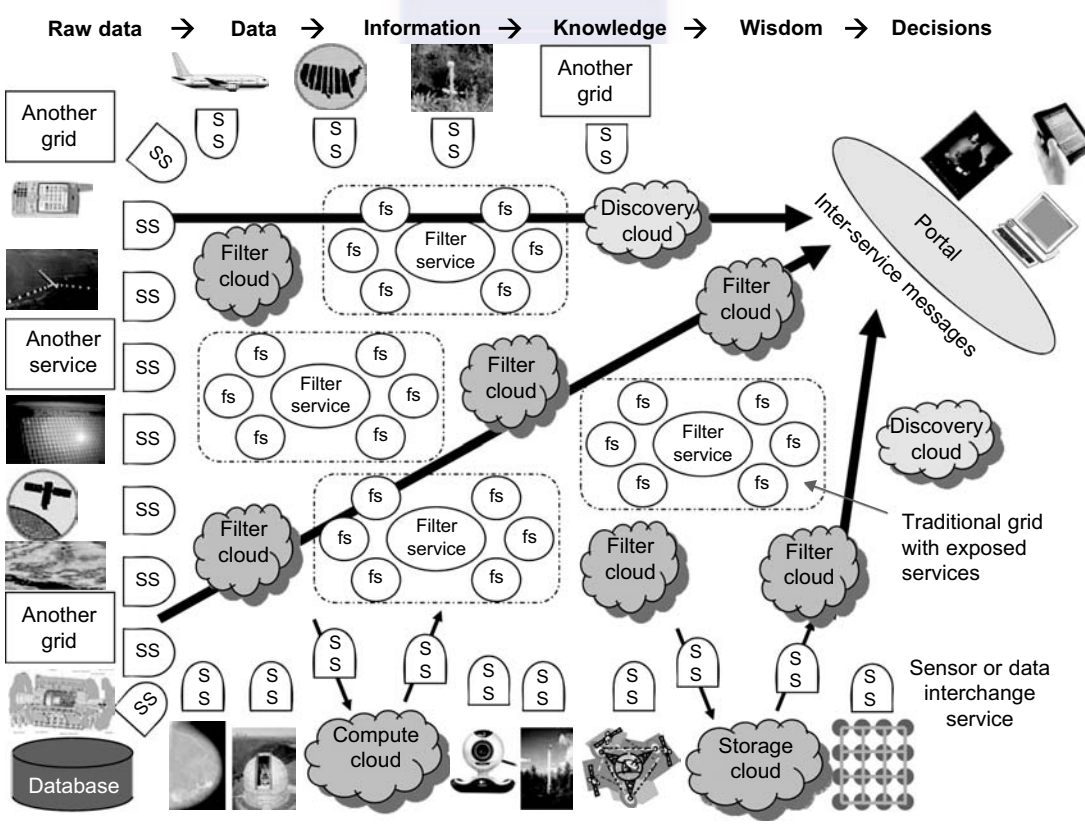


FIGURE 1.21

The evolution of SOA: grids of clouds and grids, where "SS" refers to a sensor service and "fs" to a filter or transforming service.

amounts of raw data items. Processing this data will generate useful information, and subsequently, the knowledge for our daily use. In fact, wisdom or intelligence is sorted out of large knowledge bases. Finally, we make intelligent decisions based on both biological and machine wisdom. Readers will see these structures more clearly in subsequent chapters.

Most distributed systems require a web interface or portal. For raw data collected by a large number of sensors to be transformed into useful information or knowledge, the data stream may go through a sequence of compute, storage, filter, and discovery clouds. Finally, the inter-service messages converge at the portal, which is accessed by all users. Two example portals, OGFCE and HUBzero, are described in Section 5.3 using both web service (portlet) and Web 2.0 (gadget) technologies. Many distributed programming models are also built on top of these basic constructs.

1.4.1.4 Grids versus Clouds

The boundary between grids and clouds are getting blurred in recent years. For web services, workflow technologies are used to coordinate or orchestrate services with certain specifications used to define critical business process models such as two-phase transactions. Section 5.2 discusses the general approach used in workflow, the BPEL Web Service standard, and several important workflow approaches including Pegasus, Taverna, Kepler, Trident, and Swift. In all approaches, one is building a collection of services which together tackle all or part of a distributed computing problem.

In general, a grid system applies static resources, while a cloud emphasizes elastic resources. For some researchers, the differences between grids and clouds are limited only in dynamic resource allocation based on virtualization and autonomic computing. One can build a grid out of multiple clouds. This type of grid can do a better job than a pure cloud, because it can explicitly support negotiated resource allocation. Thus one may end up building with a *system of systems*: such as a *cloud of clouds, a grid of clouds*, or a *cloud of grids,* or *inter-clouds* as a basic SOA architecture.

1.4.2 Trends toward Distributed Operating Systems

The computers in most distributed systems are loosely coupled. Thus, a distributed system inherently has multiple system images. This is mainly due to the fact that all node machines run with an independent operating system. To promote resource sharing and fast communication among node machines, it is best to have a *distributed* OS that manages all resources coherently and efficiently. Such a system is most likely to be a closed system, and it will likely rely on message passing and RPCs for internode communications. It should be pointed out that a distributed OS is crucial for upgrading the performance, efficiency, and flexibility of distributed applications.

1.4.2.1 Distributed Operating Systems

Tanenbaum [26] identifies three approaches for distributing resource management functions in a distributed computer system. The first approach is to build a *network OS* over a large number of heterogeneous OS platforms. Such an OS offers the lowest transparency to users, and is essentially a distributed file system, with independent computers relying on file sharing as a means of communication. The second approach is to develop middleware to offer a limited degree of resource sharing, similar to the MOSIX/OS developed for clustered systems (see Section 2.4.4). The third approach is to develop a truly *distributed OS* to achieve higher use or system transparency. Table 1.6 compares the functionalities of these three distributed operating systems.

Table 1.6 Feature Comparison of Three Distributed Operating Systems

Distributed OS Functionality	AMOEBA Developed at Vrije University [46]	DCE as OSF/1 by Open Software Foundation [7]	MOSIX for Linux Clusters at Hebrew University [3]
History and Current System Status	Written in C and tested in the European community; version 5.2 released in 1995	Built as a user extension on top of UNIX, VMS, Windows, OS/2, etc.	Developed since 1977, now called MOSIX2 used in HPC Linux and GPU clusters
Distributed OS Architecture	Microkernel-based and location-transparent, uses many servers to handle files, directory, replication, run, boot, and TCP/IP services	Middleware OS providing a platform for running distributed applications; The system supports RPC, security, and threads	A distributed OS with resource discovery, process migration, runtime support, load balancing, flood control, configuration, etc.
OS Kernel, Middleware, and Virtualization Support	A special microkernel that handles low-level process, memory, I/O, and communication functions	DCE packages handle file,time, directory, security services, RPC, and authentication at middleware or user space	MOSIX2 runs with Linux 2.6; extensions for use in multiple clusters and clouds with provisioned VMs
Communication Mechanisms	Uses a network-layer FLIP protocol and RPC to implement point-to-point and group communication	RPC supports authenticated communication and other security services in user programs	Using PVM, MPI in collective communications, priority process control, and queuing services

1.4.2.2 Amoeba versus DCE

DCE is a middleware-based system for distributed computing environments. The Amoeba was academically developed at Free University in the Netherlands. The Open Software Foundation (OSF) has pushed the use of DCE for distributed computing. However, the Amoeba, DCE, and MOSIX2 are still research prototypes that are primarily used in academia. No successful commercial OS products followed these research systems.

We need new web-based operating systems to support virtualization of resources in distributed environments. This is still a wide-open area of research. To balance the resource management workload, the functionalities of such a distributed OS should be distributed to any available server. In this sense, the conventional OS runs only on a centralized platform. With the distribution of OS services, the distributed OS design should take a lightweight microkernel approach like the Amoeba [46], or should extend an existing OS like the DCE [7] by extending UNIX. The trend is to free users from most resource management duties.

1.4.2.3 MOSIX2 for Linux Clusters

MOSIX2 is a distributed OS [3], which runs with a virtualization layer in the Linux environment. This layer provides a partial *single-system image* to user applications. MOSIX2 supports both sequential and parallel applications, and discovers resources and migrates software processes among Linux nodes. MOSIX2 can manage a Linux cluster or a grid of multiple clusters. Flexible management

of a grid allows owners of clusters to share their computational resources among multiple cluster owners. A MOSIX-enabled grid can extend indefinitely as long as trust exists among the cluster owners. The MOSIX2 is being explored for managing resources in all sorts of clusters, including Linux clusters, GPU clusters, grids, and even clouds if VMs are used. We will study MOSIX and its applications in Section 2.4.4.

1.4.2.4 Transparency in Programming Environments

Figure 1.22 shows the concept of a transparent computing infrastructure for future computing platforms. The user data, applications, OS, and hardware are separated into four levels. Data is owned by users, independent of the applications. The OS provides clear interfaces, standard programming interfaces, or system calls to application programmers. In future cloud infrastructure, the hardware will be separated by standard interfaces from the OS. Thus, users will be able to choose from different OSes on top of the hardware devices they prefer to use. To separate user data from specific application programs, users can enable cloud applications as SaaS. Thus, users can switch among different services. The data will not be bound to specific applications.

1.4.3 Parallel and Distributed Programming Models

In this section, we will explore four programming models for distributed computing with expected scalable performance and application flexibility. Table 1.7 summarizes three of these models, along with some software tool sets developed in recent years. As we will discuss, MPI is the most popular programming model for message-passing systems. Google's MapReduce and BigTable are for

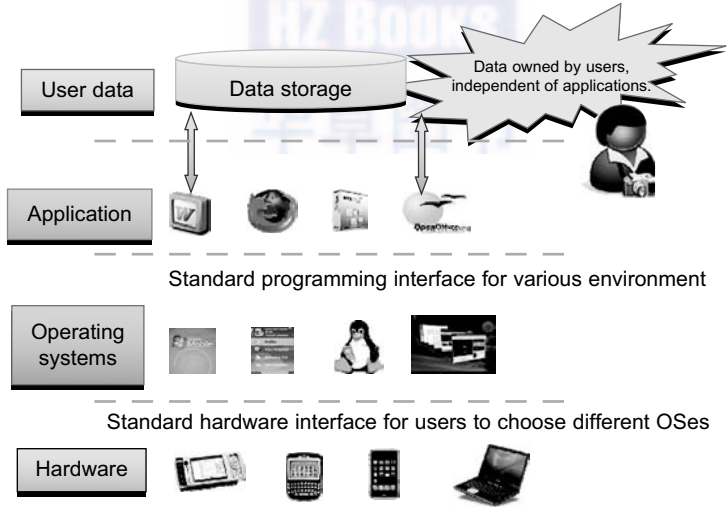


FIGURE 1.22

A transparent computing environment that separates the user data, application, OS, and hardware in time and space – an ideal model for cloud computing.

Table 1.7 Parallel and Distributed Programming Models and Tool Sets

Model	Description	Features
MPI	A library of subprograms that can be called from C or FORTRAN to write parallel programs running on distributed computer systems [6,28,42]	Specify synchronous or asynchronous point-to-point and collective communication commands and I/O operations in user programs for message-passing execution
MapReduce	A web programming model for scalable data processing on large clusters over large data sets, or in web search operations [16]	*Map* function generates a set of intermediate key/value pairs; *Reduce* function merges all intermediate values with the same key
Hadoop	A software library to write and run large user applications on vast data sets in business applications (http://hadoop.apache.org/core)	A scalable, economical, efficient, and reliable tool for providing users with easy access of commercial clusters

effective use of resources from Internet clouds and data centers. Service clouds demand extending Hadoop, EC2, and S3 to facilitate distributed computing over distributed storage systems. Many other models have also been proposed or developed in the past. In Chapters 5 and 6, we will discuss parallel and distributed programming in more details.

1.4.3.1 Message-Passing Interface (MPI)

This is the primary programming standard used to develop parallel and concurrent programs to run on a distributed system. MPI is essentially a library of subprograms that can be called from C or FORTRAN to write parallel programs running on a distributed system. The idea is to embody clusters, grid systems, and P2P systems with upgraded web services and utility computing applications. Besides MPI, distributed programming can be also supported with low-level primitives such as the *Parallel Virtual Machine* (PVM). Both MPI and PVM are described in Hwang and Xu [28].

1.4.3.2 MapReduce

This is a web programming model for scalable data processing on large clusters over large data sets [16]. The model is applied mainly in web-scale search and cloud computing applications. The user specifies a *Map* function to generate a set of intermediate key/value pairs. Then the user applies a *Reduce* function to merge all intermediate values with the same intermediate key. MapReduce is highly scalable to explore high degrees of parallelism at different job levels. A typical MapReduce computation process can handle terabytes of data on tens of thousands or more client machines. Hundreds of MapReduce programs can be executed simultaneously; in fact, thousands of MapReduce jobs are executed on Google's clusters every day.

1.4.3.3 Hadoop Library

Hadoop offers a software platform that was originally developed by a Yahoo! group. The package enables users to write and run applications over vast amounts of distributed data. Users can easily scale Hadoop to store and process petabytes of data in the web space. Also, Hadoop is economical in that it comes with an open source version of MapReduce that minimizes overhead

Table 1.8 Grid Standards and Toolkits for Scientific and Engineering Applications [6]

Standards	Service Functionalities	Key Features and Security Infrastructure
OGSA Standard	Open Grid Services Architecture; offers common grid service standards for general public use	Supports a heterogeneous distributed environment, bridging CAs, multiple trusted intermediaries, dynamic policies, multiple security mechanisms, etc.
Globus Toolkits	Resource allocation, Globus security infrastructure (GSI), and generic security service API	Sign-in multisite authentication with PKI, Kerberos, SSL, Proxy, delegation, and GSS API for message integrity and confidentiality
IBM Grid Toolbox	AIX and Linux grids built on top of Globus Toolkit, autonomic computing, replica services	Uses simple CA, grants access, grid service (ReGS), supports grid application for Java (GAF4J), GridMap in IntraGrid for security update

in task spawning and massive data communication. It is efficient, as it processes data with a high degree of parallelism across a large number of commodity nodes, and it is reliable in that it automatically keeps multiple data copies to facilitate redeployment of computing tasks upon unexpected system failures.

1.4.3.4 Open Grid Services Architecture (OGSA)

The development of grid infrastructure is driven by large-scale distributed computing applications. These applications must count on a high degree of resource and data sharing. Table 1.8 introduces OGSA as a common standard for general public use of grid services. Genesis II is a realization of OGSA. Key features include a distributed execution environment, *Public Key Infrastructure (PKI)* services using a local *certificate authority (CA)*, trust management, and security policies in grid computing.

1.4.3.5 Globus Toolkits and Extensions

Globus is a middleware library jointly developed by the U.S. Argonne National Laboratory and USC Information Science Institute over the past decade. This library implements some of the OGSA standards for resource discovery, allocation, and security enforcement in a grid environment. The Globus packages support multisite mutual authentication with PKI certificates. The current version of Globus, GT 4, has been in use since 2008. In addition, IBM has extended Globus for business applications. We will cover Globus and other grid computing middleware in more detail in Chapter 7.

1.5 PERFORMANCE, SECURITY, AND ENERGY EFFICIENCY

In this section, we will discuss the fundamental design principles along with rules of thumb for building massively distributed computing systems. Coverage includes scalability, availability, programming models, and security issues in clusters, grids, P2P networks, and Internet clouds.

1.5.1 Performance Metrics and Scalability Analysis

Performance metrics are needed to measure various distributed systems. In this section, we will discuss various dimensions of scalability and performance laws. Then we will examine system scalability against OS images and the limiting factors encountered.

1.5.1.1 Performance Metrics

We discussed *CPU speed* in MIPS and *network bandwidth* in Mbps in Section 1.3.1 to estimate processor and network performance. In a distributed system, performance is attributed to a large number of factors. *System throughput* is often measured in MIPS, *Tflops (tera floating-point operations per second)*, or *TPS (transactions per second)*. Other measures include *job response time* and *network latency*. An interconnection network that has low latency and high bandwidth is preferred. System overhead is often attributed to OS boot time, compile time, I/O data rate, and the runtime support system used. Other performance-related metrics include the QoS for Internet and web services; *system availability* and *dependability*; and *security resilience* for system defense against network attacks.

1.5.1.2 Dimensions of Scalability

Users want to have a distributed system that can achieve scalable performance. Any resource upgrade in a system should be backward compatible with existing hardware and software resources. Overdesign may not be cost-effective. System scaling can increase or decrease resources depending on many practical factors. The following dimensions of scalability are characterized in parallel and distributed systems:

- **Size scalability** This refers to achieving higher performance or more functionality by increasing the *machine size*. The word "size" refers to adding processors, cache, memory, storage, or I/O channels. The most obvious way to determine size scalability is to simply count the number of processors installed. Not all parallel computer or distributed architectures are equally size-scalable. For example, the IBM S2 was scaled up to 512 processors in 1997. But in 2008, the IBM BlueGene/L system scaled up to 65,000 processors.
- **Software scalability** This refers to upgrades in the OS or compilers, adding mathematical and engineering libraries, porting new application software, and installing more user-friendly programming environments. Some software upgrades may not work with large system configurations. Testing and fine-tuning of new software on larger systems is a nontrivial job.
- **Application scalability** This refers to matching *problem size* scalability with *machine size* scalability. Problem size affects the size of the data set or the workload increase. Instead of increasing machine size, users can enlarge the problem size to enhance system efficiency or cost-effectiveness.
- **Technology scalability** This refers to a system that can adapt to changes in building technologies, such as the component and networking technologies discussed in Section 3.1. When scaling a system design with new technology one must consider three aspects: *time*, *space*, and *heterogeneity*. (1) Time refers to generation scalability. When changing to new-generation processors, one must consider the impact to the motherboard, power supply, packaging and cooling, and so forth. Based on past experience, most systems upgrade their commodity processors every three to five years. (2) Space is related to packaging and energy concerns. Technology scalability demands harmony and portability among suppliers. (3) Heterogeneity refers to the use of hardware components or software packages from different vendors. Heterogeneity may limit the scalability.

1.5.1.3 Scalability versus OS Image Count

In Figure 1.23, *scalable performance* is estimated against the *multiplicity of OS images* in distributed systems deployed up to 2010. Scalable performance implies that the system can achieve higher speed by adding more processors or servers, enlarging the physical node's memory size, extending the disk capacity, or adding more I/O channels. The OS image is counted by the number of independent OS images observed in a cluster, grid, P2P network, or the cloud. SMP and NUMA are included in the comparison. An *SMP (symmetric multiprocessor)* server has a single system image, which could be a single node in a large cluster. By 2010 standards, the largest shared-memory SMP node was limited to a few hundred processors. The scalability of SMP systems is constrained primarily by packaging and the system interconnect used.

NUMA (nonuniform memory access) machines are often made out of SMP nodes with distributed, shared memory. A NUMA machine can run with multiple operating systems, and can scale to a few thousand processors communicating with the MPI library. For example, a NUMA machine may have 2,048 processors running 32 SMP operating systems, resulting in 32 OS images in the 2,048-processor NUMA system. The cluster nodes can be either SMP servers or high-end machines that are loosely coupled together. Therefore, clusters have much higher scalability than NUMA machines. The number of OS images in a cluster is based on the cluster nodes concurrently in use. The cloud could be a virtualized cluster. As of 2010, the largest cloud was able to scale up to a few thousand VMs.

Keeping in mind that many cluster nodes are SMP or multicore servers, the total number of processors or cores in a cluster system is one or two orders of magnitude greater than the number of OS images running in the cluster. The grid node could be a server cluster, or a mainframe, or a supercomputer, or an MPP. Therefore, the number of OS images in a large grid structure could be hundreds or thousands fewer than the total number of processors in the grid. A P2P network can easily scale to millions of independent peer nodes, essentially desktop machines. P2P performance

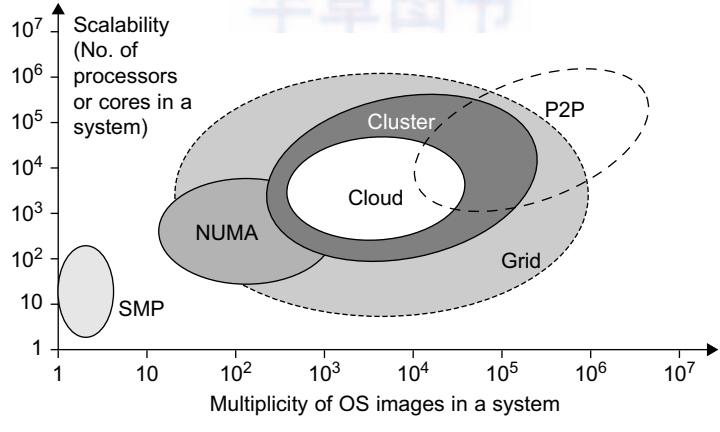


FIGURE 1.23

System scalability versus multiplicity of OS images based on 2010 technology.

depends on the QoS in a public network. Low-speed P2P networks, Internet clouds, and computer clusters should be evaluated at the same networking level.

1.5.1.4 Amdahl's Law

Consider the execution of a given program on a uniprocessor workstation with a total execution time of T minutes. Now, let's say the program has been parallelized or partitioned for parallel execution on a cluster of many processing nodes. Assume that a fraction α of the code must be executed sequentially, called the *sequential bottleneck*. Therefore, $(1-\alpha)$ of the code can be compiled for parallel execution by n processors. The total execution time of the program is calculated by $\alpha T + (1-\alpha)T/n$, where the first term is the sequential execution time on a single processor and the second term is the parallel execution time on n processing nodes.

All system or communication overhead is ignored here. The I/O time or exception handling time is also not included in the following speedup analysis. Amdahl's Law states that the *speedup factor* of using the n-processor system over the use of a single processor is expressed by:

$$\text{Speedup} = S = T/[\alpha T + (1-\alpha)T/n] = 1/[\alpha + (1-\alpha)/n] \tag{1.1}$$

The maximum speedup of n is achieved only if the *sequential bottleneck* α is reduced to zero or the code is fully parallelizable with $\alpha = 0$. As the cluster becomes sufficiently large, that is, $n \to \infty$, S approaches $1/\alpha$, an upper bound on the speedup S. Surprisingly, this upper bound is independent of the cluster size n. The sequential bottleneck is the portion of the code that cannot be parallelized. For example, the maximum speedup achieved is 4, if $\alpha = 0.25$ or $1-\alpha = 0.75$, even if one uses hundreds of processors. Amdahl's law teaches us that we should make the sequential bottleneck as small as possible. Increasing the cluster size alone may not result in a good speedup in this case.

1.5.1.5 Problem with Fixed Workload

In Amdahl's law, we have assumed the same amount of workload for both sequential and parallel execution of the program with a fixed problem size or data set. This was called *fixed-workload speedup* by Hwang and Xu [14]. To execute a fixed workload on n processors, parallel processing may lead to a s*ystem efficiency* defined as follows:

$$E = S/n = 1/[\alpha n + 1 - \alpha] \tag{1.2}$$

Very often the system efficiency is rather low, especially when the cluster size is very large. To execute the aforementioned program on a cluster with $n = 256$ nodes, extremely low efficiency $E = 1/[0.25 \times 256 + 0.75] = 1.5\%$ is observed. This is because only a few processors (say, 4) are kept busy, while the majority of the nodes are left idling.

1.5.1.6 Gustafson's Law

To achieve higher efficiency when using a large cluster, we must consider scaling the problem size to match the cluster capability. This leads to the following speedup law proposed by John Gustafson (1988), referred as *scaled-workload speedup* in [14]. Let W be the workload in a given program. When using an n-processor system, the user scales the workload to $W' = \alpha W + (1-\alpha)nW$. Note that only the parallelizable portion of the workload is scaled n times in the second term. This scaled

workload W' is essentially the sequential execution time on a single processor. The parallel execution time of a scaled workload W' on n processors is defined by a *scaled-workload speedup* as follows:

$$S' = W'/W = [\alpha W + (1-\alpha)nW]/W = \alpha + (1-\alpha)n \tag{1.3}$$

This speedup is known as Gustafson's law. By fixing the parallel execution time at level W, the following efficiency expression is obtained:

$$E' = S'/n = \alpha/n + (1-\alpha) \tag{1.4}$$

For the preceding program with a scaled workload, we can improve the efficiency of using a 256-node cluster to $E' = 0.25/256 + 0.75 = 0.751$. One should apply Amdahl's law and Gustafson's law under different workload conditions. For a fixed workload, users should apply Amdahl's law. To solve scaled problems, users should apply Gustafson's law.

1.5.2 Fault Tolerance and System Availability

In addition to performance, system availability and application flexibility are two other important design goals in a distributed computing system.

1.5.2.1 System Availability

HA (high availability) is desired in all clusters, grids, P2P networks, and cloud systems. A system is highly available if it has a long *mean time to failure (MTTF)* and a short *mean time to repair (MTTR)*. *System availability* is formally defined as follows:

$$System\ Availability = MTTF/(MTTF + MTTR) \tag{1.5}$$

System availability is attributed to many factors. All hardware, software, and network components may fail. Any failure that will pull down the operation of the entire system is called a *single point of failure*. The rule of thumb is to design a dependable computing system with no single point of failure. Adding hardware redundancy, increasing component reliability, and designing for testability will help to enhance system availability and dependability. In Figure 1.24, the effects on system availability are estimated by scaling the system size in terms of the number of processor cores in the system.

In general, as a distributed system increases in size, availability decreases due to a higher chance of failure and a difficulty in isolating the failures. Both SMP and MPP are very vulnerable with centralized resources under one OS. NUMA machines have improved in availability due to the use of multiple OSes. Most clusters are designed to have HA with failover capability. Meanwhile, private clouds are created out of virtualized data centers; hence, a cloud has an estimated availability similar to that of the hosting cluster. A grid is visualized as a hierarchical cluster of clusters. Grids have higher availability due to the isolation of faults. Therefore, clusters, clouds, and grids have decreasing availability as the system increases in size. A P2P file-sharing network has the highest aggregation of client machines. However, it operates independently with low availability, and even many peer nodes depart or fail simultaneously.

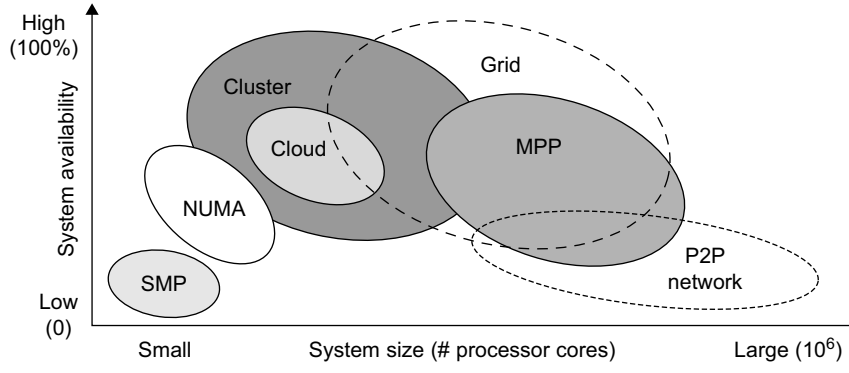


FIGURE 1.24

Estimated system availability by system size of common configurations in 2010.

1.5.3 Network Threats and Data Integrity

Clusters, grids, P2P networks, and clouds demand security and copyright protection if they are to be accepted in today's digital society. This section introduces system vulnerability, network threats, defense countermeasures, and copyright protection in distributed or cloud computing systems.

1.5.3.1 Threats to Systems and Networks

Network viruses have threatened many users in widespread attacks. These incidents have created a worm epidemic by pulling down many routers and servers, and are responsible for the loss of billions of dollars in business, government, and services. Figure 1.25 summarizes various attack types and their potential damage to users. As the figure shows, information leaks lead to a loss of confidentiality. Loss of data integrity may be caused by user alteration, Trojan horses, and service spoofing attacks. A *denial of service (DoS)* results in a loss of system operation and Internet connections.

Lack of authentication or authorization leads to attackers' illegitimate use of computing resources. Open resources such as data centers, P2P networks, and grid and cloud infrastructures could become the next targets. Users need to protect clusters, grids, clouds, and P2P systems. Otherwise, users should not use or trust them for outsourced work. Malicious intrusions to these systems may destroy valuable hosts, as well as network and storage resources. Internet anomalies found in routers, gateways, and distributed hosts may hinder the acceptance of these public-resource computing services.

1.5.3.2 Security Responsibilities

Three security requirements are often considered: *confidentiality*, *integrity*, and *availability* for most Internet service providers and cloud users. In the order of SaaS, PaaS, and IaaS, the providers gradually release the responsibility of security control to the cloud users. In summary, the SaaS model relies on the cloud provider to perform all security functions. At the other extreme, the IaaS model wants the users to assume almost all security functions, but to leave availability in the hands of the providers. The PaaS model relies on the provider to maintain data integrity and availability, but burdens the user with confidentiality and privacy control.

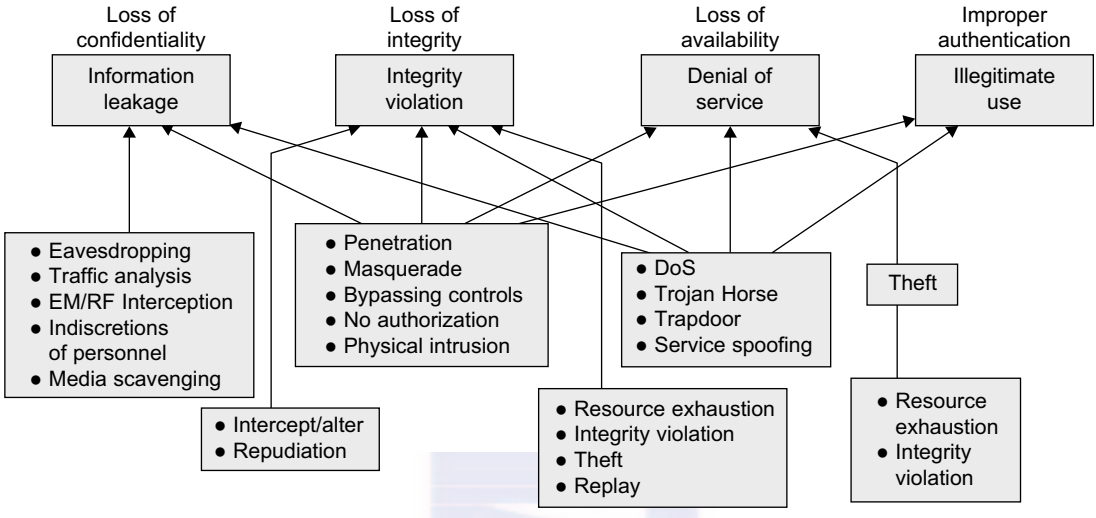


FIGURE 1.25

Various system attacks and network threats to the cyberspace, resulting 4 types of losses.

1.5.3.3 Copyright Protection

Collusive piracy is the main source of intellectual property violations within the boundary of a P2P network. Paid clients (colluders) may illegally share copyrighted content files with unpaid clients (pirates). Online piracy has hindered the use of open P2P networks for commercial content delivery. One can develop a proactive content poisoning scheme to stop colluders and pirates from alleged copyright infringements in P2P file sharing. Pirates are detected in a timely manner with identity-based signatures and timestamped tokens. This scheme stops collusive piracy from occurring without hurting legitimate P2P clients. Chapters 4 and 7 cover grid and cloud security, P2P reputation systems, and copyright protection.

1.5.3.4 System Defense Technologies

Three generations of network defense technologies have appeared in the past. In the first generation, tools were designed to prevent or avoid intrusions. These tools usually manifested themselves as access control policies or tokens, cryptographic systems, and so forth. However, an intruder could always penetrate a secure system because there is always a weak link in the security provisioning process. The second generation detected intrusions in a timely manner to exercise remedial actions. These techniques included firewalls, intrusion detection systems (IDSes), PKI services, reputation systems, and so on. The third generation provides more intelligent responses to intrusions.

1.5.3.5 Data Protection Infrastructure

Security infrastructure is required to safeguard web and cloud services. At the user level, one needs to perform trust negotiation and reputation aggregation over all users. At the application end, we need to establish security precautions in worm containment and intrusion detection

against virus, worm, and distributed DoS (DDoS) attacks. We also need to deploy mechanisms to prevent online piracy and copyright violations of digital content. In Chapter 4, we will study reputation systems for protecting cloud systems and data centers. Security responsibilities are divided between cloud providers and users differently for the three cloud service models. The providers are totally responsible for platform availability. The IaaS users are more responsible for the confidentiality issue. The IaaS providers are more responsible for data integrity. In PaaS and SaaS services, providers and users are equally responsible for preserving data integrity and confidentiality.

1.5.4 Energy Efficiency in Distributed Computing

Primary performance goals in conventional parallel and distributed computing systems are high performance and high throughput, considering some form of performance reliability (e.g., fault tolerance and security). However, these systems recently encountered new challenging issues including energy efficiency, and workload and resource outsourcing. These emerging issues are crucial not only on their own, but also for the sustainability of large-scale computing systems in general. This section reviews energy consumption issues in servers and HPC systems, an area known as *distributed power management (DPM)*.

Protection of data centers demands integrated solutions. Energy consumption in parallel and distributed computing systems raises various monetary, environmental, and system performance issues. For example, Earth Simulator and Petaflop are two systems with 12 and 100 megawatts of peak power, respectively. With an approximate price of $100 per megawatt, their energy costs during peak operation times are $1,200 and $10,000 per hour; this is beyond the acceptable budget of many (potential) system operators. In addition to power cost, cooling is another issue that must be addressed due to negative effects of high temperature on electronic components. The rising temperature of a circuit not only derails the circuit from its normal range, but also decreases the lifetime of its components.

1.5.4.1 Energy Consumption of Unused Servers

To run a server farm (data center) a company has to spend a huge amount of money for hardware, software, operational support, and energy every year. Therefore, companies should thoroughly identify whether their installed server farm (more specifically, the volume of provisioned resources) is at an appropriate level, particularly in terms of utilization. It was estimated in the past that, on average, one-sixth (15 percent) of the full-time servers in a company are left powered on without being actively used (i.e., they are idling) on a daily basis. This indicates that with 44 million servers in the world, around 4.7 million servers are not doing any useful work.

The potential savings in turning off these servers are large—$3.8 billion globally in energy costs alone, and $24.7 billion in the total cost of running nonproductive servers, according to a study by 1E Company in partnership with the Alliance to Save Energy (ASE). This amount of wasted energy is equal to 11.8 million tons of carbon dioxide per year, which is equivalent to the CO pollution of 2.1 million cars. In the United States, this equals 3.17 million tons of carbon dioxide, or 580,678 cars. Therefore, the first step in IT departments is to analyze their servers to find unused and/or underutilized servers.

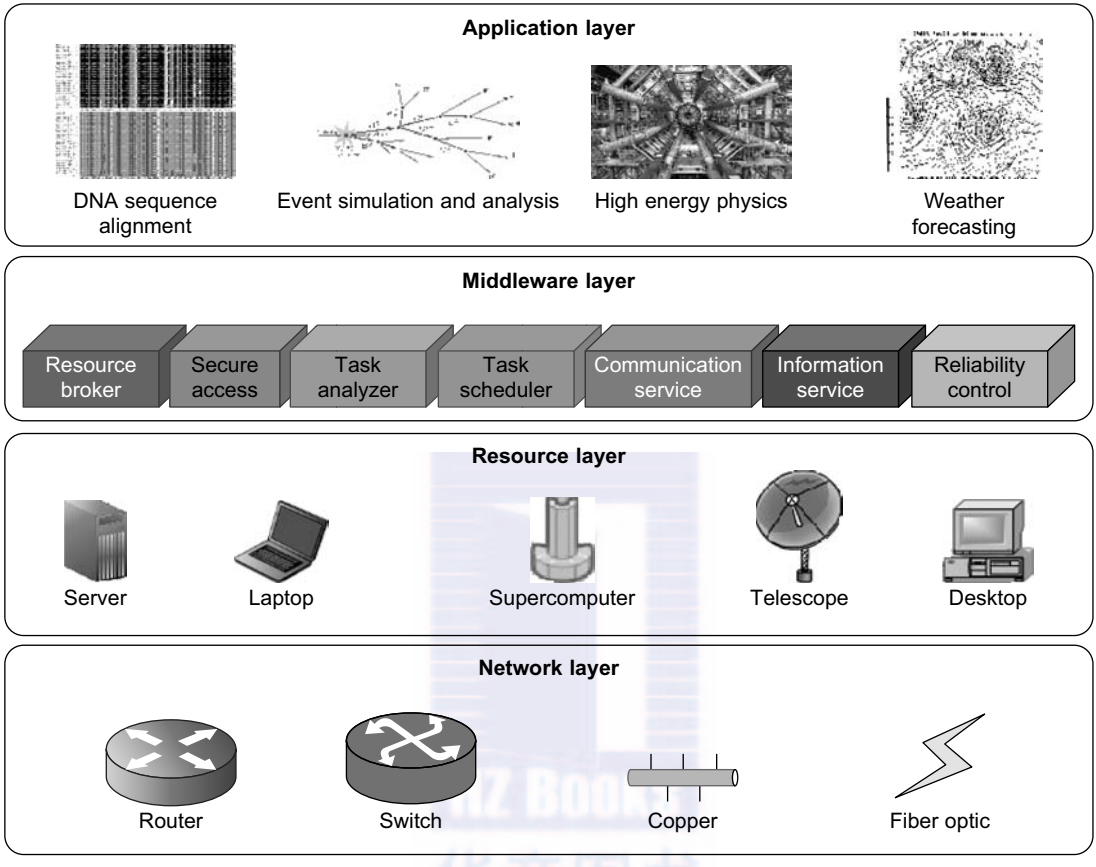


FIGURE 1.26

Four operational layers of distributed computing systems.

(*Courtesy of Zomaya, Rivandi and Lee of the University of Sydney [33]*)

1.5.4.2 Reducing Energy in Active Servers

In addition to identifying unused/underutilized servers for energy savings, it is also necessary to apply appropriate techniques to decrease energy consumption in active distributed systems with negligible influence on their performance. Power management issues in distributed computing platforms can be categorized into four layers (see Figure 1.26): the application layer, middleware layer, resource layer, and network layer.

1.5.4.3 Application Layer

Until now, most user applications in science, business, engineering, and financial areas tend to increase a system's speed or quality. By introducing energy-aware applications, the challenge is to design sophisticated multilevel and multi-domain energy management applications without hurting

performance. The first step toward this end is to explore a relationship between performance and energy consumption. Indeed, an application's energy consumption depends strongly on the number of instructions needed to execute the application and the number of transactions with the storage unit (or memory). These two factors (compute and storage) are correlated and they affect completion time.

1.5.4.4 Middleware Layer

The middleware layer acts as a bridge between the application layer and the resource layer. This layer provides resource broker, communication service, task analyzer, task scheduler, security access, reliability control, and information service capabilities. It is also responsible for applying energy-efficient techniques, particularly in task scheduling. Until recently, scheduling was aimed at minimizing *makespan*, that is, the execution time of a set of tasks. Distributed computing systems necessitate a new cost function covering both makespan and energy consumption.

1.5.4.5 Resource Layer

The resource layer consists of a wide range of resources including computing nodes and storage units. This layer generally interacts with hardware devices and the operating system; therefore, it is responsible for controlling all distributed resources in distributed computing systems. In the recent past, several mechanisms have been developed for more efficient power management of hardware and operating systems. The majority of them are hardware approaches particularly for processors.

Dynamic power management (DPM) and *dynamic voltage-frequency scaling (DVFS)* are two popular methods incorporated into recent computer hardware systems [21]. In DPM, hardware devices, such as the CPU, have the capability to switch from idle mode to one or more lower-power modes. In DVFS, energy savings are achieved based on the fact that the power consumption in CMOS circuits has a direct relationship with frequency and the square of the voltage supply. Execution time and power consumption are controllable by switching among different frequencies and voltages [31].

1.5.4.6 Network Layer

Routing and transferring packets and enabling network services to the resource layer are the main responsibility of the network layer in distributed computing systems. The major challenge to build energy-efficient networks is, again, determining how to measure, predict, and create a balance between energy consumption and performance. Two major challenges to designing energy-efficient networks are:

- The models should represent the networks comprehensively as they should give a full understanding of interactions among time, space, and energy.
- New, energy-efficient routing algorithms need to be developed. New, energy-efficient protocols should be developed against network attacks.

As information resources drive economic and social development, data centers become increasingly important in terms of where the information items are stored and processed, and where services are provided. Data centers become another core infrastructure, just like the power grid and

transportation systems. Traditional data centers suffer from high construction and operational costs, complex resource management, poor usability, low security and reliability, and huge energy consumption. It is necessary to adopt new technologies in next-generation data-center designs, a topic we will discuss in more detail in Chapter 4.

1.5.4.7 DVFS Method for Energy Efficiency

The DVFS method enables the exploitation of the slack time (idle time) typically incurred by inter-task relationship. Specifically, the slack time associated with a task is utilized to execute the task in a lower voltage frequency. The relationship between energy and voltage frequency in CMOS circuits is related by:

$$\begin{cases} E = C_{eff} f v^2 t \\ f = K \dfrac{(v - v_t)^2}{v} \end{cases} \tag{1.6}$$

where v, C_{eff}, K, and v_t are the voltage, circuit switching capacity, a technology dependent factor, and threshold voltage, respectively, and the parameter t is the execution time of the task under clock frequency f. By reducing voltage and frequency, the device's energy consumption can also be reduced.

Example 1.2 Energy Efficiency in Distributed Power Management

Figure 1.27 illustrates the DVFS method. This technique as shown on the right saves the energy compared to traditional practices shown on the left. The idea is to reduce the frequency and/or voltage during workload slack time. The transition latencies between lower-power modes are very small. Thus energy is saved by switching between operational modes. Switching between low-power modes affects performance. Storage units must interact with the computing nodes to balance power consumption. According to Ge, Feng, and Cameron [21], the storage devices are responsible for about 27 percent of the total energy consumption in a data center. This figure increases rapidly due to a 60 percent increase in storage needs annually, making the situation even worse.

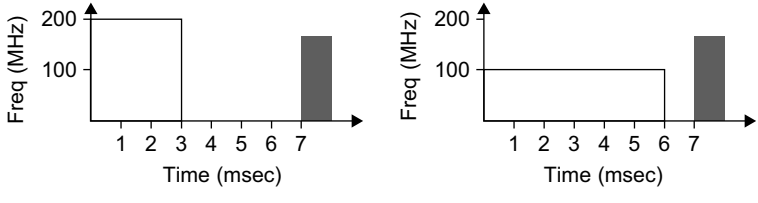


FIGURE 1.27

The DVFS technique (right) saves energy, compared to traditional practices (left) by reducing the frequency or voltage during slack time.

1.6 BIBLIOGRAPHIC NOTES AND HOMEWORK PROBLEMS

Over the past four decades, parallel processing and distributed computing have been hot topics for research and development. Earlier work in parallel computing can be found in several classic books [14,17,27,28]. Recent coverage of distributed systems can be found in [8,13,20,22]. Cluster computing was covered in [2,10,28,38] and grid computing in [6,18,42,47,51]. P2P networks were introduced in [6,34,46]. Multicore CPUs and many-core GPU processors were discussed in [15,32,36]. Information on the Top 500 supercomputers can be found in [50].

Data centers are introduced in [4,19,26], and recent computer architecture in [24,26]. Cloud computing is studied in [1,9,11,18,29,30,39,44]. The edited volume [11] on cloud computing by Buyya, Broberg, and Goscinski serves as a good resource on cloud computing research. Chou's book [12] emphasizes business models for cloud services. Virtualization techniques were introduced in [40–44]. The articles by Bell, Gray, and Szalay [5]; Foster, et al. [18]; and Hey [25] address critical issues concerning data-intensive grid and cloud computing. Massive data parallelism and programming are covered in [14,32].

Distributed algorithms and MPI programming are studied in [3,12,15,22,28]. Distributed operating systems and software tools are covered in [3,7,13,46]. Energy efficiency and power management are studied in [21,31,52]. The Internet of Things is studied in [45,48]. The work of Hwang and Li [30] suggested ways to cope with cloud security and data protection problems. In subsequent chapters, additional references will be provided. The following list highlights some international conferences, magazines, and journals that often report on the latest developments in parallelism, clusters, grids, P2P systems, and cloud and distributed systems:

- **IEEE and Related Conference Publications** Internet Computing, TPDS (Transactions on Parallel and Distributed Systems), TC (Transactions on Computers), TON (Transactions on Networking), ICDCS (International Conference on Distributed Computing Systems), IPDPS (International Parallel and Distributed Processing Symposium), INFOCOM, GLOBECOM, CCGrid (Clusters, Clouds and The Grid), P2P Computing, HPDC (High-Performance Distributed Computing), CloudCom (International Conference on Cloud Computing Technology and Science), ISCA (International Symposium on Computer Architecture), HPCA (High-Performance Computer Architecture), Computer Magazine, TDSC (Transactions on Dependable and Secure Computing), TKDE (Transactions on Knowledge and Data Engineering), HPCC (High Performance Computing and Communications), ICPADS (International Conference on Parallel and Distributed Applications and Systems), NAS (Networks, Architectures, and Storage), and GPC (Grid and Pervasive Computing)
- **ACM, Internet Society, IFIP, and Other Relevant Publications** Supercomputing Conference, ACM Transactions on Computing Systems, USENIX Technical Conference, JPDC (Journal of Parallel and Distributed Computing), Journal of Cloud Computing, Journal of Distributed Computing, Journal of Cluster Computing, Future Generation Computer Systems, Journal of Grid Computing, Journal of Parallel Computing, International Conference on Parallel Processing (ICPP), European Parallel Computing Conference (EuroPAR), Concurrency: Practice and Experiences (Wiley), NPC (IFIP Network and Parallel Computing), and PDCS (ISCA Parallel and Distributed Computer Systems)

Acknowledgments

This chapter was authored by Kai Hwang primarily. Geoffrey Fox and Albert Zomaya have contributed to Sections 1.4.1 and 1.5.4, respectively. Xiaosong Lou and Lizhong Chen at the University of Southern California have assisted in plotting Figures 1.4 and 1.10. Nikzad Rivandi and Young-Choon Lee of the University of Sydney have contributed partially to Section 1.5.4. Jack Dongarra has edited the entire Chapter 1.

References

[1] Amazon EC2 and S3, Elastic Compute Cloud (EC2) and Simple Scalable Storage (S3). http://en.wikipedia.org/wiki/Amazon_Elastic_Compute_Cloud and http://spatten_presentations.s3.amazonaws.com/s3-on-rails.pdf.

[2] M. Baker, R. Buyya, Cluster computing at a glance, in: R. Buyya (Ed.), High-Performance Cluster Computing, Architecture and Systems, vol. 1, Prentice-Hall, Upper Saddle River, NJ, 1999, pp. 3–47, Chapter 1.

[3] A. Barak, A. Shiloh, The MOSIX Management System for Linux Clusters, Multi-Clusters, CPU Clusters, and Clouds, White paper. www.MOSIX.org//txt_pub.html, 2010.

[4] L. Barroso, U. Holzle, The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Morgan & Claypool Publishers, 2009.

[5] G. Bell, J. Gray, A. Szalay, Petascale computational systems: balanced cyberstructure in a data-centric World, IEEE Comput. Mag. (2006).

[6] F. Berman, G. Fox, T. Hey (Eds.), Grid Computing, Wiley, 2003.

[7] M. Bever, et al., Distributed systems, OSF DCE, and beyond, in: A. Schill (Ed.), DCE-The OSF Distributed Computing Environment, Springer-Verlag, 1993, pp. 1–20.

[8] K. Birman, Reliable Distributed Systems: Technologies, Web Services, and Applications, Springer-Verlag, 2005.

[9] G. Boss, et al., Cloud Computing–The BlueCloud Project. www.ibm.com/developerworks/websphere/zones/hipods/, October 2007.

[10] R. Buyya (Ed.), High-Performance Cluster Computing, Vol. 1 and 2, Prentice-Hall, Englewood Cliffs, NJ, 1999.

[11] R. Buyya, J. Broberg, A. Goscinski (Eds.), Cloud Computing: Principles and Paradigms, Wiley, 2011.

[12] T. Chou, Introduction to Cloud Computing: Business and Technology. Lecture Notes at Stanford University and Tsinghua University, Active Book Press, 2010.

[13] G. Coulouris, J. Dollimore, T. Kindberg, Distributed Systems: Concepts and Design, Wesley, 2005.

[14] D. Culler, J. Singh, A. Gupta, Parallel Computer Architecture, Kaufmann Publishers, 1999.

[15] B. Dally, GPU Computing to Exascale and Beyond, Keynote Address at ACM Supercomputing Conference, November 2010.

[16] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, in: Proceedings of OSDI 2004. Also, Communication of ACM, Vol. 51, 2008, pp. 107–113.

[17] J. Dongarra, et al. (Eds.), Source Book of Parallel Computing, Morgan Kaufmann, San Francisco, 2003.

[18] I. Foster, Y. Zhao, J. Raicu, S. Lu, Cloud Computing and Grid Computing 360-Degree Compared, Grid Computing Environments Workshop, 12–16 November 2008.

[19] D. Gannon, The Client+Cloud: Changing the Paradigm for Scientific Research, Keynote Address, IEEE CloudCom2010, Indianapolis, 2 November 2010.

[20] V.K. Garg, Elements of Distributed Computing. Wiley-IEEE Press, 2002.

[21] R. Ge, X. Feng, K. Cameron, Performance Constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters, in: Proceedings Supercomputing Conf., Washington, DC, 2005.

[22] S. Ghosh, Distributed Systems–An Algorithmic Approach, Chapman & Hall/CRC, 2007.
[23] B. He, W. Fang, Q. Luo, N. Govindaraju, T. Wang, Mars: A MapReduce Framework on Graphics Processor, ACM PACT'08, Toronto, Canada, 25–29 October 2008.
[24] J. Hennessy, D. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, 2007.
[25] T. Hey, et al., The Fourth Paradigm: Data-Intensive Scientific Discovery, Microsoft Research, 2009.
[26] M.D. Hill, et al., The Data Center as a Computer, Morgan & Claypool Publishers, 2009.
[27] K. Hwang, Advanced Computer Architecture: Parallelism, Scalability, Programming, McGraw-Hill, 1993.
[28] K. Hwang, Z. Xu, Scalable Parallel Computing, McGraw-Hill, 1998.
[29] K. Hwang, S. Kulkarni, Y. Hu, Cloud Security with Virtualized Defense and Reputation-based Trust Management, in: IEEE Conference on Dependable, Autonomous, and Secure Computing (DAC-2009), Chengdu, China, 14 December 2009.
[30] K. Hwang, D. Li, Trusted Cloud Computing with Secure Resources and Data Coloring, in: IEEE Internet Computing, Special Issue on Trust and Reputation Management, September 2010, pp. 14–22.
[31] Kelton Research, Server Energy & Efficiency Report. www.1e.com/EnergyCampaign/downloads/Server_Energy_and_Efficiency_Report_2009.pdf, September 2009.
[32] D. Kirk, W. Hwu, Programming Massively Processors: A Hands-on Approach, Morgan Kaufmann, 2010.
[33] Y.C. Lee, A.Y. Zomaya, A Novel State Transition Method for Metaheuristic-Based Scheduling in Heterogeneous Computing Systems, in: IEEE Transactions on Parallel and Distributed Systems, September 2008.
[34] Z.Y. Li, G. Xie, K. Hwang, Z.C. Li, Churn-Resilient Protocol for Massive Data Dissemination in P2P Networks, in: IEEE Trans. Parallel and Distributed Systems, May 2011.
[35] X. Lou, K. Hwang, Collusive Piracy Prevention in P2P Content Delivery Networks, in: IEEE Trans. on Computers, July, 2009, pp. 970–983.
[36] NVIDIA Corp. Fermi: NVIDIA's Next-Generation CUDA Compute Architecture, White paper, 2009.
[37] D. Peleg, Distributed Computing: A Locality-Sensitive Approach, SIAM, 2000.
[38] G.F. Pfister, In Search of Clusters, Second ed., Prentice-Hall, 2001.
[39] J. Qiu, T. Gunarathne, J. Ekanayake, J. Choi, S. Bae, H. Li, et al., Hybrid Cloud and Cluster Computing Paradigms for Life Science Applications, in: 11th Annual Bioinformatics Open Source Conference BOSC 2010, 9–10 July 2010.
[40] M. Rosenblum, T. Garfinkel, Virtual machine monitors: current technology and future trends, IEEE Computer (May) (2005) 39–47.
[41] M. Rosenblum, Recent Advances in Virtual Machines and Operating Systems, Keynote Address, ACM ASPLOS 2006.
[42] J. Smith, R. Nair, Virtual Machines, Morgan Kaufmann, 2005.
[43] B. Sotomayor, R. Montero, I. Foster, Virtual Infrastructure Management in Private and Hybrid Clouds, IEEE Internet Computing, September 2009.
[44] SRI. The Internet of Things, in: Disruptive Technologies: Global Trends 2025, www.dni.gov/nic/PDF_GIF_Confreports/disruptivetech/appendix_F.pdf, 2010.
[45] A. Tanenbaum, Distributed Operating Systems, Prentice-Hall, 1995.
[46] I. Taylor, From P2P to Web Services and Grids, Springer-Verlag, London, 2005.
[47] Twister, Open Source Software for Iterative MapReduce, http://www.iterativemapreduce.org/.
[48] Wikipedia. Internet of Things, http://en.wikipedia.org/wiki/Internet_of_Things, June 2010.
[49] Wikipedia. CUDA, http://en.wikipedia.org/wiki/CUDA, March 2011.
[50] Wikipedia. TOP500, http://en.wikipedia.org/wiki/TOP500, February 2011.
[51] Y. Wu, K. Hwang, Y. Yuan, W. Zheng, Adaptive Workload Prediction of Grid Performance in Confidence Windows, in: IEEE Trans. on Parallel and Distributed Systems, July 2010.
[52] Z. Zong, Energy-Efficient Resource Management for High-Performance Computing Platforms, Ph.D. dissertation, Auburn University, 9 August 2008.

HOMEWORK PROBLEMS

Problem 1.1

Briefly define the following basic techniques and technologies that represent recent related advances in computer architecture, parallel processing, distributed computing, Internet technology, and information services:

a. High-performance computing (HPC) system
b. High-throughput computing (HTC) system
c. Peer-to-peer (P2P) network
d. Computer cluster versus computational grid
e. Service-oriented architecture (SOA)
f. Pervasive computing versus Internet computing
g. Virtual machine versus virtual infrastructure
h. Public cloud versus private cloud
i. Radio-frequency identifier (RFID)
j. Global positioning system (GPS)
k. Sensor network
l. Internet of Things (IoT)
m. Cyber-physical system (CPS)

Problem 1.2

Circle only one correct answer in each of the following two questions:

1. In the 2009 Top 500 list of the fastest computer systems, which architecture dominates?
   a. Symmetric shared-memory multiprocessor systems.
   b. Centralized massively parallel processor (MPP) systems.
   c. Clusters of cooperative computers.
2. In a cloud formed by a cluster of servers, all servers must be selected as follows:
   a. All cloud machines must be built on physical servers.
   b. All cloud machines must be built with virtual servers.
   c. The cloud machines can be either physical or virtual servers.

Problem 1.3

An increasing number of organizations in industry and business sectors adopt cloud systems. Answer the following questions regarding cloud computing:

a. List and describe the main characteristics of cloud computing systems.
b. Discuss key enabling technologies in cloud computing systems.
c. Discuss different ways for cloud service providers to maximize their revenues.

Problem 1.4

Match 10 abbreviated terms and system models on the left with their descriptions on the right. Enter the description label (**a, b, c, …, j**) in the underlined blanks in front of the terms.

_____**Globus**	(**a**) A scalable software platform promoted by Apache for web users to write and run applications over vast amounts of distributed data
_____**BitTorrent**	(**b**) A P2P network for MP3 music delivery with a centralized directory server
_____**Gnutella**	(**c**) The programming model and associated implementation by Google for distributed mapping and reduction of very large data sets
_____**EC2**	(**d**) A middleware library jointly developed by USC/ISI and Argonne National Lab for grid resource management and job scheduling
_____**TeraGrid**	(**e**) A distributed storage program by Google for managing structured data that can scale to very large sizes
_____**EGEE**	(**f**) A P2P file-sharing network using multiple file index trackers
_____**Hadoop**	(**g**) A critical design goal of clusters of computers to tolerate nodal faults or recovery from host failures
_____**SETI@home**	(**h**) The service architecture specification as an open grid standard
_____**Napster**	(**i**) An elastic and flexible computing environment that allows web application developers to acquire cloud resources effectively
_____**BigTable**	(**j**) A P2P grid over 3 million desktops for distributed signal processing in search of extraterrestrial intelligence

Problem 1.5

Consider a multicore processor with four heterogeneous cores labeled A, B, C, and D. Assume cores A and D have the same speed. Core B runs twice as fast as core A, and core C runs three times faster than core A. Assume that all four cores start executing the following application at the same time and no cache misses are encountered in all core operations. Suppose an application needs to compute the square of each element of an array of 256 elements. Assume 1 unit time for core A or D to compute the square of an element. Thus, core B takes $\frac{1}{2}$ unit time and core C takes $\frac{1}{3}$ unit time to compute the square of an element. Given the following division of labor in four cores:

Core A	32 elements
Core B	128 elements
Core C	64 elements
Core D	32 elements

a. Compute the *total execution time* (in time units) for using the four-core processor to compute the squares of 256 elements in parallel. The four cores have different speeds. Some faster cores finish the job and may become idle, while others are still busy computing until all squares are computed.

b. Calculate the *processor utilization rate*, which is the total amount of time the cores are busy (not idle) divided by the total execution time they are using all cores in the processor to execute the above application.

Problem 1.6

Consider parallel execution of an MPI-coded C program in SPMD (single program and multiple data streams) mode on a server cluster consisting of n identical Linux servers. SPMD mode means

the same MPI program is running simultaneously on all servers but over different data sets of identical workloads. Assume that 25 percent of the program execution is attributed to the execution of MPI commands. For simplicity, assume that all MPI commands take the same amount of execution time. Answer the following questions using Amdahl's law:

a. Given that the total execution time of the MPI program on a four-server cluster is T minutes, what is the speedup factor of executing the same MPI program on a 256-server cluster, compared with using the four-server cluster? Assume that the program execution is deadlock-free and ignore all other runtime execution overheads in the calculation.

b. Suppose that all MPI commands are now enhanced by a factor of 2 by using active messages executed by message handlers at the user space. The enhancement can reduce the execution time of all MPI commands by half. What is the speedup of the 256-server cluster installed with this MPI enhancement, computed with the old 256-server cluster without MPI enhancement?

Problem 1.7

Consider a program for multiplying two large-scale $N \times N$ matrices, where N is the matrix size. The sequential multiply time on a single server is $T_1 = cN^3$ minutes, where c is a constant determined by the server used. An MPI-code parallel program requires $T_n = cN^3/n + dN^2/n^{0.5}$ minutes to complete execution on an n-server cluster system, where d is a constant determined by the MPI version used. Assume the program has a zero sequential bottleneck ($\alpha = 0$). The second term in T_n accounts for the total message-passing overhead experienced by n servers.

Answer the following questions for a given cluster configuration with $n = 64$ servers, $c = 0.8$, and $d = 0.1$. Parts (a, b) have a fixed workload corresponding to the matrix size $N = 15{,}000$. Parts (c, d) have a scaled workload associated with an enlarged matrix size $N' = n^{1/3} N = 64^{1/3} \times 15{,}000 = 4 \times 15{,}000 = 60{,}000$. Assume the same cluster configuration to process both workloads. Thus, the system parameters n, c, and d stay unchanged. Running the scaled workload, the overhead also increases with the enlarged matrix size N'.

a. Using Amdahl's law, calculate the speedup of the n-server cluster over a single server.

b. What is the efficiency of the cluster system used in Part (a)?

c. Calculate the speedup in executing the scaled workload for an enlarged $N' \times N'$ matrix on the same cluster configuration using Gustafson's law.

d. Calculate the efficiency of running the scaled workload in Part (c) on the 64-processor cluster.

e. Compare the above speedup and efficiency results and comment on their implications.

Problem 1.8

Compare the similarities and differences between traditional computing clusters/grids and the computing clouds launched in recent years. Consider all technical and economic aspects as listed below. Answer the following questions against real example systems or platforms built in recent years. Also discuss the possible convergence of the two computing paradigms in the future.

a. Hardware, software, and networking support

b. Resource allocation and provisioning methods

c. Infrastructure management and protection
d. Support of utility computing services
e. Operational and cost models applied

Problem 1.9

Answer the following questions regarding PC and HPC systems:

a. Explain why PCs and HPCs were evolutionary rather than revolutionary in the past 30 years.
b. Discuss the drawbacks in disruptive changes in processor architecture. Why is the memory wall a major problem in achieving scalable changes in performance?
c. Explain why x-86 processors are still dominating the PC and HPC markets.

Problem 1.10

Multicore and many-core processors have appeared in widespread use in both desktop computers and HPC systems. Answer the following questions regarding advanced processors, memory devices, and system interconnects:

a. What are the differences between multicore CPUs and GPUs in terms of architecture and usage?
b. Explain why parallel programming cannot match the progress of processor technology.
c. Suggest ideas and defend your argument with some plausible solutions to this mismatch problem between core scaling and effective programming and use of multicores.
d. Explain why flash memory SSD can deliver better speedups in some HPC or HTC applications.
e. Justify the prediction that InfiniBand and Ethernet will continue to dominate the HPC market.

Problem 1.11

In Figure 1.7, you studied five categories of modern processors. Characterize in Table 1.9 five micro-architectures for designing these processors. Comment on their advantages/shortcomings and identify the names of two example commercial processors that are built in each processor category. Assume a single core in the superscalar processor and the three multithreaded processors. The last processor category is a multicore CMP and each core is assumed to handle one thread at a time.

Table 1.9 Comparison of Five Micro-architectures for Modern Processors

Processor Micro-architectures	Architecture Charcteristics	Advantages/ Shortcomings	Representative Processors
Single-threaded Superscalar			
Fine-grain Multithreading			
Coarse-grain Multithreading			
Simultaneous Multithreading (SMT)			
Multicore Chip Multiprocessor (CMP)			

Problem 1.12

Discuss the major advantages and disadvantages in the following areas:

a. Why are virtual machines and virtual clusters suggested in cloud computing systems?
b. What breakthroughs are required to build virtualized cloud systems cost-effectively?
c. What are the impacts of cloud platforms on the future of the HPC and HTC industry?

Problem 1.13

Characterize the following three cloud computing models:

a. What is an IaaS (Infrastructure-as-a-Service) cloud? Give one example system.
b. What is a PaaS (Platform-as-a-Service) cloud? Give one example system.
c. What is a SaaS (Software-as-a-Service) cloud? Give one example system.

Problem 1.14

Briefly explain each of the following cloud computing services. Identify two cloud providers by company name in each service category.

a. Application cloud services
b. Platform cloud services
c. Compute and storage services
d. Collocation cloud services
e. Network cloud services

Problem 1.15

Briefly explain the following terms associated with network threats or security defense in a distributed computing system:

a. Denial of service (DoS)
b. Trojan horse
c. Network worm
d. Service spoofing
e. Authorization
f. Authentication
g. Data integrity
h. Confidentiality

Problem 1.16

Briefly answer the following questions regarding green information technology and energy efficiency in distributed systems:

a. Why is power consumption critical to data-center operations?
b. What constitutes the dynamic voltage frequency scaling (DVFS) technique?
c. Conduct in-depth research on recent progress in green IT research, and write a report on its applications to data-center design and cloud service applications.

Problem 1.17

Compare GPU and CPU chips in terms of their strengths and weaknesses. In particular, discuss the trade-offs between power efficiency, programmability, and performance. Also compare various MPP architectures in processor selection, performance target, efficiency, and packaging constraints.

Problem 1.18

Compare three distributed operating systems: Amoeba, DCE, and MOSIX. Research their recent developments and their impact on applications in clusters, grids, and clouds. Discuss the suitability of each system in its commercial or experimental distributed applications. Also discuss each system's limitations and explain why they were not successful as commercial systems.

HZ BOOKS
华章图书

CHAPTER

Computer Clusters for Scalable Parallel Computing

2

CHAPTER OUTLINE

SUMMARY

Clustering of computers enables scalable parallel and distributed computing in both science and business applications. This chapter is devoted to building cluster-structured massively parallel processors. We focus on the design principles and assessment of the hardware, software, middleware, and operating system support to achieve scalability, availability, programmability, single-system images, and fault tolerance in clusters. We will examine the cluster architectures of Tianhe-1A, Cray XT5 Jaguar, and IBM Roadrunner. The study also covers the LSF middleware and MOSIX/OS for job and resource management in Linux clusters, GPU clusters and cluster extensions to building grids and clouds. Only physical clusters are studied in this chapter. Virtual clusters will be studied in Chapters 3 and 4.

2.1 CLUSTERING FOR MASSIVE PARALLELISM

A *computer cluster* is a collection of interconnected stand-alone computers which can work together collectively and cooperatively as a single integrated computing resource pool. Clustering explores massive parallelism at the job level and achieves *high availability* (HA) through stand-alone operations. The benefits of computer clusters and *massively parallel processors* (MPPs) include scalable performance, HA, fault tolerance, modular growth, and use of commodity components. These features can sustain the generation changes experienced in hardware, software, and network components. Cluster computing became popular in the mid-1990s as traditional mainframes and vector supercomputers were proven to be less cost-effective in many *high-performance computing* (HPC) applications.

Of the Top 500 supercomputers reported in 2010, 85 percent were computer clusters or MPPs built with homogeneous nodes. Computer clusters have laid the foundation for today's supercomputers, computational grids, and Internet clouds built over data centers. We have come a long way toward becoming addicted to computers. According to a recent IDC prediction, the HPC market will increase from $8.5 billion in 2010 to $10.5 billion by 2013. A majority of the Top 500 supercomputers are used for HPC applications in science and engineering. Meanwhile, the use of *high-throughput computing* (HTC) clusters of servers is growing rapidly in business and web services applications.

2.1.1 Cluster Development Trends

Support for clustering of computers has moved from interconnecting high-end mainframe computers to building clusters with massive numbers of x86 engines. Computer clustering started with the linking of large mainframe computers such as the IBM Sysplex and the SGI Origin 3000. Originally, this was motivated by a demand for cooperative group computing and to provide higher availability in critical enterprise applications. Subsequently, the clustering trend moved toward the networking of many minicomputers, such as DEC's VMS cluster, in which multiple VAXes were interconnected to share the same set of disk/tape controllers. Tandem's Himalaya was designed as a business cluster for fault-tolerant *online transaction processing* (OLTP) applications.

In the early 1990s, the next move was to build UNIX-based workstation clusters represented by the Berkeley NOW (Network of Workstations) and IBM SP2 AIX-based server cluster. Beyond

2000, we see the trend moving to the clustering of RISC or x86 PC engines. Clustered products now appear as integrated systems, software tools, availability infrastructure, and operating system extensions. This clustering trend matches the downsizing trend in the computer industry. Supporting clusters of smaller nodes will increase sales by allowing modular incremental growth in cluster configurations. From IBM, DEC, Sun, and SGI to Compaq and Dell, the computer industry has leveraged clustering of low-cost servers or x86 desktops for their cost-effectiveness, scalability, and HA features.

2.1.1.1 Milestone Cluster Systems

Clustering has been a hot research challenge in computer architecture. Fast communication, job scheduling, SSI, and HA are active areas in cluster research. Table 2.1 lists some milestone cluster research projects and commercial cluster products. Details of these old clusters can be found in [14]. These milestone projects have pioneered clustering hardware and middleware development over the past two decades. Each cluster project listed has developed some unique features. Modern clusters are headed toward HPC clusters as studied in Section 2.5.

The NOW project addresses a whole spectrum of cluster computing issues, including architecture, software support for web servers, single system image, I/O and file system, efficient communication, and enhanced availability. The Rice University TreadMarks is a good example of software-implemented shared-memory cluster of workstations. The memory sharing is implemented with a user-space runtime library. This was a research cluster built over Sun Solaris workstations. Some cluster OS functions were developed, but were never marketed successfully.

Table 2.1 Milestone Research or Commercial Cluster Computer Systems [14]

Project	Special Features That Support Clustering
DEC VAXcluster (1991)	A UNIX cluster of symmetric multiprocessing (SMP) servers running the VMS OS with extensions, mainly used in HA applications
U.C. Berkeley NOW Project (1995)	A serverless network of workstations featuring active messaging, cooperative filing, and GLUnix development
Rice University TreadMarks (1996)	Software-implemented distributed shared memory for use in clusters of UNIX workstations based on page migration
Sun Solaris MC Cluster (1995)	A research cluster built over Sun Solaris workstations; some cluster OS functions were developed but were never marketed successfully
Tandem Himalaya Cluster (1994)	A scalable and fault-tolerant cluster for OLTP and database processing, built with nonstop operating system support
IBM SP2 Server Cluster (1996)	An AIX server cluster built with Power2 nodes and the Omega network, and supported by IBM LoadLeveler and MPI extensions
Google Search Engine Cluster (2003)	A 4,000-node server cluster built for Internet search and web service applications, supported by a distributed file system and fault tolerance
MOSIX (2010) www.mosix.org	A distributed operating system for use in Linux clusters, multiclusters, grids, and clouds; used by the research community

A Unix cluster of SMP servers running VMS/OS with extensions, mainly used in high-availability applications. An AIX server cluster built with Power2 nodes and Omega network and supported by IBM Loadleveler and MPI extensions. A scalable and fault-tolerant cluster for OLTP and database processing built with non-stop operating system support. The Google search engine was built at Google using commodity components. MOSIX is a distributed operating systems for use in Linux clusters, multi-clusters, grids, and the clouds, originally developed by Hebrew University in 1999.

2.1.2 Design Objectives of Computer Clusters

Clusters have been classified in various ways in the literature. We classify clusters using six orthogonal attributes: *scalability*, *packaging*, *control*, *homogeneity*, *programmability*, and *security*.

2.1.2.1 Scalability

Clustering of computers is based on the concept of modular growth. To scale a cluster from hundreds of uniprocessor nodes to a supercluster with 10,000 multicore nodes is a nontrivial task. The scalability could be limited by a number of factors, such as the multicore chip technology, cluster topology, packaging method, power consumption, and cooling scheme applied. The purpose is to achieve scalable performance constrained by the aforementioned factors. We have to also consider other limiting factors such as the memory wall, disk I/O bottlenecks, and latency tolerance, among others.

2.1.2.2 Packaging

Cluster nodes can be packaged in a compact or a slack fashion. In a *compact* cluster, the nodes are closely packaged in one or more racks sitting in a room, and the nodes are not attached to peripherals (monitors, keyboards, mice, etc.). In a *slack* cluster, the nodes are attached to their usual peripherals (i.e., they are complete SMPs, workstations, and PCs), and they may be located in different rooms, different buildings, or even remote regions. Packaging directly affects communication wire length, and thus the selection of interconnection technology used. While a compact cluster can utilize a high-bandwidth, low-latency communication network that is often proprietary, nodes of a slack cluster are normally connected through standard LANs or WANs.

2.1.2.3 Control

A cluster can be either controlled or managed in a *centralized* or *decentralized* fashion. A compact cluster normally has centralized control, while a slack cluster can be controlled either way. In a centralized cluster, all the nodes are owned, controlled, managed, and administered by a central operator. In a decentralized cluster, the nodes have individual owners. For instance, consider a cluster comprising an interconnected set of desktop workstations in a department, where each workstation is individually owned by an employee. The owner can reconfigure, upgrade, or even shut down the workstation at any time. This lack of a single point of control makes system administration of such a cluster very difficult. It also calls for special techniques for process scheduling, workload migration, checkpointing, accounting, and other similar tasks.

2.1.2.4 Homogeneity

A *homogeneous* cluster uses nodes from the same platform, that is, the same processor architecture and the same operating system; often, the nodes are from the same vendors. A *heterogeneous*

cluster uses nodes of different platforms. Interoperability is an important issue in heterogeneous clusters. For instance, process migration is often needed for load balancing or availability. In a homogeneous cluster, a binary process image can migrate to another node and continue execution. This is not feasible in a heterogeneous cluster, as the binary code will not be executable when the process migrates to a node of a different platform.

2.1.2.5 Security

Intracluster communication can be either *exposed* or *enclosed*. In an exposed cluster, the communication paths among the nodes are exposed to the outside world. An outside machine can access the communication paths, and thus individual nodes, using standard protocols (e.g., TCP/IP). Such exposed clusters are easy to implement, but have several disadvantages:

- Being exposed, intracluster communication is not secure, unless the communication subsystem performs additional work to ensure privacy and security.
- Outside communications may disrupt intracluster communications in an unpredictable fashion. For instance, heavy BBS traffic may disrupt production jobs.
- Standard communication protocols tend to have high overhead.

In an enclosed cluster, intracluster communication is shielded from the outside world, which alleviates the aforementioned problems. A disadvantage is that there is currently no standard for efficient, enclosed intracluster communication. Consequently, most commercial or academic clusters realize fast communications through one-of-a-kind protocols.

2.1.2.6 Dedicated versus Enterprise Clusters

A *dedicated cluster* is typically installed in a deskside rack in a central computer room. It is homogeneously configured with the same type of computer nodes and managed by a single administrator group like a frontend host. Dedicated clusters are used as substitutes for traditional mainframes or supercomputers. A dedicated cluster is installed, used, and administered as a single machine. Many users can log in to the cluster to execute both interactive and batch jobs. The cluster offers much enhanced throughput, as well as reduced response time.

An *enterprise cluster* is mainly used to utilize idle resources in the nodes. Each node is usually a full-fledged SMP, workstation, or PC, with all the necessary peripherals attached. The nodes are typically geographically distributed, and are not necessarily in the same room or even in the same building. The nodes are individually owned by multiple owners. The cluster administrator has only limited control over the nodes, as a node can be turned off at any time by its owner. The owner's "local" jobs have higher priority than enterprise jobs. The cluster is often configured with heterogeneous computer nodes. The nodes are often connected through a low-cost Ethernet network. Most data centers are structured with clusters of low-cost servers. Virtual clusters play a crucial role in upgrading data centers. We will discuss virtual clusters in Chapter 6 and clouds in Chapters 7, 8, and 9.

2.1.3 Fundamental Cluster Design Issues

In this section, we will classify various cluster and MPP families. Then we will identify the major design issues of clustered and MPP systems. Both physical and virtual clusters are covered. These systems are often found in computational grids, national laboratories, business data centers,

supercomputer sites, and virtualized cloud platforms. A good understanding of how clusters and MPPs work collectively will pave the way toward understanding the ins and outs of large-scale grids and Internet clouds in subsequent chapters. Several issues must be considered in developing and using a cluster. Although much work has been done in this regard, this is still an active research and development area.

2.1.3.1 Scalable Performance

This refers to the fact that scaling of resources (cluster nodes, memory capacity, I/O bandwidth, etc.) leads to a proportional increase in performance. Of course, both scale-up and scale-down capabilities are needed, depending on application demand or cost-effectiveness considerations. Clustering is driven by scalability. One should not ignore this factor in all applications of cluster or MPP computing systems.

2.1.3.2 Single-System Image (SSI)

A set of workstations connected by an Ethernet network is not necessarily a cluster. A cluster is a single system. For example, suppose a workstation has a 300 Mflops/second processor, 512 MB of memory, and a 4 GB disk and can support 50 active users and 1,000 processes. By clustering 100 such workstations, can we get a single system that is equivalent to one huge workstation, or a *megastation*, that has a 30 Gflops/second processor, 50 GB of memory, and a 400 GB disk and can support 5,000 active users and 100,000 processes? This is an appealing goal, but it is very difficult to achieve. SSI techniques are aimed at achieving this goal.

2.1.3.3 Availability Support

Clusters can provide cost-effective HA capability with lots of redundancy in processors, memory, disks, I/O devices, networks, and operating system images. However, to realize this potential, availability techniques are required. We will illustrate these techniques later in the book, when we discuss how DEC clusters (Section 10.4) and the IBM SP2 (Section 10.3) attempt to achieve HA.

2.1.3.4 Cluster Job Management

Clusters try to achieve high system utilization from traditional workstations or PC nodes that are normally not highly utilized. Job management software is required to provide batching, load balancing, parallel processing, and other functionality. We will study cluster job management systems in Section 3.4. Special software tools are needed to manage multiple jobs simultaneously.

2.1.3.5 Internode Communication

Because of their higher node complexity, cluster nodes cannot be packaged as compactly as MPP nodes. The internode physical wire lengths are longer in a cluster than in an MPP. This is true even for centralized clusters. A long wire implies greater interconnect network latency. But more importantly, longer wires have more problems in terms of reliability, clock skew, and cross talking. These problems call for reliable and secure communication protocols, which increase overhead. Clusters often use commodity networks (e.g., Ethernet) with standard protocols such as TCP/IP.

2.1.3.6 Fault Tolerance and Recovery

Clusters of machines can be designed to eliminate all single points of failure. Through redundancy, a cluster can tolerate faulty conditions up to a certain extent. Heartbeat mechanisms can be installed

to monitor the running condition of all nodes. In case of a node failure, critical jobs running on the failing nodes can be saved by failing over to the surviving node machines. Rollback recovery schemes restore the computing results through periodic checkpointing.

2.1.3.7 Cluster Family Classification

Based on application demand, computer clusters are divided into three classes:

- **Compute clusters** These are clusters designed mainly for collective computation over a single large job. A good example is a cluster dedicated to numerical simulation of weather conditions. The compute clusters do not handle many I/O operations, such as database services. When a single compute job requires frequent communication among the cluster nodes, the cluster must share a dedicated network, and thus the nodes are mostly homogeneous and tightly coupled. This type of clusters is also known as a ***Beowulf cluster***.
  When the nodes require internode communication over a small number of heavy-duty nodes, they are essentially known as a *computational grid*. Tightly coupled compute clusters are designed for supercomputing applications. Compute clusters apply middleware such as a message-passing interface (**MPI**) or Parallel Virtual Machine (**PVM**) to port programs to a wide variety of clusters.
- **High-Availability clusters** HA (high-availability) clusters are designed to be fault-tolerant and achieve HA of services. HA clusters operate with many redundant nodes to sustain faults or failures. The simplest HA cluster has only two nodes that can fail over to each other. Of course, high redundancy provides higher availability. HA clusters should be designed to avoid all single points of failure. Many commercial HA clusters are available for various operating systems.
- **Load-balancing clusters** These clusters shoot for higher resource utilization through load balancing among all participating nodes in the cluster. All nodes share the workload or function as a single *virtual machine* (VM). Requests initiated from the user are distributed to all node computers to form a cluster. This results in a balanced workload among different machines, and thus higher resource utilization or higher performance. Middleware is needed to achieve dynamic load balancing by job or process migration among all the cluster nodes.

2.1.4 Analysis of the Top 500 Supercomputers

Every six months, the world's Top 500 supercomputers are evaluated by running the Linpack Benchmark program over very large data sets. The ranking varies from year to year, similar to a competition. In this section, we will analyze the historical share in architecture, speed, operating systems, countries, and applications over time. In addition, we will compare the top five fastest systems in 2010.

2.1.4.1 Architectural Evolution

It is interesting to observe in Figure 2.1 the architectural evolution of the Top 500 supercomputers over the years. In 1993, 250 systems assumed the SMP architecture and these SMP systems all disappeared in June of 2002. Most SMPs are built with shared memory and shared I/O devices. There were 120 MPP systems built in 1993, MPPs reached the peak mof 350 systems in mod-200, and dropped to less than 100 systems in 2010. The single instruction, multiple data (SIMD) machines disappeared in 1997. The cluster architecture appeared in a few systems in 1999. The cluster systems are now populated in the Top-500 list with more than 400 systems as the dominating architecture class.

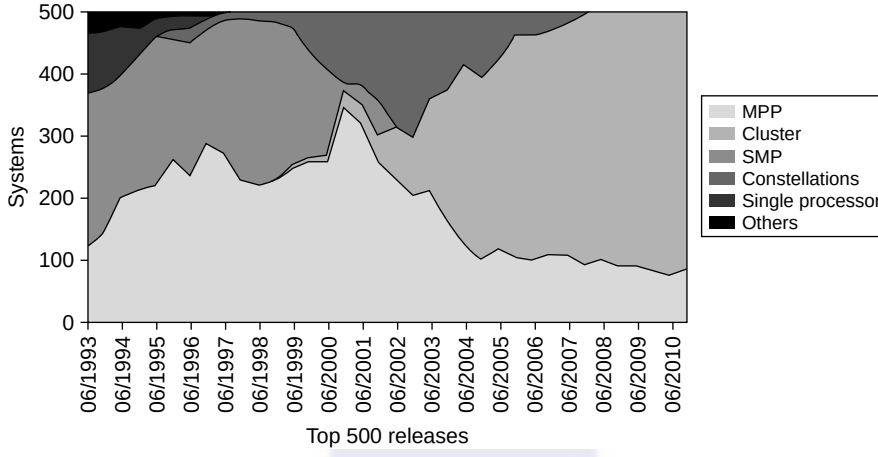


FIGURE 2.1

Architectural share of the Top 500 systems.

(*Courtesy of www.top500.org [25]*)

In 2010, the Top 500 architecture is dominated by clusters (420 systems) and MPPs (80 systems). The basic distinction between these two classes lies in the components they use to build the systems. Clusters are often built with commodity hardware, software, and network components that are commercially available. MPPs are built with custom-designed compute nodes, boards, modules, and cabinets that are interconnected by special packaging. MPPs demand high bandwidth, low latency, better power efficiency, and high reliability. Cost-wise, clusters are affordable by allowing modular growth with scaling capability. The fact that MPPs appear in a much smaller quantity is due to their high cost. Typically, only a few MPP-based supercomputers are installed in each country.

2.1.4.2 Speed Improvement over Time

Figure 2.2 plots the measured performance of the Top 500 fastest computers from 1993 to 2010. The y-axis is scaled by the sustained speed performance in terms of Gflops, Tflops, and Pflops. The middle curve plots the performance of the fastest computers recorded over 17 years; peak performance increases from 58.7 Gflops to 2.566 Pflops. The bottom curve corresponds to the speed of the 500th computer, which increased from 0.42 Gflops in 1993 to 31.1 Tflops in 2010. The top curve plots the speed sum of all 500 computers over the same period. In 2010, the total speed sum of 43.7 Pflops was achieved by all 500 computers, collectively. It is interesting to observe that the total speed sum increases almost linearly with time.

2.1.4.3 Operating System Trends in the Top 500

The five most popular operating systems have more than a 10 percent share among the Top 500 computers, according to data released by TOP500.org (www.top500.org/stats/list/36/os) in November 2010. According to the data, 410 supercomputers are using Linux with a total processor count exceeding 4.5 million. This constitutes 82 percent of the systems adopting Linux. The IBM AIX/OS is in second place with 17 systems (a 3.4 percent share) and more than 94,288 processors.

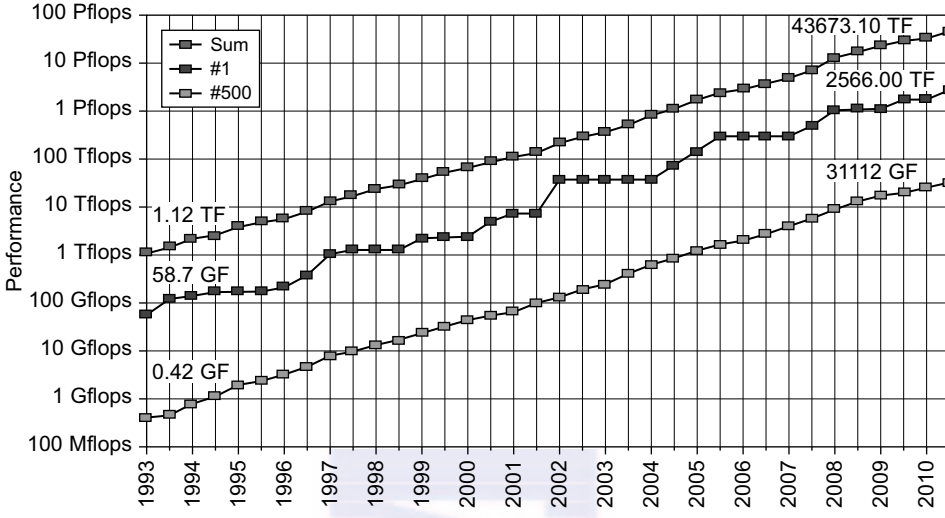


FIGURE 2.2

Performance plot of the Top 500 supercomputers from 1993 to 2010.

(*Courtesy of www.top500.org [25]*)

Third place is represented by the combined use of the SLEs10 with the SGI ProPack5, with 15 systems (3 percent) over 135,200 processors. Fourth place goes to the CNK/SLES9 used by 14 systems (2.8 percent) over 1.13 million processors. Finally, the CNL/OS was used in 10 systems (2 percent) over 178,577 processors. The remaining 34 systems applied 13 other operating systems with a total share of only 6.8 percent. In conclusion, the Linux OS dominates the systems in the Top 500 list.

2.1.4.4 The Top Five Systems in 2010

In Table 2.2, we summarize the key architecture features and sustained Linpack Benchmark performance of the top five supercomputers reported in November 2010. The Tianhe-1A was ranked as the fastest MPP in late 2010. This system was built with 86,386 Intel Xeon CPUs and NVIDIA GPUs by the National University of Defense Technology in China. We will present in Section 2.5 some of the top winners: namely the Tianhe-1A, Cray Jaguar, Nebulae, and IBM Roadrunner that were ranked among the top systems from 2008 to 2010. All the top five machines in Table 2.3 have achieved a speed higher than 1 Pflops. The *sustained speed*, $\boldsymbol{R_{max}}$, in Pflops is measured from the execution of the Linpack Benchmark program corresponding to a maximum matrix size.

The *system efficiency* reflects the ratio of the *sustained speed* to the *peak speed*, $\boldsymbol{R_{peak}}$, when all computing elements are fully utilized in the system. Among the top five systems, the two U.S.-built systems, the Jaguar and the Hopper, have the highest efficiency, more than 75 percent. The two systems built in China, the Tianhe-1A and the Nebulae, and Japan's TSUBAME 2.0, are all low in efficiency. In other words, these systems should still have room for improvement in the future. The average power consumption in these 5 systems is 3.22 MW. This implies that excessive power consumption may post the limit to build even faster supercomputers in the future. These top systems all emphasize massive parallelism using up to 250,000 processor cores per system.

Table 2.2 The Top Five Supercomputers Evaluated in November 2010

System Name, Site, and URL	System Name, Processors, OS, Topology, and Developer	Linpack Speed (R_{max}), Power	Efficiency (R_{max}/R_{peak})
1. Tianhe-1A, National Supercomputing Center, Tianjin, China, http://www.nscc-tj.gov.cn/en/	NUDT TH1A with 14,336 Xeon X5670 CPUs (six cores each) plus 7168 NVIDIA Tesla M2050 GPUs (448 CUDA cores each), running Linux, built by National Univ. of Defense Technology, China	2.57 Pflops, 4.02 MW	54.6% (over a peak of 4.7 Pflops)
2. Jaguar, DOE/SC/Oak Ridge National Lab., United States, http://computing.ornl.gov	Cray XT5-HE: MPP with 224,162 x 6 AMD Opteron, 3D torus network, Linux (CLE), manufactured by Cray, Inc.	1.76 Pflops, 6.95 MW	75.6% (over a peak of 4.7 Pflops)
3. Nebulae at China's National Supercomputer Center, ShenZhen, China http://www.ict.cas.cas.cn	TC3600 Blade, 120,640 cores in 55,680 Xeon X5650 plus 64,960 NVIDIA Tesla C2050 GPUs, Linux, InfiniBand, built by Dawning, Inc.	1.27 Pflops, 2.55 MW	42.6% (over a peak of 2.98 Pflops)
4. TSUBAME 2.0, GSIC Center, Tokyo Institute of Technology, Tokyo, Japan, http://www.gsic.titech.ac.jp/	HP cluster, 3000SL, 73,278 x 6 Xeon X5670 processors, NVIDIA GPU, Linux/SLES 11, built by NEC/HP	1.19 Pflops, 1.8 MW	52.19% (over a peak of 2.3 Pflops)
5. Hopper, DOE/SC/LBNL/ NERSC, Berkeley, CA. USA, http//www/nersc.gov/	Cray XE6 150,408 x 12 AMD Opteron, Linux (CLE), built by Cray, Inc.	1.05 Pflops, 2.8 MW	78.47% (over a peak of 1.35 Pflops)

Table 2.3 Sample Compute Node Architectures for Large Cluster Construction

Node Architecture	Major Characteristics	Representative Systems
Homogeneous node using the same multicore processors	Multicore processors mounted on the same node with a crossbar connected to shared memory or local disks	The Cray XT5 uses two six-core AMD Opteron processors in each compute node
Hybrid nodes using CPU plus GPU or FLP accelerators	General-purpose CPU for integer operations, with GPUs acting as coprocessors to speed up FLP operations	China's Tianhe-1A uses two Intel Xeon processors plus one NVIDIA GPU per compute node

2.1.4.5 Country Share and Application Share

In the 2010 Top-500 list, there were 274 supercomputing systems installed in the US, 41 systems in China, and 103 systems in Japan, France, UK, and Germany. The remaining countries have 82 systems. This country share roughly reflects the countries' economic growth over the years. Countries compete in the Top 500 race every six months. Major increases of supercomputer applications are in the areas of database, research, finance, and information services.

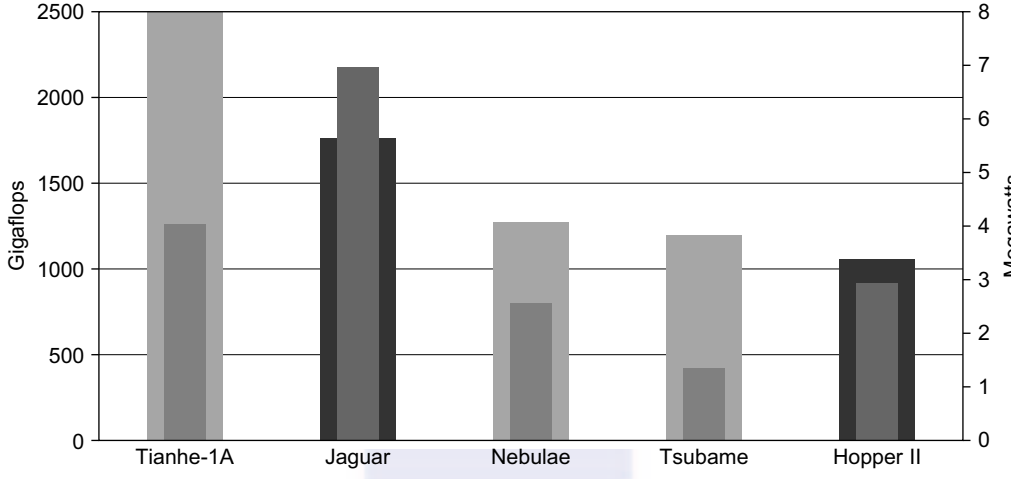


FIGURE 2.3

Power and performance of the top 5 supercomputers in November 2010.

(*Courtesy of www.top500.org [25] and B. Dally [10]*)

2.1.4.6 Power versus Performance in the Top Five in 2010

In Figure 2.3, the top five supercomputers are ranked by their speed (Gflops) on the left side and by their power consumption (MW per system) on the right side. The Tiahhe-1A scored the highest with a 2.57 Pflops speed and 4.01 MW power consumption. In second place, the Jaguar consumes the highest power of 6.9 MW. In fourth place, the TSUBAME system consumes the least power, 1.5 MW, and has a speed performance that almost matches that of the Nebulae system. One can define a performance/power ratio to see the trade-off between these two metrics. There is also a Top 500 Green contest that ranks the supercomputers by their power efficiency. This chart shows that all systems using the hybrid CPU/GPU architecture consume much less power.

2.2 COMPUTER CLUSTERS AND MPP ARCHITECTURES

Most clusters emphasize higher availability and scalable performance. Clustered systems evolved from the Microsoft Wolfpack and Berkeley NOW to the SGI Altix Series, IBM SP Series, and IBM Roadrunner. NOW was a UC Berkeley research project designed to explore new mechanisms for clustering of UNIX workstations. Most clusters use commodity networks such as Gigabit Ethernet, Myrinet switches, or InfiniBand networks to interconnect the compute and storage nodes. The clustering trend moves from supporting large rack-size, high-end computer systems to high-volume, desktop or deskside computer systems, matching the downsizing trend in the computer industry.

2.2.1 Cluster Organization and Resource Sharing

In this section, we will start by discussing basic, small-scale PC or server clusters. We will discuss how to construct large-scale clusters and MPPs in subsequent sections.

2.2.1.1 A Basic Cluster Architecture

Figure 2.4 shows the basic architecture of a computer cluster over PCs or workstations. The figure shows a simple cluster of computers built with commodity components and fully supported with desired SSI features and HA capability. The processing nodes are commodity workstations, PCs, or servers. These commodity nodes are easy to replace or upgrade with new generations of hardware. The node operating systems should be designed for multiuser, multitasking, and multithreaded applications. The nodes are interconnected by one or more fast commodity networks. These networks use standard communication protocols and operate at a speed that should be two orders of magnitude faster than that of the current TCP/IP speed over Ethernet.

The network interface card is connected to the node's standard I/O bus (e.g., PCI). When the processor or the operating system is changed, only the driver software needs to change. We desire to have a platform-independent *cluster operating system*, sitting on top of the node platforms. But such a cluster OS is not commercially available. Instead, we can deploy some cluster middleware to glue together all node platforms at the user space. An availability middleware offers HA services. An SSI layer provides a single entry point, a single file hierarchy, a single point of control, and a

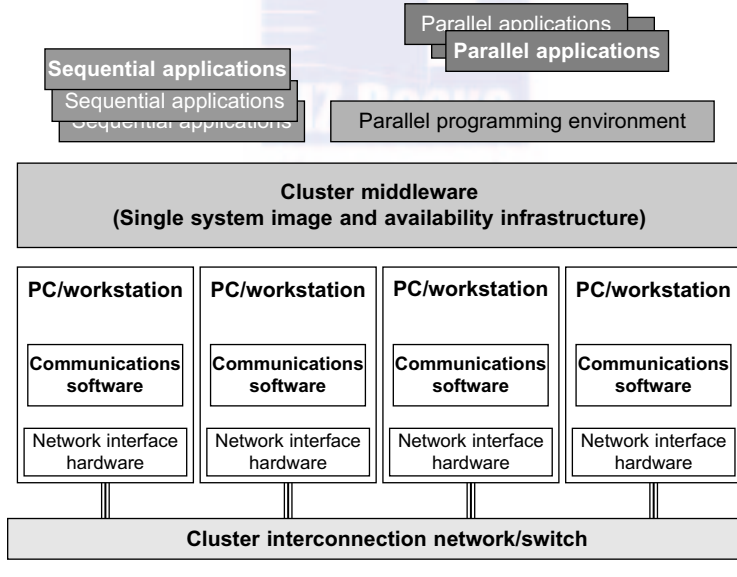


FIGURE 2.4

The architecture of a computer cluster built with commodity hardware, software, middleware, and network components supporting HA and SSI.

(*Courtesy of M. Baker and R. Buyya, reprinted with Permission [3]*)

single job management system. Single memory may be realized with the help of the compiler or a runtime library. A single process space is not necessarily supported.

In general, an idealized cluster is supported by three subsystems. First, conventional databases and OLTP monitors offer users a desktop environment in which to use the cluster. In addition to running sequential user programs, the cluster supports parallel programming based on standard languages and communication libraries using PVM, MPI, or OpenMP. The programming environment also includes tools for debugging, profiling, monitoring, and so forth. A user interface subsystem is needed to combine the advantages of the web interface and the Windows GUI. It should also provide user-friendly links to various programming environments, job management tools, hypertext, and search support so that users can easily get help in programming the computer cluster.

2.2.1.2 Resource Sharing in Clusters

Supporting clusters of smaller nodes will increase computer sales. Clustering improves both availability and performance. These two clustering goals are not necessarily in conflict. Some HA clusters use hardware redundancy for scalable performance. The nodes of a cluster can be connected in one of three ways, as shown in Figure 2.5. The shared-nothing architecture is used in most clusters, where the nodes are connected through the I/O bus. The shared-disk architecture is in favor of small-scale *availability clusters* in business applications. When one node fails, the other node takes over.

The shared-nothing configuration in Part (a) simply connects two or more autonomous computers via a LAN such as Ethernet. A shared-disk cluster is shown in Part (b). This is what most business clusters desire so that they can enable recovery support in case of node failure. The shared disk can hold checkpoint files or critical system images to enhance cluster availability. Without shared disks, checkpointing, rollback recovery, failover, and failback are not possible in a cluster. The shared-memory cluster in Part (c) is much more difficult to realize. The nodes could be connected by a *scalable coherence interface (SCI)* ring, which is connected to the memory bus of each node through an NIC module. In the other two architectures, the interconnect is attached to the I/O bus. The memory bus operates at a higher frequency than the I/O bus.

There is no widely accepted standard for the memory bus. But there are such standards for the I/O buses. One recent, popular standard is the PCI I/O bus standard. So, if you implement an NIC card to attach a faster Ethernet network to the PCI bus you can be assured that this card can be used in other systems that use PCI as the I/O bus. The I/O bus evolves at a much slower rate than the memory bus. Consider a cluster that uses connections through the PCI bus. When the processors are upgraded, the interconnect and the NIC do not have to change, as long as the new system still uses PCI. In a shared-memory cluster, changing the processor implies a redesign of the node board and the NIC card.

2.2.2 Node Architectures and MPP Packaging

In building large-scale clusters or MPP systems, cluster nodes are classified into two categories: *compute nodes* and *service nodes*. Compute nodes appear in larger quantities mainly used for large-scale searching or parallel floating-point computations. Service nodes could be built with different processors mainly used to handle I/O, file access, and system monitoring. For MPP clusters, the

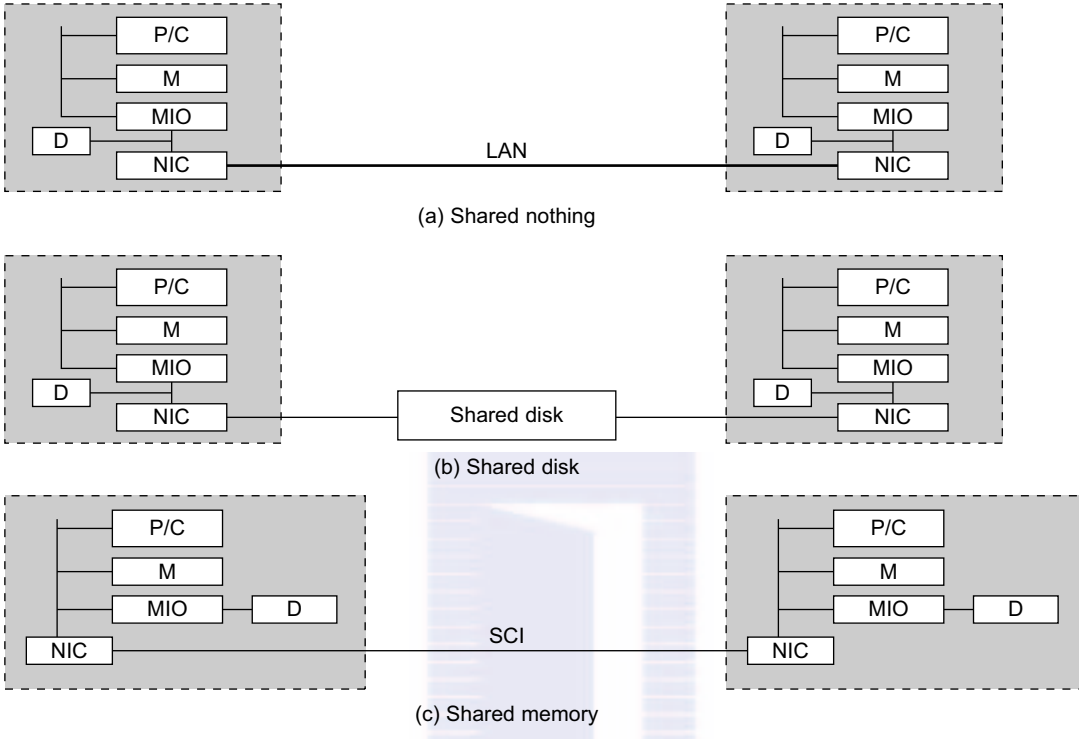


FIGURE 2.5

Three ways to connect cluster nodes (P/C: Processor and Cache; M: Memory; D: Disk; NIC: Network Interface Circuitry; MIO: Memory-I/O Bridge.)

(*Courtesy of Hwang and Xu [14]*)

compute nodes dominate in system cost, because we may have 1,000 times more compute nodes than service nodes in a single large clustered system. Table 2.3 introduces two example compute node architectures: *homogeneous design* and *hybrid node design.*

In the past, most MPPs are built with a homogeneous architecture by interconnecting a large number of the same compute nodes. In 2010, the Cray XT5 Jaguar system was built with 224,162 AMD Opteron processors with six cores each. The Tiahe-1A adopted a hybrid node design using two Xeon CPUs plus two AMD GPUs per each compute node. The GPU could be replaced by special floating-point accelerators. A homogeneous node design makes it easier to program and maintain the system.

Example 2.1 Modular Packaging of the IBM Blue Gene/L System

The Blue Gene/L is a supercomputer jointly developed by IBM and Lawrence Livermore National Laboratory. The system became operational in 2005 with a 136 Tflops performance at the No. 1 position in the

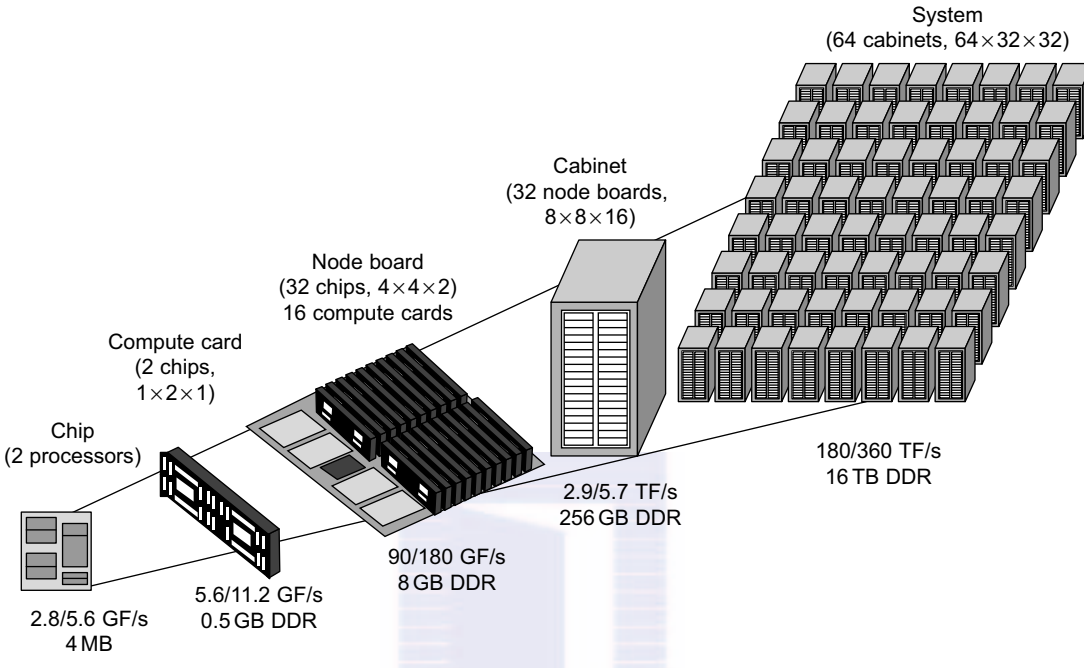


FIGURE 2.6

The IBM Blue Gene/L architecture built with modular components packaged hierarchically in five levels.

(*Courtesy of N. Adiga, et al., IBM Corp., 2005 [1]*)

Top-500 list—toped the Japanese Earth Simulator. The system was upgraded to score a 478 Tflops speed in 2007. By examining the architecture of the Blue Gene series, we reveal the modular construction of a scalable MPP system as shown in Figure 2.6. With modular packaging, the Blue Gene/L system is constructed hierarchically from processor chips to 64 physical racks. This system was built with a total of 65,536 nodes with two PowerPC 449 FP2 processors per node. The 64 racks are interconnected by a huge 3D 64 x 32 x 32 torus network.

■

In the lower-left corner, we see a dual-processor chip. Two chips are mounted on a computer card. Sixteen computer cards (32 chips or 64 processors) are mounted on a node board. A cabinet houses 32 node boards with an 8 x 8 x 16 torus interconnect. Finally, 64 cabinets (racks) form the total system at the upper-right corner. This packaging diagram corresponds to the 2005 configuration. Customers can order any size to meet their computational needs. The Blue Gene cluster was designed to achieve scalable performance, reliability through built-in testability, resilience by preserving locality of failures and checking mechanisms, and serviceability through partitioning and isolation of fault locations.

2.2.3 Cluster System Interconnects

2.2.3.1 High-Bandwidth Interconnects

Table 2.4 compares four families of high-bandwidth system interconnects. In 2007, Ethernet used a 1 Gbps link, while the fastest InfiniBand links ran at 30 Gbps. The Myrinet and Quadrics perform in between. The MPI latency represents the state of the art in long-distance message passing. All four technologies can implement any network topology, including crossbar switches, fat trees, and torus networks. The InfiniBand is the most expensive choice with the fastest link speed. The Ethernet is still the most cost-effective choice. We consider two example cluster interconnects over 1,024 nodes in Figure 2.7 and Figure 2.9. The popularity of five cluster interconnects is compared in Figure 2.8.

Table 2.4 Comparison of Four Cluster Interconnect Technologies Reported in 2007

Feature	Myrinet	Quadrics	InfiniBand	Ethernet
Available link speeds	1.28 Gbps (*M-XP*) 10 Gbps (*M-10G*)	2.8 Gbps (*QsNet*) 7.2 Gbps (*QsNetII*)	2.5 Gbps (*1X*) 10 Gbps (*4X*) 30 Gbps (*12X*)	1 Gbps
MPI latency	~3 us	~3 us	~4.5 us	~40 us
Network processor	Yes	Yes	Yes	No
Topologies	Any	Any	Any	Any
Network topology	Clos	Fat tree	Fat tree	Any
Routing	Source-based, cut-through	Source-based, cut-through	Destination-based	Destination-based
Flow control	Stop and go	Worm-hole	Credit-based	802.3x

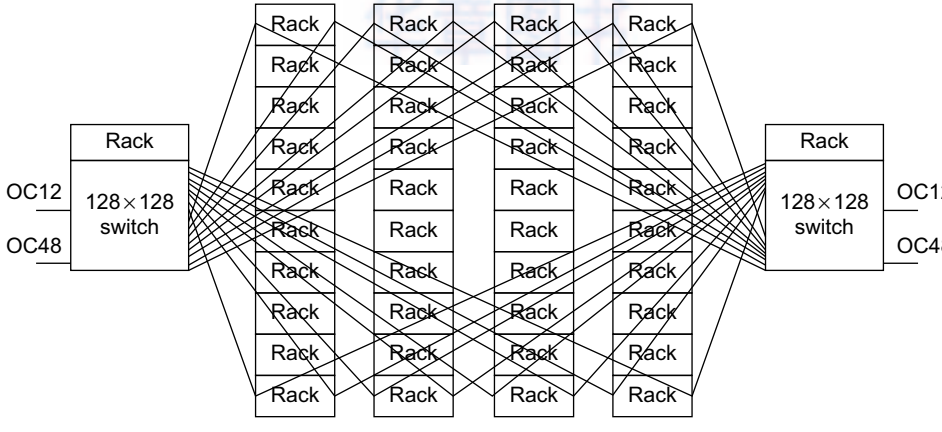


FIGURE 2.7

Google search engine cluster architecture.

(*Courtesy of Google, Inc. [6]*)

Example 2.2 Crossbar Switch in Google Search Engine Cluster

Google has many data centers using clusters of low-cost PC engines. These clusters are mainly used to support Google's web search business. Figure 2.7 shows a Google cluster interconnect of 40 racks of PC engines via two racks of 128 x 128 Ethernet switches. Each Ethernet switch can handle 128 one Gbps Ethernet links. A rack contains 80 PCs. This is an earlier cluster of 3,200 PCs. Google's search engine clusters are built with a lot more nodes. Today's server clusters from Google are installed in data centers with container trucks.

Two switches are used to enhance cluster availability. The cluster works fine even when one switch fails to provide the links among the PCs. The front ends of the switches are connected to the Internet via 2.4 Gbps OC 12 links. The 622 Mbps OC 12 links are connected to nearby data-center networks. In case of failure of the OC 48 links, the cluster is still connected to the outside world via the OC 12 links. Thus, the Google cluster eliminates all single points of failure.

2.2.3.2 Share of System Interconnects over Time

Figure 2.8 shows the distribution of large-scale system interconnects in the Top 500 systems from 2003 to 2008. Gigabit Ethernet is the most popular interconnect due to its low cost and market readiness. The InfiniBand network has been chosen in about 150 systems for its high-bandwidth performance. The Cray interconnect is designed for use in Cray systems only. The use of Myrinet and Quadrics networks had declined rapidly in the Top 500 list by 2008.

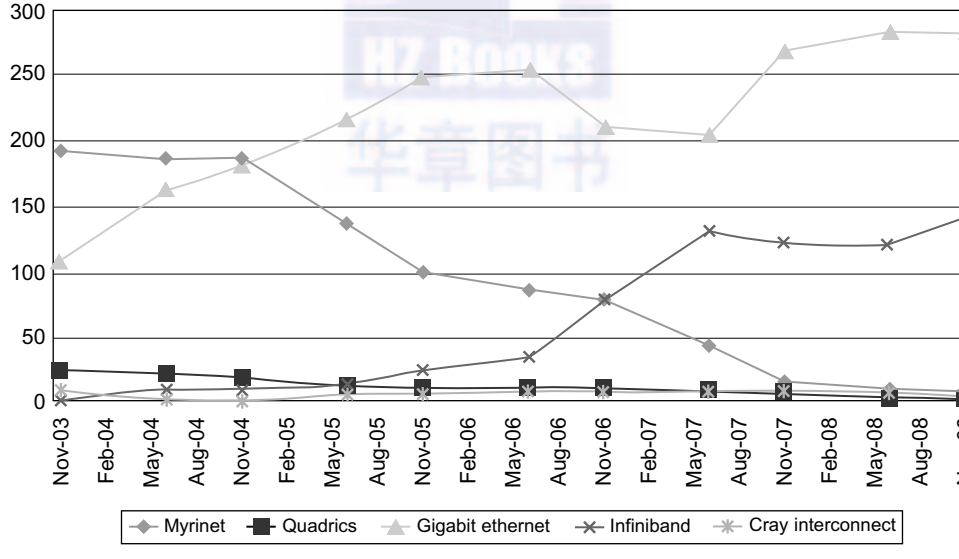


FIGURE 2.8

Distribution of high-bandwidth interconnects in the Top 500 systems from 2003 to 2008.

(*Courtesy of www.top500.org [25]*)

Example 2.3 Understanding the InfiniBand Architecture [8]

The InfiniBand has a switch-based point-to-point interconnect architecture. A large InfiniBand has a layered architecture. The interconnect supports the virtual interface architecture (VIA) for distributed messaging. The InfiniBand switches and links can make up any topology. Popular ones include crossbars, fat trees, and torus networks. Figure 2.9 shows the layered construction of an InfiniBand network. According to Table 2.5, the InfiniBand provides the highest speed links and the highest bandwidth in reported large-scale systems. However, InfiniBand networks cost the most among the four interconnect technologies.

Each end point can be a storage controller, a network interface card (NIC), or an interface to a host system. A host channel adapter (HCA) connected to the host processor through a standard peripheral component interconnect (PCI), PCI extended (PCI-X), or PCI express bus provides the host interface. Each HCA has more than one InfiniBand port. A target channel adapter (TCA) enables I/O devices to be loaded within the network. The TCA includes an I/O controller that is specific to its particular device's protocol such as SCSI, Fibre Channel, or Ethernet. This architecture can be easily implemented to build very large scale cluster interconnects that connect thousands or more hosts together. Supporting the InfiniBand in cluster applications can be found in [8].

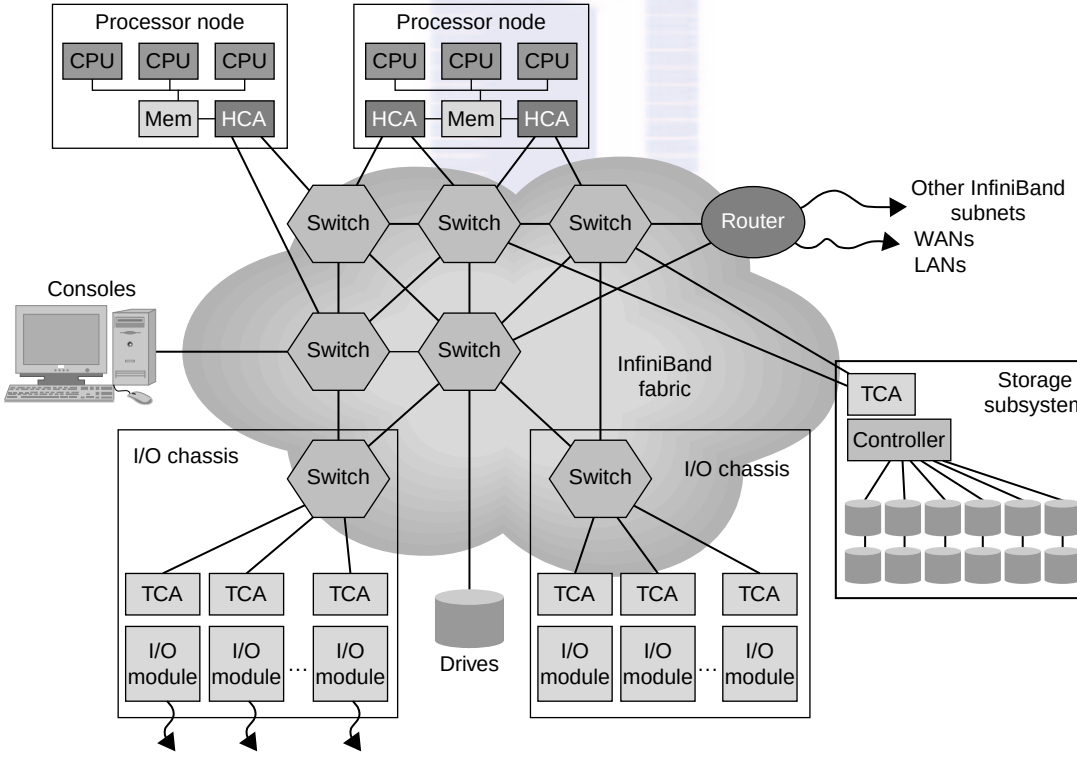


FIGURE 2.9

The InfiniBand system fabric built in a typical high-performance computer cluster.

(*Courtesy of Celebioglu, et al. [8]*)

2.2.4 Hardware, Software, and Middleware Support

Realistically, SSI and HA features in a cluster are not obtained free of charge. They must be supported by hardware, software, middleware, or OS extensions. Any change in hardware design and OS extensions must be done by the manufacturer. The hardware and OS support could be cost-prohibitive to ordinary users. However, programming level is a big burden to cluster users. Therefore, the middleware support at the application level costs the least to implement. As an example, we show in Figure 2.10 the middleware, OS extensions, and hardware support needed to achieve HA in a typical Linux cluster system.

Close to the user application end, middleware packages are needed at the cluster management level: one for fault management to support *failover* and *failback*, to be discussed in Section 2.3.3. Another desired feature is to achieve HA using failure detection and recovery and packet switching. In the middle of Figure 2.10, we need to modify the Linux OS to support HA, and we need special drivers to support HA, I/O, and hardware devices. Toward the bottom, we need special hardware to support hot-swapped devices and provide router interfaces. We will discuss various supporting mechanisms in subsequent sections.

2.2.5 GPU Clusters for Massive Parallelism

Commodity GPUs are becoming high-performance accelerators for data-parallel computing. Modern GPU chips contain hundreds of processor cores per chip. Based on a 2010 report [19], each GPU

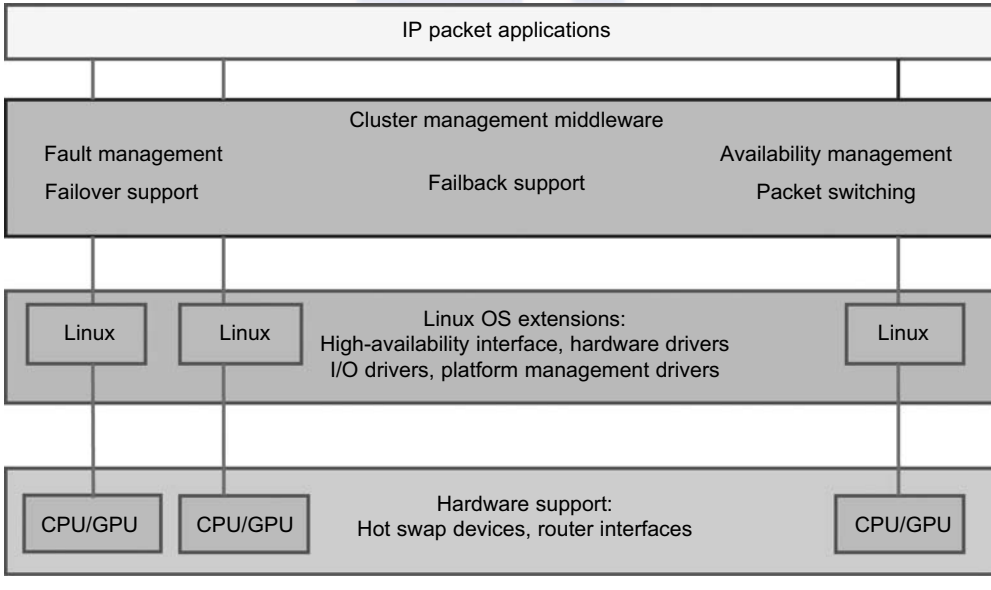


FIGURE 2.10

Middleware, Linux extensions, and hardware support for achieving massive parallelism and HA in a Linux cluster system built with CPUs and GPUs.

chip is capable of achieving up to 1 Tflops for single-precision (SP) arithmetic, and more than 80 Gflops for double-precision (DP) calculations. Recent HPC-optimized GPUs contain up to 4 GB of on-board memory, and are capable of sustaining memory bandwidths exceeding 100 GB/second. GPU clusters are built with a large number of GPU chips. GPU clusters have already demonstrated their capability to achieve Pflops performance in some of the Top 500 systems. Most GPU clusters are structured with homogeneous GPUs of the same hardware class, make, and model. The software used in a GPU cluster includes the OS, GPU drivers, and clustering API such as an MPI.

The high performance of a GPU cluster is attributed mainly to its massively parallel multicore architecture, high throughput in multithreaded floating-point arithmetic, and significantly reduced time in massive data movement using large on-chip cache memory. In other words, GPU clusters already are more cost-effective than traditional CPU clusters. GPU clusters result in not only a quantum jump in speed performance, but also significantly reduced space, power, and cooling demands. A GPU cluster can operate with a reduced number of operating system images, compared with CPU-based clusters. These reductions in power, environment, and management complexity make GPU clusters very attractive for use in future HPC applications.

2.2.5.1 The Echelon GPU Chip Design

Figure 2.11 shows the architecture of a future GPU accelerator that was suggested for use in building a NVIDIA Echelon GPU cluster for Exascale computing. This Echelong project led by Bill Dally at NVIDIA is partially funded by DARPA under the Ubiquitous High-Performance Computing (UHPC) program. This GPU design incorporates 1024 stream cores and 8 latency-optimized CPU-like cores (called latency processor) on a single chip. Eight stream cores form a *stream multiprocessor* (SM) and there are 128 SMs in the Echelon GPU chip.

Each SM is designed with 8 processor cores to yield a 160 Gflops peak speed. With 128 SMs, the chip has a peak speed of 20.48 Tflops. These nodes are interconnected by a NoC (network on chip) to 1,024 SRAM banks (L2 caches). Each cache band has a 256 KB capacity. The MCs (*memory controllers*) are used to connect to off-chip DRAMs and the NI (*network interface*) is to scale the size of the GPU cluster hierarchically, as shown in Figure 2.14. At the time of this writing, the Echelon is only a research project. With permission from Bill Dally, we present the design for

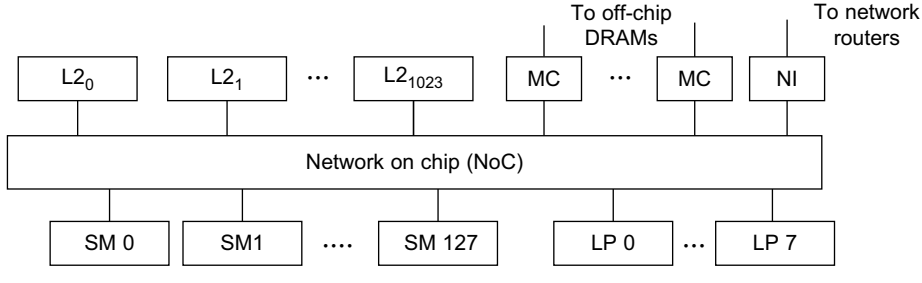


FIGURE 2.11

The proposed GPU chip design for 20 Tflops performance and 1.6 TB/s memory bandwidth in the Echelon system.

(*Courtesy of Bill Dally, Reprinted with Permission [10]*)

academic interest to illustrate how one can explore the many-core GPU technology to achieve Exascale computing in the future GPU technology to achieve Exascale computing in the future.

2.2.5.2 GPU Cluster Components

A GPU cluster is often built as a heterogeneous system consisting of three major components: the CPU host nodes, the GPU nodes and the cluster interconnect between them. The GPU nodes are formed with general-purpose GPUs, known as GPGPUs, to carry out numerical calculations. The host node controls program execution. The cluster interconnect handles inter-node communications. To guarantee the performance, multiple GPUs must be fully supplied with data streams over high-bandwidth network and memory. Host memory should be optimized to match with the on-chip cache bandwidths on the GPUs. Figure 2.12 shows the proposed Echelon GPU clusters using the GPU chips shown in Figure 2.11 as building blocks interconnected by a hierarchically constructed network.

2.2.5.3 Echelon GPU Cluster Architecture

The Echelon system architecture is shown in Figure 2.11, hierarchically. The entire Echelon system is built with *N* cabinets, labeled C0, C1, ... , C*N*. Each cabinet is built with 16 compute module labeled as M0, M1, ... , M15. Each compute module is built with 8 GPU nodes labeled as N0, N1, ... , N7. Each GPU node is the innermost block labeled as PC in Figure 2.12 (also detailed in Figure 2.11).

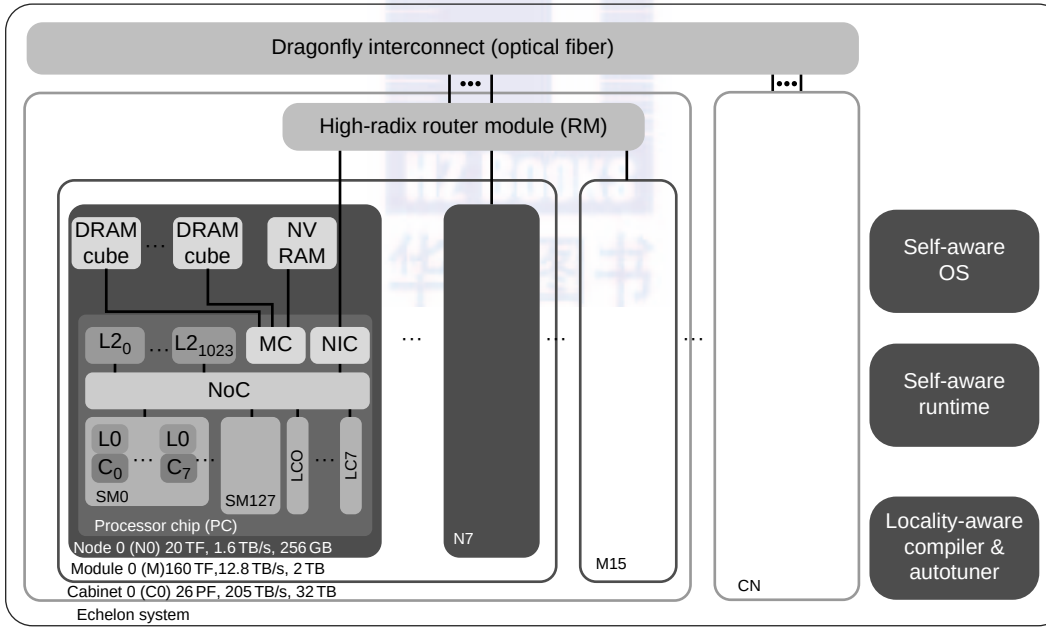


FIGURE 2.12

The architecture of NVIDIA Echelon system built with a hierarchical network of GPUs that can deliver 2.6 Pflops per cabinet, and takes at least N = 400 cabinets to achieve the desired Eflops performance.

(*Courtesy of Bill Dally, Reprinted with Permission [10]*)

Each compute module features a performance of 160 Tflops and 12.8 TB/s over 2 TB of memory. Thus, a single cabinet can house 128 GPU nodes or 16,000 processor cores. Each cabinet has the potential to deliver 2.6 Pflops over 32 TB memory and 205 TB/s bandwidth. The *N* cabinets are interconnected by a Dragonfly network with optical fiber.

To achieve Eflops performance, we need to use at least $N = 400$ cabinets. In total, an Exascale system needs 327,680 processor cores in 400 cabinets. The Echelon system is supported by a self-aware OS and runtime system. The Echelon system is also designed to preserve locality with the support of compiler and autotuner. At present, NVIDIA Fermi (GF110) chip has 512 stream processors. Thus the Echelon design is about 25 times faster. It is highly likely that the Echelon will employ post-Maxwell NVIDIA GPU planned to appear in 2013 ~ 2014 time frame.

2.2.5.4 CUDA Parallel Programming

CUDA (Compute Unified Device Architecture) offers a parallel computing architecture developed by NVIDIA. CUDA is the computing engine in NVIDIA GPUs. This software is accessible to developers through standard programming languages. Programmers use C for CUDA C with NVIDIA extensions and certain restrictions. This CUDA C is compiled through a PathScale Open64 C compiler for parallel execution on a large number of GPU cores. Example 2.4 shows the advantage of using CUDA C in parallel processing.

Example 2.4 Parallel SAXPY Execution Using CUDA C Code on GPUs

SAXPY is a kernel operation frequently performed in matrix multiplication. It essentially performs repeated *multiply and add* operations to generate the dot product of two long vectors. The following `saxpy_serial` routine is written in standard C code. This code is only suitable for sequential execution on a single processor core.

```
Void saxpy_serial (int n, float a, float*x, float *
  { for (int i = 0; i < n; ++i), y[i] = a*x[i] + y[i] }
    // Invoke the serial SAXPY kernel
  saxpy_serial (n, 2.0, x, y);
```

The following `saxpy_parallel` routine is written in CUDA C code for parallel execution by 256 threads/block on many processing cores on the GPU chip. Note that `n` blocks are handled by `n` processing cores, where `n` could be on the order of hundreds of blocks.

```
_global__void saxpy_parallel (int n, float a, float*x, float *y)
  { Int i = blockIndex.x*blockDim.x + threadIndex.x; if (i < n) y [i] = a*x[i] + y[i] }
 // Invoke the parallel SAXPY kernel with 256 threads/block int nblocks = (n + 255)/256;
  saxpy_parallel <<< nblocks, 256 >>> (n, 2.0, x, y);
```

This is a good example of using CUDA C to exploit massive parallelism on a cluster of multicore and multithreaded processors using the CODA GPGPUs as building blocks.

2.2.5.5 CUDA Programming Interfaces

CUDA architecture shares a range of computational interfaces with two competitors: the *Khronos Group*'s *Open Computing Language* and Microsoft's *DirectCompute*. Third-party wrappers are also

available for using Python, Perl, FORTRAN, Java, Ruby, Lua, MATLAB, and IDL. CUDA has been used to accelerate nongraphical applications in computational biology, cryptography, and other fields by an order of magnitude or more. A good example is the BOINC distributed computing client. CUDA provides both a low-level API and a higher-level API. CUDA works with all NVIDIA GPUs from the G8X series onward, including the GeForce, Quadro, and Tesla lines. NVIDIA states that programs developed for the GeForce 8 series will also work without modification on all future NVIDIA video cards due to binary compatibility.

2.2.5.6 Trends in CUDA Usage

Tesla and Fermi are two generations of CUDA architecture released by NVIDIA in 2007 and 2010, respectively. The CUDA version 3.2 is used for using a single GPU module in 2010. A newer CUDA version 4.0 will allow multiple GPUs to address use an unified virtual address space of shared memory. The next NVIDIA GPUs will be Kepler-designed to support C++. The Fermi has eight times the peak double-precision floating-point performance of the Tesla GPU (5.8 Gflops/W versus 0.7 Gflops/W). Currently, the Fermi has up to 512 CUDA cores on 3 billion transistors.

Future applications of the CUDA GPUs and the Echelon system may include the following:

- The search for extraterrestrial intelligence (SETI@Home)
- Distributed calculations to predict the native conformation of proteins
- Medical analysis simulations based on CT and MRI scan images
- Physical simulations in fluid dynamics and environment statistics
- Accelerated 3D graphics, cryptography, compression, and interconversion of video file formats
- Building the *single-chip cloud computer* (SCC) through virtualization in many-core architecture.

2.3 DESIGN PRINCIPLES OF COMPUTER CLUSTERS

Clusters should be designed for scalability and availability. In this section, we will cover the design principles of SSI, HA, fault tolerance, and rollback recovery in general-purpose computers and clusters of cooperative computers.

2.3.1 Single-System Image Features

SSI does not mean a single copy of an operating system image residing in memory, as in an SMP or a workstation. Rather, it means the *illusion* of a single system, single control, symmetry, and transparency as characterized in the following list:

- **Single system** The entire cluster is viewed by users as one system that has multiple processors. The user could say, "Execute my application using five processors." This is different from a distributed system.
- **Single control** Logically, an end user or system user utilizes services from one place with a single interface. For instance, a user submits batch jobs to one set of queues; a system administrator configures all the hardware and software components of the cluster from one control point.

- **Symmetry** A user can use a cluster service from any node. In other words, all cluster services and functionalities are symmetric to all nodes and all users, except those protected by access rights.
- **Location-transparent** The user is not aware of the where abouts of the physical device that eventually provides a service. For instance, the user can use a tape drive attached to any cluster node as though it were physically attached to the local node.

The main motivation to have SSI is that it allows a cluster to be used, controlled, and maintained as a familiar workstation is. The word "single" in "single-system image" is sometimes synonymous with "global" or "central." For instance, a global file system means a single file hierarchy, which a user can access from any node. A single point of control allows an operator to monitor and configure the cluster system. Although there is an illusion of a single system, a cluster service or functionality is often realized in a distributed manner through the cooperation of multiple components. A main requirement (and advantage) of SSI techniques is that they provide both the performance benefits of distributed implementation and the usability benefits of a single image.

From the viewpoint of a process *P*, cluster nodes can be classified into three types. The *home node* of a process *P* is the node where *P* resided when it was created. The *local node* of a process *P* is the node where *P* currently resides. All other nodes are *remote nodes* to *P*. Cluster nodes can be configured to suit different needs. A *host node* serves user logins through Telnet, rlogin, or even FTP and HTTP. A *compute node* is one that performs computational jobs. An *I/O node* is one that serves file I/O requests. If a cluster has large shared disks and tape units, they are normally physically attached to I/O nodes.

There is one home node for each process, which is fixed throughout the life of the process. At any time, there is only one local node, which may or may not be the host node. The local node and remote nodes of a process may change when the process migrates. A node can be configured to provide multiple functionalities. For instance, a node can be designated as a host, an I/O node, and a compute node at the same time. The illusion of an SSI can be obtained at several layers, three of which are discussed in the following list. Note that these layers may overlap with one another.

- **Application software layer** Two examples are parallel web servers and various parallel databases. The user sees an SSI through the application and is not even aware that he is using a cluster. This approach demands the modification of workstation or SMP applications for clusters.
- **Hardware or kernel layer** Ideally, SSI should be provided by the operating system or by the hardware. Unfortunately, this is not a reality yet. Furthermore, it is extremely difficult to provide an SSI over heterogeneous clusters. With most hardware architectures and operating systems being proprietary, only the manufacturer can use this approach.
- **Middleware layer** The most viable approach is to construct an SSI layer just above the OS kernel. This approach is promising because it is platform-independent and does not require application modification. Many cluster job management systems have already adopted this approach.

Each computer in a cluster has its own operating system image. Thus, a cluster may display multiple system images due to the stand-alone operations of all participating node computers. Determining how to merge the multiple system images in a cluster is as difficult as regulating many individual personalities in a community to a single personality. With different degrees of resource sharing, multiple systems could be integrated to achieve SSI at various operational levels.

2.3.1.1 Single Entry Point

Single-system image (SSI) is a very rich concept, consisting of single entry point, single file hierarchy, single I/O space, single networking scheme, single control point, single job management system, single memory space, and single process space. The single entry point enables users to log in (e.g., through Telnet, rlogin, or HTTP) to a cluster as one virtual host, although the cluster may have multiple physical host nodes to serve the login sessions. The system transparently distributes the user's login and connection requests to different physical hosts to balance the load. Clusters could substitute for mainframes and supercomputers. Also, in an Internet cluster server, thousands of HTTP or FTP requests may come simultaneously. Establishing a single entry point with multiple hosts is not a trivial matter. Many issues must be resolved. The following is just a partial list:

- **Home directory** Where do you put the user's home directory?
- **Authentication** How do you authenticate user logins?
- **Multiple connections** What if the same user opens several sessions to the same user account?
- **Host failure** How do you deal with the failure of one or more hosts?

Example 2.5 Realizing a Single Entry Point in a Cluster of Computers

Figure 2.13 illustrates how to realize a single entry point. Four nodes of a cluster are used as host nodes to receive users' login requests. Although only one user is shown, thousands of users can connect to the cluster in the same fashion. When a user logs into the cluster, he issues a standard UNIX command such as *telnet* cluster.cs.hku.hk, using the symbolic name of the cluster system.

The DNS translates the symbolic name and returns the IP address 159.226.41.150 of the least-loaded node, which happens to be node Host1. The user then logs in using this IP address. The DNS periodically receives load information from the host nodes to make load-balancing translation decisions. In the ideal case, if 200 users simultaneously log in, the login sessions are evenly distributed among our hosts with 50 users each. This allows a single host to be four times more powerful.

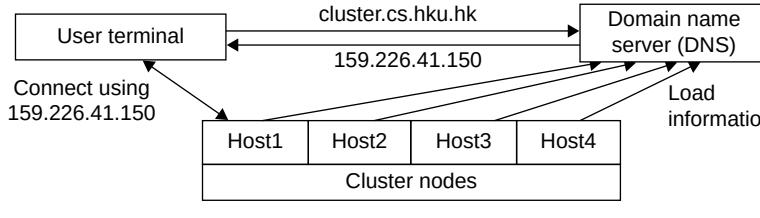


FIGURE 2.13

Realizing a single entry point using a load-balancing domain name system (DNS).

(*Courtesy of Hwang and Xu [14]*)

2.3.1.2 Single File Hierarchy

We use the term "single file hierarchy" in this book to mean the illusion of a single, huge file system image that transparently integrates local and global disks and other file devices (e.g., tapes). In other words, all files a user needs are stored in some subdirectories of the root directory /, and they can be accessed through ordinary UNIX calls such as *open*, *read*, and so on. This should not be confused with the fact that multiple file systems can exist in a workstation as subdirectories of the root directory.

The functionalities of a single file hierarchy have already been partially provided by existing distributed file systems such as *Network File System (NFS)* and *Andrew File System (AFS)*. From the viewpoint of any process, files can reside on three types of locations in a cluster, as shown in Figure 2.14.

Local storage is the disk on the local node of a process. The disks on remote nodes are *remote storage*. A *stable storage* requires two aspects: It is *persistent*, which means data, once written to the stable storage, will stay there for a sufficiently long time (e.g., a week), even after the cluster shuts down; and it is fault-tolerant to some degree, by using redundancy and periodic backup to tapes.

Figure 2.14 uses stable storage. Files in stable storage are called *global files*, those in local storage *local files*, and those in remote storage *remote files*. Stable storage could be implemented as one centralized, large RAID disk. But it could also be distributed using local disks of cluster nodes. The first approach uses a large disk, which is a single point of failure and a potential performance bottleneck. The latter approach is more difficult to implement, but it is potentially more economical, more efficient, and more available. On many cluster systems, it is customary for the system to make visible to the user processes the following directories in a single file hierarchy: the usual *system directories* as in a traditional UNIX workstation, such as /usr and /usr/local; and the user's *home directory* ~/ that has a small disk quota (1–20 MB). The user stores his code files and other files here. But large data files must be stored elsewhere.

- A *global directory* is shared by all users and all processes. This directory has a large disk space of multiple gigabytes. Users can store their large data files here.
- On a cluster system, a process can access a special directory on the local disk. This directory has medium capacity and is faster to access than the global directory.

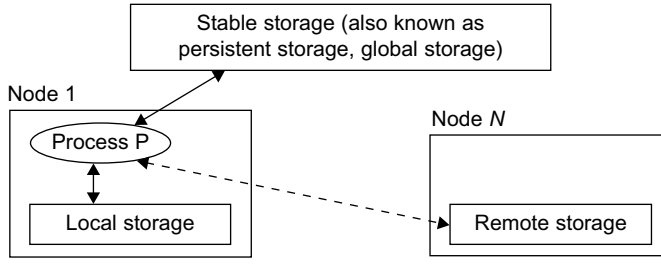


FIGURE 2.14

Three types of storage in a single file hierarchy. Solid lines show what process P can access and the dashed line shows what P may be able to access.

(*Courtesy of Hwang and Xu [14]*)

2.3.1.3 Visibility of Files

The term "visibility" here means a process can use traditional UNIX system or library calls such as *fopen*, *fread*, and *fwrite* to access files. Note that there are multiple local scratch directories in a cluster. The local scratch directories in remote nodes are not in the single file hierarchy, and are not directly visible to the process. A user process can still access them with commands such as *rcp* or some special library functions, by specifying both the node name and the filename.

The name "scratch" indicates that the storage is meant to act as a scratch pad for temporary information storage. Information in the local scratch space could be lost once the user logs out. Files in the global scratch space will normally persist even after the user logs out, but will be deleted by the system if not accessed in a predetermined time period. This is to free disk space for other users. The length of the period can be set by the system administrator, and usually ranges from one day to several weeks. Some systems back up the global scratch space to tapes periodically or before deleting any files.

2.3.1.4 Support of Single-File Hierarchy

It is desired that a single file hierarchy have the SSI properties discussed, which are reiterated for file systems as follows:

- **Single system** There is just one file hierarchy from the user's viewpoint.
- **Symmetry** A user can access the global storage (e.g., */scratch*) using a cluster service from any node. In other words, all file services and functionalities are symmetric to all nodes and all users, except those protected by access rights.
- **Location-transparent** The user is not aware of the whereabouts of the physical device that eventually provides a service. For instance, the user can use a RAID attached to any cluster node as though it were physically attached to the local node. There may be some performance differences, though.

A cluster file system should maintain *UNIX semantics*: Every file operation (*fopen*, *fread*, *fwrite*, *fclose*, etc.) is a transaction. When an *fread* accesses a file after an *fwrite* modifies the same file, the *fread* should get the updated value. However, existing distributed file systems do not completely follow UNIX semantics. Some of them update a file only at close or flush. A number of alternatives have been suggested to organize the global storage in a cluster. One extreme is to use a single file server that hosts a big RAID. This solution is simple and can be easily implemented with current software (e.g., NFS). But the file server becomes both a performance bottleneck and a single point of failure. Another extreme is to utilize the local disks in all nodes to form global storage. This could solve the performance and availability problems of a single file server.

2.3.1.5 Single I/O, Networking, and Memory Space

To achieve SSI, we desire a single control point, a single address space, a single job management system, a single user interface, and a single process control, as depicted in Figure 2.15. In this example, each node has exactly one network connection. Two of the four nodes each have two I/O devices attached.

Single Networking: A properly designed cluster should behave as one system (the shaded area). In other words, it is like a big workstation with four network connections and four I/O devices

attached. Any process on any node can use any network and I/O device as though it were attached to the local node. Single networking means any node can access any network connection.

Single Point of Control: The system administrator should be able to configure, monitor, test, and control the entire cluster and each individual node from a single point. Many clusters help with this through a system console that is connected to all nodes of the cluster. The system console is normally connected to an external LAN (not shown in Figure 2.15) so that the administrator can log in remotely to the system console from anywhere in the LAN to perform administration work.

Note that single point of control does not mean all system administration work should be carried out solely by the system console. In reality, many administrative functions are distributed across the cluster. It means that controlling a cluster should be no more difficult than administering an SMP or a mainframe. It implies that administration-related system information (such as various configuration files) should be kept in one logical place. The administrator monitors the cluster with one graphics tool, which shows the entire picture of the cluster, and the administrator can zoom in and out at will.

Single point of control (or *single point of management*) is one of the most challenging issues in constructing a cluster system. Techniques from distributed and networked system management can be transferred to clusters. Several de facto standards have already been developed for network management. An example is *Simple Network Management Protocol (SNMP)*. It demands an efficient cluster management package that integrates with the availability support system, the file system, and the job management system.

Single Memory Space: *Single memory space* gives users the illusion of a big, centralized main memory, which in reality may be a set of distributed local memory spaces. PVPs, SMPs, and DSMs have an edge over MPPs and clusters in this respect, because they allow a program to utilize all global or local memory space. A good way to test if a cluster has a single memory space is to run a *sequential* program that needs a memory space larger than any single node can provide.

Suppose each node in Figure 2.15 has 2 GB of memory available to users. An ideal single memory image would allow the cluster to execute a sequential program that needs 8 GB of memory. This would enable a cluster to operate like an SMP system. Several approaches have

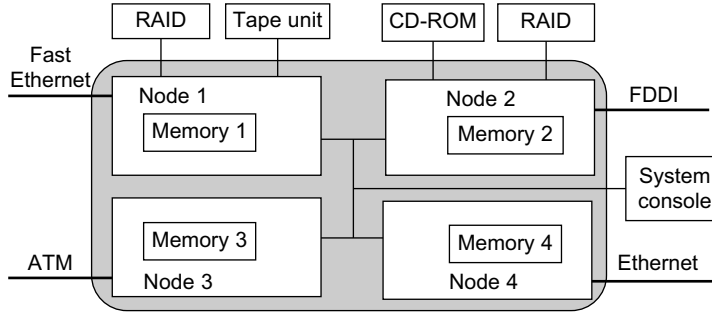


FIGURE 2.15

A cluster with single networking, single I/O space, single memory, and single point of control.

(*Courtesy of Hwang and Xu [14]*)

been attempted to achieve a single memory space on clusters. Another approach is to let the compiler distribute the data structures of an application across multiple nodes. It is still a challenging task to develop a single memory scheme that is efficient, platform-independent, and able to support sequential binary codes.

Single I/O Address Space: Assume the cluster is used as a web server. The web information database is distributed between the two RAIDs. An HTTP daemon is started on each node to handle web requests, which come from all four network connections. A single I/O space implies that any node can access the two RAIDs. Suppose most requests come from the ATM network. It would be beneficial if the functions of the HTTP on node 3 could be distributed to all four nodes. The following example shows a distributed RAID-x architecture for I/O-centric cluster computing [9].

Example 2.6 Single I/O Space over Distributed RAID for I/O-Centric Clusters

A distributed disk array architecture was proposed by Hwang, et al. [9] for establishing a single I/O space in I/O-centric cluster applications. Figure 2.16 shows the architecture for a four-node Linux PC cluster, in which three disks are attached to the SCSI bus of each host node. All 12 disks form an integrated RAID-x with a single address space. In other words, all PCs can access both local and remote disks. The addressing scheme for all disk blocks is interleaved horizontally. Orthogonal stripping and mirroring make it possible to have a RAID-1 equivalent capability in the system.

The shaded blocks are images of the blank blocks. A disk block and its image will be mapped on different physical disks in an orthogonal manner. For example, the block B_0 is located on disk D0. The image block M_o of block B_0 is located on disk D3. The four disks D0, D1, D2, and D3 are attached to four servers, and thus can be accessed in parallel. Any single disk failure will not lose the data block, because its image is available in recovery. All disk blocks are labeled to show image mapping. Benchmark experiments show that this RAID-x is scalable and can restore data after any single disk failure. The distributed RAID-x has improved aggregate I/O bandwidth in both parallel read and write operations over all physical disks in the cluster.

2.3.1.6 Other Desired SSI Features

The ultimate goal of SSI is for a cluster to be as easy to use as a desktop computer. Here are additional types of SSI, which are present in SMP servers:

- **Single job management system** All cluster jobs can be submitted from any node to a single job management system.
- **Single user interface** The users use the cluster through a single graphical interface. Such an interface is available for workstations and PCs. A good direction to take in developing a cluster GUI is to utilize web technology.
- **Single process space** All user processes created on various nodes form a single process space and share a uniform process identification scheme. A process on any node can create (e.g., through a UNIX fork) or communicate with (e.g., through signals, pipes, etc.) processes on remote nodes.

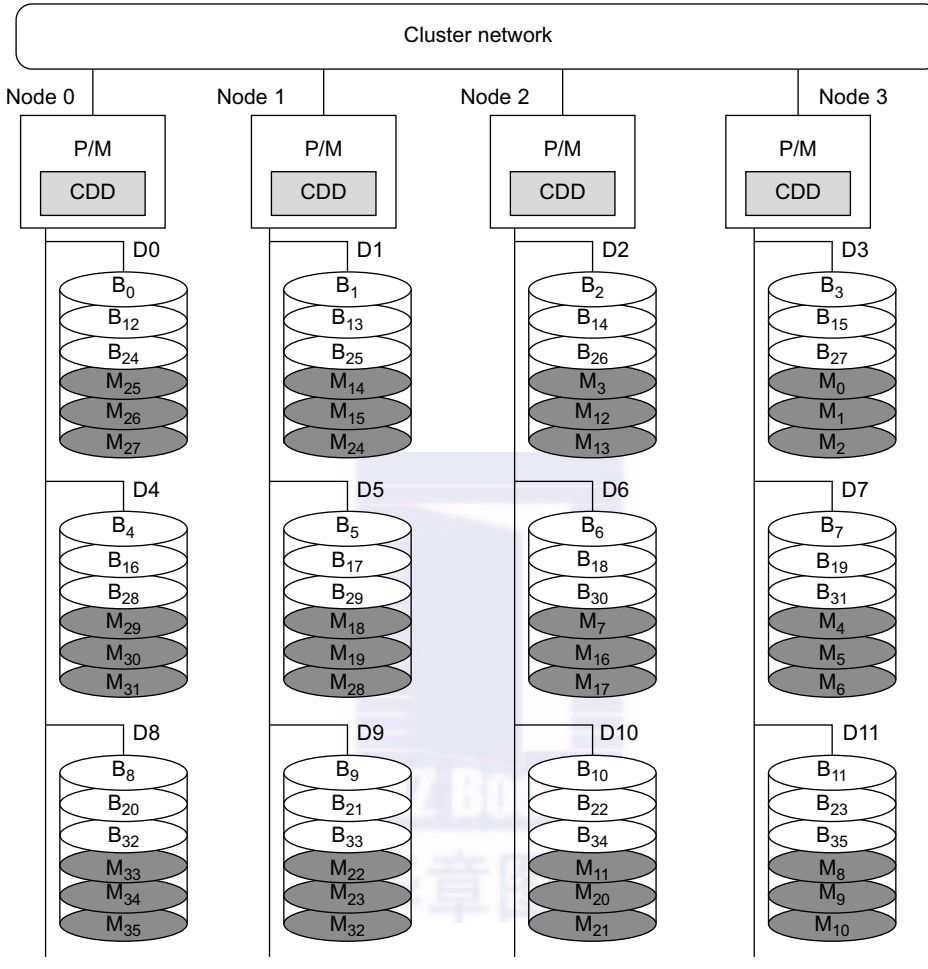


FIGURE 2.16

Distributed RAID architecture with a single I/O space over 12 distributed disks attached to 4 host computers in the cluster (Di stands for Disk i, B_j for disk block j, M_j an image for blocks B_j, P/M for processor/memory node, and CDD for cooperative disk driver.)

(*Courtesy of Hwang, Jin, and Ho [13]*)

- **Middleware support for SSI clustering** As shown in Figure 2.17, various SSI features are supported by middleware developed at three cluster application levels:
- **Management level** This level handles user applications and provides a job management system such as GLUnix, MOSIX, *Load Sharing Facility (LSF)*, or Codine.
- **Programming level** This level provides single file hierarchy (NFS, xFS, AFS, Proxy) and distributed shared memory (TreadMark, Wind Tunnel).

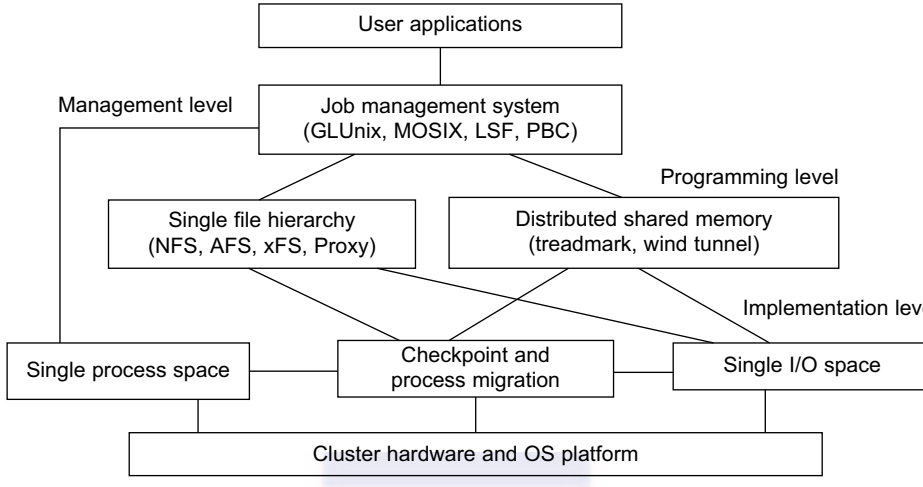


FIGURE 2.17

Relationship among clustering middleware at the job management, programming, and implementation levels.
(*Courtesy of K. Hwang, H. Jin, C.L. Wang and Z. Xu [16]*)

- **Implementation level** This level supports a single process space, checkpointing, process migration, and a single I/O space. These features must interface with the cluster hardware and OS platform. The distributed disk array, RAID-x, in Example 2.6 implements a single I/O space.

2.3.2 High Availability through Redundancy

When designing robust, highly available systems three terms are often used together: *reliability*, *availability*, and *serviceability (RAS)*. Availability is the most interesting measure since it combines the concepts of reliability and serviceability as defined here:

- *Reliability* measures how long a system can operate without a breakdown.
- *Availability* indicates the percentage of time that a system is available to the user, that is, the percentage of system uptime.
- *Serviceability* refers to how easy it is to service the system, including hardware and software maintenance, repair, upgrades, and so on.

The demand for RAS is driven by practical market needs. A recent Find/SVP survey found the following figures among Fortune 1000 companies: An average computer is down nine times per year with an average downtime of four hours. The average loss of revenue per hour of downtime is $82,500. With such a hefty penalty for downtime, many companies are striving for systems that offer *24/365 availability*, meaning the system is available 24 hours per day, 365 days per year.

2.3.2.1 Availability and Failure Rate

As Figure 2.18 shows, a computer system operates normally for a period of time before it fails. The failed system is then repaired, and the system returns to normal operation. This operate-repair cycle

then repeats. A system's reliability is measured by the *mean time to failure* (MTTF), which is the average time of normal operation before the system (or a component of the system) fails. The metric for serviceability is the *mean time to repair* (MTTR), which is the average time it takes to repair the system and restore it to working condition after it fails. The *availability* of a system is defined by:

$$Availability = \text{MTTF}/(\text{MTTF} + \text{MTTR}) \tag{2.1}$$

2.3.2.2 Planned versus Unplanned Failure

When studying RAS, we call any event that prevents the system from normal operation a *failure*. This includes:

- **Unplanned failures** The system breaks, due to an operating system crash, a hardware failure, a network disconnection, human operation errors, a power outage, and so on. All these are simply called failures. The system must be repaired to correct the failure.
- **Planned shutdowns** The system is not broken, but is periodically taken off normal operation for upgrades, reconfiguration, and maintenance. A system may also be shut down for weekends or holidays. The MTTR in Figure 2.18 for this type of failure is the planned downtime.

Table 2.5 shows the availability values of several representative systems. For instance, a conventional workstation has an availability of 99 percent, meaning it is up and running 99 percent of the time or it has a downtime of 3.6 days per year. An optimistic definition of availability does not consider planned downtime, which may be significant. For instance, many supercomputer installations have a planned downtime of several hours per week, while a telephone system cannot tolerate a downtime of a few minutes per year.

2.3.2.3 Transient versus Permanent Failures

A lot of failures are *transient* in that they occur temporarily and then disappear. They can be dealt with without replacing any components. A standard approach is to roll back the system to a known

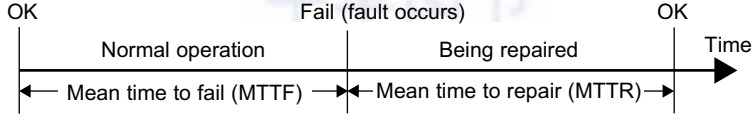


FIGURE 2.18

The operate-repair cycle of a computer system.

Table 2.5 Availability of Computer System Types

System Type	Availability (%)	Downtime in a Year
Conventional workstation	99	3.6 days
HA system	99.9	8.5 hours
Fault-resilient system	99.99	1 hour
Fault-tolerant system	99.999	5 minutes

state and start over. For instance, we all have rebooted our PC to take care of transient failures such as a frozen keyboard or window. *Permanent failures* cannot be corrected by rebooting. Some hardware or software component must be repaired or replaced. For instance, rebooting will not work if the system hard disk is broken.

2.3.2.4 Partial versus Total Failures

A failure that renders the entire system unusable is called a *total failure*. A failure that only affects part of the system is called a *partial failure* if the system is still usable, even at a reduced capacity. A key approach to enhancing availability is to make as many failures as possible partial failures, by systematically removing *single points of failure*, which are hardware or software components whose failure will bring down the entire system.

Example 2.7 Single Points of Failure in an SMP and in Clusters of Computers

In an SMP (Figure 2.19(a)), the shared memory, the OS image, and the memory bus are all single points of failure. On the other hand, the processors are not forming a single point of failure. In a cluster of workstations (Figure 2.19(b)), interconnected by Ethernet, there are multiple OS images, each residing in a workstation. This avoids the single point of failure caused by the OS as in the SMP case. However, the Ethernet network now becomes a single point of failure, which is eliminated in Figure 2.17(c), where a high-speed network is added to provide two paths for communication.

When a node fails in the clusters in Figure 2.19(b) and Figure 2.19(c), not only will the node applications all fail, but also all node data cannot be used until the node is repaired. The shared disk cluster in Figure 2.19(d) provides a remedy. The system stores persistent data on the shared disk, and periodically *checkpoints* to save intermediate results. When one WS node fails, the data will not be lost in this shared-disk cluster.

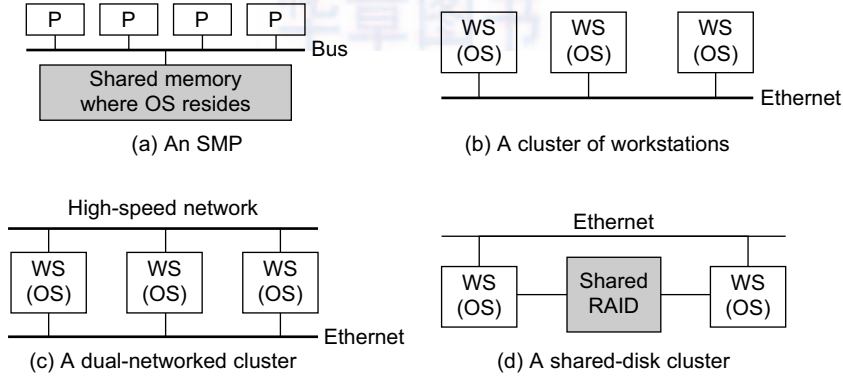


FIGURE 2.19

Single points of failure (SPF) in an SMP and in three clusters, where greater redundancy eliminates more SPFs in systems from (a) to (d).

(*Courtesy of Hwang and Xu [14]*)

2.3.2.5 Redundancy Techniques

Consider the cluster in Figure 2.19(d). Assume only the nodes can fail. The rest of the system (e.g., interconnect and the shared RAID disk) is 100 percent available. Also assume that when a node fails, its workload is switched over to the other node in zero time. We ask, what is the availability of the cluster if planned downtime is ignored? What is the availability if the cluster needs one hour/week for maintenance? What is the availability if it is shut down one hour/week, one node at a time?

According to Table 2.4, a workstation is available 99 percent of the time. The time both nodes are down is only 0.01 percent. Thus, the availability is 99.99 percent. It is now a fault-resilient system, with only one hour of downtime per year. The planned downtime is 52 hours per year, that is, $52 / (365 \times 24) = 0.0059$. The total downtime is now 0.59 percent + 0.01 percent = 0.6 percent. The availability of the cluster becomes 99.4 percent. Suppose we ignore the unlikely situation in which the other node fails while one node is maintained. Then the availability is 99.99 percent.

There are basically two ways to increase the availability of a system: increasing MTTF or reducing MTTR. Increasing MTTF amounts to increasing the reliability of the system. The computer industry has strived to make reliable systems, and today's workstations have MTTFs in the range of hundreds to thousands of hours. However, to further improve MTTF is very difficult and costly. Clusters offer an HA solution based on reducing the MTTR of the system. A multinode cluster has a lower MTTF (thus lower reliability) than a workstation. However, the failures are taken care of quickly to deliver higher availability. We consider several redundancy techniques used in cluster design.

2.3.2.6 Isolated Redundancy

A key technique to improve availability in any system is to use redundant components. When a component (the *primary* component) fails, the service it provided is taken over by another component (the *backup* component). Furthermore, the primary and the backup components should be *isolated* from each other, meaning they should not be subject to the same cause of failure. Clusters provide HA with redundancy in power supplies, fans, processors, memories, disks, I/O devices, networks, operating system images, and so on. In a carefully designed cluster, redundancy is also isolated. Isolated redundancy provides several benefits:

- First, a component designed with isolated redundancy is not a single point of failure, and the failure of that component will not cause a total system failure.
- Second, the failed component can be repaired while the rest of the system is still working.
- Third, the primary and the backup components can mutually test and debug each other.

The IBM SP2 communication subsystem is a good example of isolated-redundancy design. All nodes are connected by two networks: an Ethernet network and a high-performance switch. Each node uses two separate interface cards to connect to these networks. There are two communication protocols: a standard IP and a *user-space* (US) protocol; each can run on either network. If either network or protocol fails, the other network or protocol can take over.

2.3.2.7 N-Version Programming to Enhance Software Reliability

A common isolated-redundancy approach to constructing a mission-critical software system is called *N-version programming*. The software is implemented by N isolated teams who may not even know the others exist. Different teams are asked to implement the software using different algorithms, programming languages, environment tools, and even platforms. In a fault-tolerant system, the

N versions all run simultaneously and their results are constantly compared. If the results differ, the system is notified that a fault has occurred. But because of isolated redundancy, it is extremely unlikely that the fault will cause a majority of the *N* versions to fail at the same time. So the system continues working, with the correct result generated by majority voting. In a highly available but less mission-critical system, only one version needs to run at a time. Each version has a built-in self-test capability. When one version fails, another version can take over.

2.3.3 Fault-Tolerant Cluster Configurations

The cluster solution was targeted to provide availability support for two server nodes with three ascending levels of availability: *hot standby*, *active takeover*, and *fault-tolerant*. In this section, we will consider the *recovery time*, *failback feature*, and *node activeness*. The level of availability increases from standby to active and fault-tolerant cluster configurations. The shorter is the recovery time, the higher is the cluster availability. *Failback* refers to the ability of a recovered node returning to normal operation after repair or maintenance. *Activeness* refers to whether the node is used in active work during normal operation.

- **Hot standby server clusters** In a *hot standby* cluster, only the primary node is actively doing all the useful work normally. The standby node is powered on (hot) and running some monitoring programs to communicate heartbeat signals to check the status of the primary node, but is not actively running other useful workloads. The primary node must mirror any data to shared disk storage, which is accessible by the standby node. The standby node requires a second copy of data.
- **Active-takeover clusters** In this case, the architecture is symmetric among multiple server nodes. Both servers are primary, doing useful work normally. Both failover and failback are often supported on both server nodes. When a node fails, the user applications fail over to the available node in the cluster. Depending on the time required to implement the failover, users may experience some delays or may lose some data that was not saved in the last checkpoint.
- **Failover cluster** This is probably the most important feature demanded in current clusters for commercial applications. When a component fails, this technique allows the remaining system to take over the services originally provided by the failed component. A failover mechanism must provide several functions, such as *failure diagnosis*, *failure notification*, and *failure recovery*. Failure diagnosis refers to the detection of a failure and the location of the failed component that caused the failure. A commonly used technique is *heartbeat*, whereby the cluster nodes send out a stream of heartbeat messages to one another. If the system does not receive the stream of heartbeat messages from a node, it can conclude that either the node or the network connection has failed.

Example 2.8 Failure Diagnosis and Recovery in a Dual-Network Cluster

A cluster uses two networks to connect its nodes. One node is designated as the *master node*. Each node has a *heartbeat daemon* that periodically (every 10 seconds) sends a heartbeat message to the master

node through both networks. The master node will detect a failure if it does not receive messages for a beat (10 seconds) from a node and will make the following diagnoses:

- A node's connection to one of the two networks failed if the master receives a heartbeat from the node through one network but not the other.
- The node failed if the master does not receive a heartbeat through either network. It is assumed that the chance of both networks failing at the same time is negligible.

The failure diagnosis in this example is simple, but it has several pitfalls. What if the master node fails? Is the 10-second heartbeat period too long or too short? What if the heartbeat messages are dropped by the network (e.g., due to network congestion)? Can this scheme accommodate hundreds of nodes? Practical HA systems must address these issues. A popular trick is to use the heartbeat messages to carry load information so that when the master receives the heartbeat from a node, it knows not only that the node is alive, but also the resource utilization status of the node. Such load information is useful for load balancing and job management.

Once a failure is diagnosed, the system notifies the components that need to know the failure event. Failure notification is needed because the master node is not the only one that needs to have this information. For instance, in case of the failure of a node, the DNS needs to be told so that it will not connect more users to that node. The resource manager needs to reassign the workload and to take over the remaining workload on that node. The system administrator needs to be alerted so that she can initiate proper actions to repair the node.

2.3.3.1 Recovery Schemes

Failure recovery refers to the actions needed to take over the workload of a failed component. There are two types of recovery techniques. In *backward recovery*, the processes running on a cluster periodically save a consistent state (called a *checkpoint*) to a stable storage. After a failure, the system is reconfigured to isolate the failed component, restores the previous checkpoint, and resumes normal operation. This is called *rollback*.

Backward recovery is relatively easy to implement in an application-independent, portable fashion, and has been widely used. However, rollback implies wasted execution. If execution time is crucial, such as in real-time systems where the rollback time cannot be tolerated, a *forward recovery* scheme should be used. With such a scheme, the system is not rolled back to the previous checkpoint upon a failure. Instead, the system utilizes the failure diagnosis information to reconstruct a valid system state and continues execution. Forward recovery is application-dependent and may need extra hardware.

Example 2.9 MTTF, MTTR, and Failure Cost Analysis

Consider a cluster that has little availability support. Upon a node failure, the following sequence of events takes place:

1. The entire system is shut down and powered off.
2. The faulty node is replaced if the failure is in hardware.
3. The system is powered on and rebooted.
4. The user application is reloaded and rerun from the start.

Assume one of the cluster nodes fails every 100 hours. Other parts of the cluster never fail. Steps 1 through 3 take two hours. On average, the mean time for step 4 is two hours. What is the availability of the cluster? What is the yearly failure cost if each one-hour downtime costs $82,500?

Solution: The cluster's MTTF is 100 hours; the MTTR is 2 + 2 = 4 hours. According to Table 2.5, the availability is 100/104 = 96.15 percent. This corresponds to 337 hours of downtime in a year, and the failure cost is $82500 × 337, that is, more than $27 million.

Example 2.10 Availability and Cost Analysis of a Cluster of Computers

Repeat Example 2.9, but assume that the cluster now has much increased availability support. Upon a node failure, its workload automatically fails over to other nodes. The failover time is only six minutes. Meanwhile, the cluster has *hot swap* capability: The faulty node is taken off the cluster, repaired, replugged, and rebooted, and it rejoins the cluster, all without impacting the rest of the cluster. What is the availability of this ideal cluster, and what is the yearly failure cost?

Solution: The cluster's MTTF is still 100 hours, but the MTTR is reduced to 0.1 hours, as the cluster is available while the failed node is being repaired. From to Table 2.5, the availability is 100/100.5 = 99.9 percent. This corresponds to 8.75 hours of downtime per year, and the failure cost is $82,500, a 27M/722K = 38 times reduction in failure cost from the design in Example 3.8.

2.3.4 Checkpointing and Recovery Techniques

Checkpointing and recovery are two techniques that must be developed hand in hand to enhance the availability of a cluster system. We will start with the basic concept of checkpointing. This is the process of periodically saving the state of an executing program to stable storage, from which the system can recover after a failure. Each program state saved is called a *checkpoint*. The disk file that contains the saved state is called the *checkpoint file*. Although all current checkpointing software saves program states in a disk, research is underway to use node memories in place of stable storage in order to improve performance.

Checkpointing techniques are useful not only for availability, but also for program debugging, process migration, and load balancing. Many job management systems and some operating systems support checkpointing to a certain degree. The Web Resource contains pointers to numerous checkpoint-related web sites, including some public domain software such as Condor and Libckpt. Here we will present the important issues for the designer and the user of checkpoint software. We will first consider the issues that are common to both sequential and parallel programs, and then we will discuss the issues pertaining to parallel programs.

2.3.4.1 Kernel, Library, and Application Levels

Checkpointing can be realized by the operating system at the *kernel level*, where the OS transparently checkpoints and restarts processes. This is ideal for users. However, checkpointing is not supported in most operating systems, especially for parallel programs. A less transparent approach links the user code with a checkpointing library in the user space. Checkpointing and restarting are

handled by this runtime support. This approach is used widely because it has the advantage that user applications do not have to be modified.

A main problem is that most current checkpointing libraries are static, meaning the application source code (or at least the object code) must be available. It does not work if the application is in the form of executable code. A third approach requires the user (or the compiler) to insert checkpointing functions in the application; thus, the application has to be modified, and the transparency is lost. However, it has the advantage that the user can specify where to checkpoint. This is helpful to reduce checkpointing overhead. Checkpointing incurs both time and storage overheads.

2.3.4.2 Checkpoint Overheads

During a program's execution, its states may be saved many times. This is denoted by the time consumed to save one checkpoint. The storage overhead is the extra memory and disk space required for checkpointing. Both time and storage overheads depend on the size of the checkpoint file. The overheads can be substantial, especially for applications that require a large memory space. A number of techniques have been suggested to reduce these overheads.

2.3.4.3 Choosing an Optimal Checkpoint Interval

The time period between two checkpoints is called the *checkpoint interval*. Making the interval larger can reduce checkpoint time overhead. However, this implies a longer computation time after a failure. Wong and Franklin [28] derived an expression for optimal checkpoint interval as illustrated in Figure 2.20.

$$\textit{Optimal checkpoint interval} = \text{Square root}\,(MTTF \times t_c)/h \tag{2.2}$$

Here, MTTF is the system's *mean time to failure*. This MTTF accounts the time consumed to save one checkpoint, and h is the average percentage of normal computation performed in a checkpoint interval before the system fails. The parameter h is always in the range. After a system is restored, it needs to spend $h \times$ (checkpoint interval) time to recompute.

2.3.4.4 Incremental Checkpoint

Instead of saving the full state at each checkpoint, an *incremental checkpoint* scheme saves only the portion of the state that is changed from the previous checkpoint. However, care must be taken regarding *old* checkpoint files. In full-state checkpointing, only one checkpoint file needs to be kept on disk. Subsequent checkpoints simply overwrite this file. With incremental checkpointing, old files needed to be kept, because a state may span many files. Thus, the total storage requirement is larger.

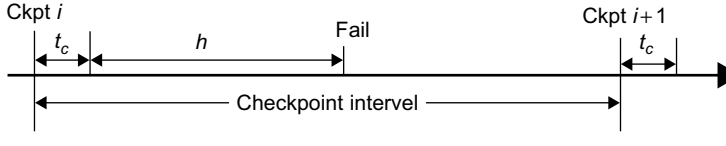


FIGURE 2.20

Time parameters between two checkpoints.

(*Courtesy of Hwang and Xu [14]*)

2.3.4.5 Forked Checkpointing

Most checkpoint schemes are blocking in that the normal computation is stopped while checkpointing is in progress. With enough memory, checkpoint overhead can be reduced by making a copy of the program state in memory and invoking another asynchronous thread to perform the checkpointing concurrently. A simple way to overlap checkpointing with computation is to use the UNIX *fork( )* system call. The forked child process duplicates the parent process's address space and checkpoints it. Meanwhile, the parent process continues execution. Overlapping is achieved since checkpointing is disk-I/O intensive. A further optimization is to use the copy-on-write mechanism.

2.3.4.6 User-Directed Checkpointing

The checkpoint overheads can sometimes be substantially reduced if the user inserts code (e.g., library or system calls) to tell the system when to save, what to save, and what not to save. What should be the exact contents of a checkpoint? It should contain just enough information to allow a system to recover. The state of a process includes its data state and control state. For a UNIX process, these states are stored in its address space, including the text (code), the data, the stack segments, and the process descriptor. Saving and restoring the full state is expensive and sometimes impossible.

For instance, the process ID and the parent process ID are not restorable, nor do they need to be saved in many applications. Most checkpointing systems save a partial state. For instance, the code segment is usually not saved, as it does not change in most applications. What kinds of applications can be checkpointed? Current checkpoint schemes require programs to be *well behaved*, the exact meaning of which differs in different schemes. At a minimum, a well-behaved program should not need the exact contents of state information that is not restorable, such as the numeric value of a process ID.

2.3.4.7 Checkpointing Parallel Programs

We now turn to checkpointing parallel programs. The state of a parallel program is usually much larger than that of a sequential program, as it consists of the set of the states of individual processes, plus the state of the communication network. Parallelism also introduces various timing and consistency problems.

Example 2.11 Checkpointing a Parallel Program

Figure 2.21 illustrates checkpointing of a three-process parallel program. The arrows labeled *x*, *y*, and *z* represent point-to-point communication among the processes. The three thick lines labeled *a*, *b*, and *c* represent three *global snapshots* (or simply *snapshots*), where a global snapshot is a set of checkpoints

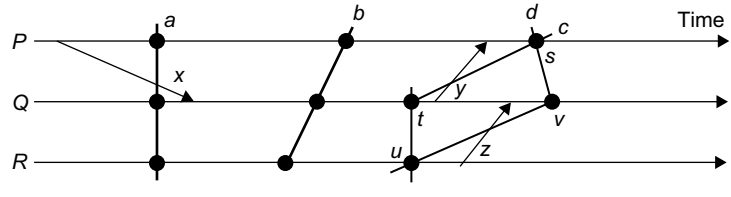


FIGURE 2.21

Consistent and inconsistent checkpoints in a parallel program.

(*Courtesy of Hwang and Xu [14]*)

(represented by dots), one from every process. In addition, some communication states may need to be saved. The intersection of a snapshot line with a process's time line indicates where the process should take a (local) checkpoint. Thus, the program's snapshot *c* consists of three local checkpoints: *s*, *t*, *u* for processes *P*, *Q*, and *R*, respectively, plus saving the communication *y*.

2.3.4.8 Consistent Snapshot

A global snapshot is called *consistent* if there is no message that is received by the checkpoint of one process, but not yet sent by another process. Graphically, this corresponds to the case that no arrow crosses a snapshot line from right to left. Thus, snapshot *a* is consistent, because arrow *x* is from left to right. But snapshot *c* is inconsistent, as *y* goes from right to left. To be consistent, there should not be any *zigzag path* between two checkpoints [20]. For instance, checkpoints *u* and *s* cannot belong to a consistent global snapshot. A stronger consistency requires that no arrows cross the snapshot. Thus, only snapshot *b* is consistent in Figure 2.23.

2.3.4.9 Coordinated versus Independent Checkpointing

Checkpointing schemes for parallel programs can be classified into two types. In *coordinated checkpointing* (also called *consistent checkpointing*), the parallel program is frozen, and all processes are checkpointed at the same time. In *independent checkpointing*, the processes are checkpointed independent of one another. These two types can be combined in various ways. Coordinated checkpointing is difficult to implement and tends to incur a large overhead. Independent checkpointing has a small overhead and can utilize existing checkpointing schemes for sequential programs.

2.4 CLUSTER JOB AND RESOURCE MANAGEMENT

This section covers various scheduling methods for executing multiple jobs on a clustered system. The LSF is described as middleware for cluster computing. MOSIX is introduced as a distributed OS for managing resources in large-scale clusters or in clouds.

2.4.1 Cluster Job Scheduling Methods

Cluster jobs may be scheduled to run at a specific time (*calendar scheduling*) or when a particular event happens (*event scheduling*). Table 2.6 summarizes various schemes to resolve job scheduling issues on a cluster. Jobs are scheduled according to priorities based on submission time, resource nodes, execution time, memory, disk, job type, and user identity. With *static priority*, jobs are assigned priorities according to a predetermined, fixed scheme. A simple scheme is to schedule jobs in a first-come, first-serve fashion. Another scheme is to assign different priorities to users. With *dynamic priority*, the priority of a job may change over time.

Three schemes are used to share cluster nodes. In the *dedicated mode*, only one job runs in the cluster at a time, and at most, one process of the job is assigned to a node at a time. The single job runs until completion before it releases the cluster to run other jobs. Note that even in the dedicated mode, some nodes may be reserved for system use and not be open to the user job. Other than that, all cluster resources are devoted to run a single job. This may lead to poor system utilization. The job resource

Table 2.6 Job Scheduling Issues and Schemes for Cluster Nodes

Issue	Scheme	Key Problems
Job priority	Nonpreemptive	Delay of high-priority jobs
	Preemptive	Overhead, implementation
Resource required	Static	Load imbalance
	Dynamic	Overhead, implementation
Resource sharing	Dedicated	Poor utilization
	Space sharing	Tiling, large job
Scheduling	Time sharing	Process-based job control with context switch overhead
	Independent	Severe slowdown
	Gang scheduling	Implementation difficulty
Competing with foreign (local) jobs	Stay	Local job slowdown
	Migrate	Migration threshold, migration overhead

requirement can be *static* or *dynamic*. Static scheme fixes the number of nodes for a single job for its entire period. Static scheme may underutilize the cluster resource. It cannot handle the situation when the needed nodes become unavailable, such as when the workstation owner shuts down the machine.

Dynamic resource allows a job to acquire or release nodes during execution. However, it is much more difficult to implement, requiring cooperation between a running job and the Java Message Service (JMS). The jobs make asynchronous requests to the JMS to add/delete resources. The JMS needs to notify the job when resources become available. The synchrony means that a job should not be delayed (blocked) by the request/notification. Cooperation between jobs and the JMS requires modification of the programming languages/libraries. A primitive mechanism for such cooperation exists in PVM and MPI.

2.4.1.1 Space Sharing

A common scheme is to assign higher priorities to short, interactive jobs in daytime and during evening hours using *tiling*. In this *space-sharing* mode, multiple jobs can run on disjointed partitions (groups) of nodes simultaneously. At most, one process is assigned to a node at a time. Although a partition of nodes is dedicated to a job, the interconnect and the I/O subsystem may be shared by all jobs. Space sharing must solve the tiling problem and the large-job problem.

Example 2.12 Job Scheduling by Tiling over Cluster Nodes

Figure 2.22 illustrates the *tiling technique*. In Part (a), the JMS schedules four jobs in a first-come first-serve fashion on four nodes. Jobs 1 and 2 are small and thus assigned to nodes 1 and 2. Jobs 3 and 4 are parallel; each needs three nodes. When job 3 comes, it cannot run immediately. It must wait until job 2 finishes to free up the needed nodes. Tiling will increase the utilization of the nodes as shown in Figure 2.22(b). The overall execution time of the four jobs is reduced after repacking the jobs over the available nodes. This problem cannot be solved in dedicated or space-sharing modes. However, it can be alleviated by timesharing.

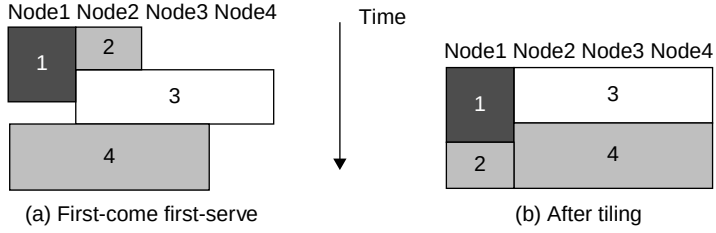


FIGURE 2.22

The tiling technique for scheduling more jobs to cluster nodes to shorten the total makespan and thus increase the job throughput.

(*Courtesy of Hwang and Xu [14]*)

2.4.1.2 Time Sharing

In the dedicated or space-sharing model, only one user process is allocated to a node. However, the system processes or daemons are still running on the same node. In the *time-sharing* mode, multiple user processes are assigned to the same node. Time sharing introduces the following parallel scheduling policies:

1. **Independent scheduling** The most straightforward implementation of time sharing is to use the operating system of each cluster node to schedule different processes as in a traditional workstation. This is called *local scheduling* or *independent scheduling*. However, the performance of parallel jobs could be significantly degraded. Processes of a parallel job need to interact. For instance, when one process wants to barrier-synchronize with another, the latter may be scheduled out. So the first process has to wait. As the second process is rescheduled, the first process may be swapped out.
2. **Gang scheduling** The *gang scheduling* scheme schedules all processes of a parallel job together. When one process is active, all processes are active. The cluster nodes are not perfectly clock-synchronized. In fact, most clusters are asynchronous systems, and are not driven by the same clock. Although we say, "All processes are scheduled to run at the same time," they do not start exactly at the same time. *Gang-scheduling skew* is the maximum difference between the time the first process starts and the time the last process starts. The execution time of a parallel job increases as the gang-scheduling skew becomes larger, leading to longer execution time. We should use a homogeneous cluster, where gang scheduling is more effective. However, gang scheduling is not yet realized in most clusters, because of implementation difficulties.
3. **Competition with foreign (local) jobs** Scheduling becomes more complicated when both cluster jobs and local jobs are running. Local jobs should have priority over cluster jobs. With one keystroke, the owner wants command of all workstation resources. There are basically two ways to deal with this situation: The cluster job can either stay in the workstation node or migrate to another idle node. A *stay scheme* has the advantage of avoiding migration cost. The cluster process can be run at the lowest priority. The workstation's cycles can be divided into three portions, for kernel processes, local processes, and cluster processes. However, to stay slows down both the local and the cluster jobs, especially when the cluster job is a load-balanced parallel job that needs frequent synchronization and communication. This leads to the migration approach to flow the jobs around available nodes, mainly for balancing the workload.

2.4.2 Cluster Job Management Systems

Job management is also known as *workload management*, *load sharing*, or *load management*. We will first discuss the basic issues facing a job management system and summarize the available software packages. A *Job Management System (JMS)* should have three parts:

- A *user server* lets the user submit jobs to one or more queues, specify resource requirements for each job, delete a job from a queue, and inquire about the status of a job or a queue.
- A *job scheduler* performs job scheduling and queuing according to job types, resource requirements, resource availability, and scheduling policies.
- A *resource manager* allocates and monitors resources, enforces scheduling policies, and collects accounting information.

2.4.2.1 JMS Administration

The functionality of a JMS is often distributed. For instance, a user server may reside in each host node, and the resource manager may span all cluster nodes. However, the administration of a JMS should be *centralized*. All configuration and log files should be maintained in one location. There should be a single user interface to use the JMS. It is undesirable to force the user to run PVM jobs through one software package, MPI jobs through another, and HPF jobs through yet another.

The JMS should be able to dynamically reconfigure the cluster with minimal impact on the running jobs. The administrator's prologue and epilogue scripts should be able to run before and after each job for security checking, accounting, and cleanup. Users should be able to cleanly kill their own jobs. The administrator or the JMS should be able to cleanly suspend or kill any job. *Clean* means that when a job is suspended or killed, all its processes must be included. Otherwise, some "orphan" processes are left in the system, which wastes cluster resources and may eventually render the system unusable.

2.4.2.2 Cluster Job Types

Several types of jobs execute on a cluster. *Serial jobs* run on a single node. *Parallel jobs* use multiple nodes. *Interactive jobs* are those that require fast turnaround time, and their input/output is directed to a terminal. These jobs do not need large resources, and users expect them to execute immediately, not to wait in a queue. *Batch jobs* normally need more resources, such as large memory space and long CPU time. But they do not need immediate responses. They are submitted to a job queue to be scheduled to run when the resource becomes available (e.g., during off hours).

While both interactive and batch jobs are managed by the JMS, *foreign jobs* are created outside the JMS. For instance, when a network of workstations is used as a cluster, users can submit interactive or batch jobs to the JMS. Meanwhile, the owner of a workstation can start a foreign job at any time, which is not submitted through the JMS. Such a job is also called a *local job*, as opposed to *cluster jobs* (interactive or batch, parallel or serial) that are submitted through the JMS of the cluster. The characteristic of a local job is fast response time. The owner wants all resources to execute his job, as though the cluster jobs did not exist.

2.4.2.3 Characteristics of a Cluster Workload

To realistically address job management issues, we must understand the workload behavior of clusters. It may seem ideal to characterize workload based on long-time operation data on real clusters. The parallel workload traces include both development and production jobs. These traces are then

fed to a simulator to generate various statistical and performance results, based on different sequential and parallel workload combinations, resource allocations, and scheduling policies. The following workload characteristics are based on a NAS benchmark experiment. Of course, different workloads may have variable statistics.

- Roughly half of parallel jobs are submitted during regular working hours. Almost 80 percent of parallel jobs run for three minutes or less. Parallel jobs running longer than 90 minutes account for 50 percent of the total time.
- The sequential workload shows that 60 percent to 70 percent of workstations are available to execute parallel jobs at any time, even during peak daytime hours.
- On a workstation, 53 percent of all idle periods are three minutes or less, but 95 percent of idle time is spent in periods of time that are 10 minutes or longer.
- A 2:1 rule applies, which says that a network of 64 workstations, with proper JMS software, can sustain a 32-node parallel workload in addition to the original sequential workload. In other words, clustering gives a supercomputer half of the cluster size for free!

2.4.2.4 Migration Schemes

A *migration scheme* must consider the following three issues:

- **Node availability** This refers to node availability for job migration. The Berkeley NOW project has reported such opportunity does exist in university campus environment. Even during peak hours, 60 percent of workstations in a cluster are available at Berkeley campus.
- **Migration overhead** What is the effect of the migration overhead? The migration time can significantly slow down a parallel job. It is important to reduce the migration overhead (e.g., by improving the communication subsystem) or to migrate only rarely. The slowdown is significantly reduced if a parallel job is run on a cluster of twice the size. For instance, for a 32-node parallel job run on a 60-node cluster, the slowdown caused by migration is no more than 20 percent, even when the migration time is as long as three minutes. This is because more nodes are available, and thus the migration demand is less frequent.
- **Recruitment threshold** What should be the recruitment threshold? In the worst scenario, right after a process migrates to a node, the node is immediately claimed by its owner. Thus, the process has to migrate again, and the cycle continues. The recruitment threshold is the amount of time a workstation stays unused before the cluster considers it an idle node.

2.4.2.5 Desired Features in A JMS

Here are some features that have been built in some commercial JMSes in cluster computing applications:

- Most support heterogeneous Linux clusters. All support parallel and batch jobs. However, Connect:Queue does not support interactive jobs.
- Enterprise cluster jobs are managed by the JMS. They will impact the owner of a workstation in running local jobs. However, NQE and PBS allow the impact to be adjusted. In DQS, the impact can be configured to be minimal.
- All packages offer some kind of load-balancing mechanism to efficiently utilize cluster resources. Some packages support checkpointing.
- Most packages cannot support dynamic process migration. They support static migration: A process can be dispatched to execute on a remote node when the process is first created.

However, once it starts execution, it stays in that node. A package that does support dynamic process migration is Condor.

- All packages allow dynamic suspension and resumption of a user job by the user or by the administrator. All packages allow resources (e.g., nodes) to be dynamically added to or deleted.
- Most packages provide both a command-line interface and a graphical user interface. Besides UNIX security mechanisms, most packages use the Kerberos authentication system.

2.4.3 Load Sharing Facility (LSF) for Cluster Computing

LSF is a commercial workload management system from Platform Computing [29]. LSF emphasizes job management and load sharing on both parallel and sequential jobs. In addition, it supports checkpointing, availability, load migration, and SSI. LSF is highly scalable and can support a cluster of thousands of nodes. LSF has been implemented for various UNIX and Windows/NT platforms. Currently, LSF is being used not only in clusters but also in grids and clouds.

2.4.3.1 LSF Architecture

LSF supports most UNIX platforms and uses the standard IP for JMS communication. Because of this, it can convert a heterogeneous network of UNIX computers into a cluster. There is no need to change the underlying OS kernel. The end user utilizes the LSF functionalities through a set of utility commands. PVM and MPI are supported. Both a command-line interface and a GUI are provided. LSF also offers skilled users an API that is a runtime library called *LSLIB (load sharing library)*. Using LSLIB explicitly requires the user to modify the application code, whereas using the utility commands does not. Two LSF daemons are used on each server in the cluster. The *load information managers (LIMs)* periodically exchange load information. The *remote execution server (RES)* executes remote tasks.

2.4.3.2 LSF Utility Commands

A cluster node may be a single-processor host or an SMP node with multiple processors, but always runs with only a single copy of the operating system on the node. Here are interesting features built into the LSF facilities:

- LSF supports all four combinations of interactive, batch, sequential, and parallel jobs. A job that is not executed through LSF is called a *foreign job*. A *server node* is one which can execute LSF jobs. A *client node* is one that can initiate or submit LSF jobs but cannot execute them. Only the resources on the server nodes can be shared. Server nodes can also initiate or submit LSF jobs.
- LSF offers a set of tools (*lstools*) to get information from LSF and to run jobs remotely. For instance, *lshosts* lists the static resources (discussed shortly) of every server node in the cluster. The command *lsrun* executes a program on a remote node.
- When a user types the command line *%lsrun-R 'swp>100' myjob* at a client node, the application *myjob* will be automatically executed on the most lightly loaded server node that has an available swap space greater than 100 MB.
- The *lsbatch* utility allows users to submit, monitor, and execute batch jobs through LSF. This utility is a load-sharing version of the popular UNIX command interpreter *tcsh*. Once a user enters the *lstcsh* shell, every command issued will be automatically executed on a suitable node. This is done transparently: The user sees a shell exactly like a *tcsh* running on the local node.

- The *lsmake* utility is a parallel version of the UNIX make utility, allowing a makefile to be processed in multiple nodes simultaneously.

Example 2.13 Application of the LSF on a Cluster of Computers

Suppose a cluster consists of eight expensive server nodes and 100 inexpensive client nodes (workstations or PCs). The server nodes are expensive due to better hardware and software, including application software. A license is available to install a FORTRAN compiler and a CAD simulation package, both valid for up to four users. Using a JMS such as LSF, all the hardware and software resources of the server nodes are made available to the clients transparently.

A user sitting in front of a client's terminal feels as though the client node has all the software and speed of the servers locally. By typing *lsmake my.makefile*, the user can compile his source code on up to four servers. LSF selects the nodes with the least amount of load. Using LSF also benefits resource utilization. For instance, a user wanting to run a CAD simulation can submit a batch job. LSF will schedule the job as soon as the software becomes available.

2.4.4 MOSIX: An OS for Linux Clusters and Clouds

MOSIX is a distributed operating system that was developed at Hebrew University in 1977. Initially, the system extended BSD/OS system calls for resource sharing in Pentium clusters. In 1999, the system was redesigned to run on Linux clusters built with x86 platforms. The MOSIX project is still active as of 2011, with 10 versions released over the years. The latest version, MOSIX2, is compatible with Linux 2.6.

2.4.4.1 MOXIS2 for Linux Clusters

MOSIX2 runs as a virtualization layer in the Linux environment. This layer provides SSI to users and applications along with runtime Linux support. The system runs applications in remote nodes as though they were run locally. It supports both sequential and parallel applications, and can discover resources and migrate software processes transparently and automatically among Linux nodes. MOSIX2 can also manage a Linux cluster or a grid of multiple clusters.

Flexible management of a grid allows owners of clusters to share their computational resources among multiple cluster owners. Each cluster can still preserve its autonomy over its own clusters and its ability to disconnect its nodes from the grid at any time. This can be done without disrupting the running programs. A MOSIX-enabled grid can extend indefinitely as long as trust exists among the cluster owners. The condition is to guarantee that guest applications cannot be modified while running in remote clusters. Hostile computers are not allowed to connect to the local network.

2.4.4.2 SSI Features in MOSIX2

The system can run in *native mode* or as a VM. In native mode, the performance is better, but it requires that you modify the base Linux kernel, whereas a VM can run on top of any unmodified OS that supports virtualization, including Microsoft Windows, Linux, and Mac OS X. The system

is most suitable for running compute-intensive applications with low to moderate amounts of I/O. Tests of MOSIX2 show that the performance of several such applications over a 1 GB/second campus grid is nearly identical to that of a single cluster. Here are some interesting features of MOSIX2:

- Users can log in on any node and do not need to know where their programs run.
- There is no need to modify or link applications with special libraries.
- There is no need to copy files to remote nodes, thanks to automatic resource discovery and workload distribution by process migration.
- Users can load-balance and migrate processes from slower to faster nodes and from nodes that run out of free memory.
- Sockets are migratable for direct communication among migrated processes.
- The system features a secure runtime environment (sandbox) for guest processes.
- The system can run batch jobs with checkpoint recovery along with tools for automatic installation and configuration scripts.

2.4.4.3 Applications of MOSIX for HPC

The MOSIX is a research OS for HPC cluster, grid, and cloud computing. The system's designers claim that MOSIX offers efficient utilization of wide-area grid resources through automatic resource discovery and load balancing. The system can run applications with unpredictable resource requirements or runtimes by running long processes, which are automatically sent to grid nodes. The system can also combine nodes of different capacities by migrating processes among nodes based on their load index and available memory.

MOSIX became proprietary software in 2001. Application examples include scientific computations for genomic sequence analysis, molecular dynamics, quantum dynamics, nanotechnology and other parallel HPC applications; engineering applications including CFD, weather forecasting, crash simulations, oil industry simulations, ASIC design, and pharmaceutical design; and cloud applications such as for financial modeling, rendering farms, and compilation farms.

Example 2.14 Memory-Ushering Algorithm Using MOSIX versus PVM

Memory ushering is practiced to borrow the main memory of a remote cluster node, when the main memory on a local node is exhausted. The remote memory access is done by process migration instead of paging or swapping to local disks. The ushering process can be implemented with PVM commands or it can use MOSIX process migration. In each execution, an average memory chunk can be assigned to the nodes using PVM. Figure 2.23 shows the execution time of the memory algorithm using PVM compared with the use of the MOSIX routine.

For a small cluster of eight nodes, the execution times are closer. When the cluster scales to 32 nodes, the MOSIX routine shows a 60 percent reduction in ushering time. Furthermore, MOSIX performs almost the same when the cluster size increases. The PVM ushering time increases monotonically by an average of 3.8 percent per node increase, while MOSIX consistently decreases by 0.4 percent per node increase. The reduction in time results from the fact that the memory and load-balancing algorithms of MOSIX are more scalable than PVM.

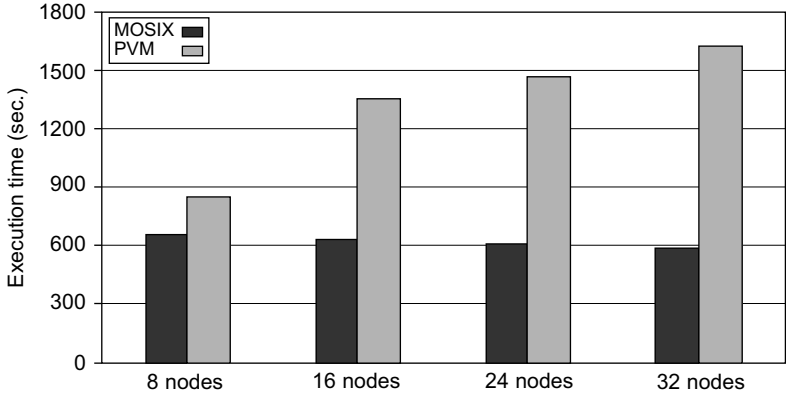


FIGURE 2.23

Performance of the memory-ushering algorithm using MOSIX versus PVM.

(*Courtesy of A. Barak and O. La'adan [5]*)

2.5 CASE STUDIES OF TOP SUPERCOMPUTER SYSTEMS

This section reviews three top supercomputers that have been singled out as winners in the Top 500 List for the years 2008–2010. The IBM Roadrunner was the world's first petaflops computer, ranked No. 1 in 2008. Subsequently, the Cray XT5 Jaguar became the top system in 2009. In November 2010, China's Tianhe-1A became the fastest system in the world. All three systems are Linux cluster-structured with massive parallelism in term of large number of compute nodes that can execute concurrently.

2.5.1 Tianhe-1A: The World Fastest Supercomputer in 2010

In November 2010, the Tianhe-1A was unveiled as a hybrid supercomputer at the 2010 ACM Supercomputing Conference. This system demonstrated a sustained speed of 2.507 Pflops in Linpack Benchmark testing runs and thus became the No. 1 supercomputer in the 2010 Top 500 list. The system was built by the National University of Defense Technology (NUDT) and was installed in August 2010 at the National Supercomputer Center (NSC), Tianjin, in northern China (www.nscc.tj.gov.cn). The system is intended as an open platform for research and education. Figure 2.24 shows the Tianh-1A system installed at NSC.

2.5.1.1 Architecture of Tianhe-1A

Figure 2.25 shows the abstract architecture of the Tianhe-1A system. The system consists of five major components. The compute subsystem houses all the CPUs and GPUs on 7,168 compute nodes. The service subsystem comprises eight operation nodes. The storage subsystem has a large number of shared disks. The monitoring and diagnosis subsystem is used for control and I/O operations. The communication subsystem is composed of switches for connecting to all functional subsystems.

2.5.1.2 Hardware Implementation

This system is equipped with 14,336 six-core Xeon E5540/E5450 processors running 2.93 GHz with 7,168 NVIDIA Tesla M2050s. It has 7,168 *compute nodes*, each composed of two Intel Xeon X5670 (Westmere) processors at 2.93 GHz, six cores per socket, and one NVIDIA M2050 GPU

FIGURE 2.24

The Tianhe-1A system built by the National University of Defense Technology and installed at the National Supercomputer Center, Tianjin, China, in 2010 [11].

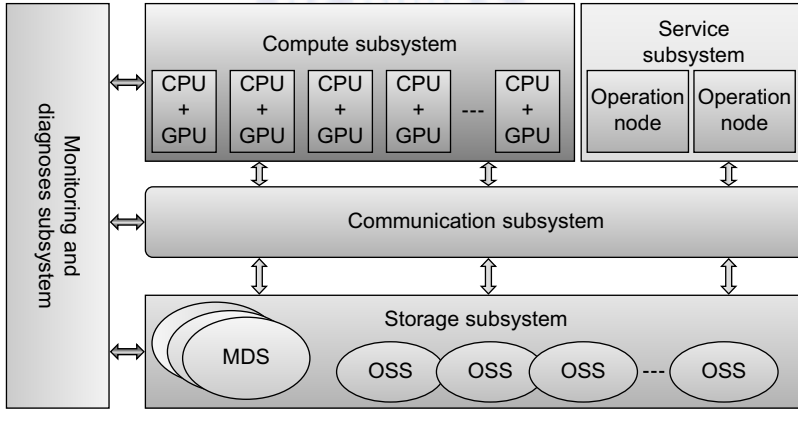


FIGURE 2.25

Abstract architecture of the Tianhe-1A system.

connected via PCI-E. A blade has two nodes and is 2U in height (Figure 2.25). The complete system has 14,336 Intel sockets (Westmere) plus 7,168 NVIDIA Fermi boards plus 2,048 Galaxy sockets (the Galaxy processor-based nodes are used as frontend processing for the system). A compute node has two Intel sockets plus a Fermi board plus 32 GB of memory.

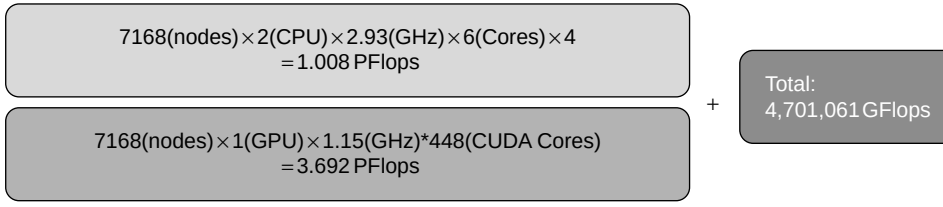


FIGURE 2.26

Calculation of the theoretical peak speed of Tianhe-1A system.

The total system has a theoretical peak of 4.7 Pflops/second as calculated in Figure 2.26. Note that there are 448 CUDA cores in each GPU node. The peak speed is achieved through 14,236 Xeon CPUs (with 380,064 cores) and 7,168 Tesla GPUs (with 448 CUDA cores per node and 3,496,884 CUDA cores in total). There are 3,876,948 processing cores in both the CPU and GPU chips. An operational node has two eight-core Galaxy chips (1 GHz, SPARC architecture) plus 32 GB of memory. The Tianhe-1A system is packaged in 112 compute cabinets, 12 storage cabinets, six communications cabinets, and eight I/O cabinets.

The operation nodes are composed of two eight-core Galaxy FT-1000 chips. These processors were designed by NUDT and run at 1 GHz. The theoretical peak for the eight-core chip is 8 Gflops/second. The complete system has 1,024 of these operational nodes with each having 32 GB of memory. These operational nodes are intended to function as service nodes for job creation and submission. They are not intended as general-purpose computational nodes. Their speed is excluded from the calculation of the peak or sustained speed. The peak speed of the Tianhe-1A is calculated as 3.692 Pflops [11]. It uses 7,168 compute nodes (with 448 CUDA cores/GPU/compute node) in parallel with 14,236 CPUs with six cores in four subsystems.

The system has total disk storage of 2 petabytes implemented with a Lustre clustered file system. There are 262 terabytes of main memory distributed in the cluster system. The Tianhe-1A epitomizes modern heterogeneous CPU/GPU computing, enabling significant achievements in performance, size, and power. The system would require more than 50,000 CPUs and twice as much floor space to deliver the same performance using CPUs alone. A 2.507-petaflop system built entirely with CPUs would consume at least 12 megawatts, which is three times more power than what the Tianhe-1A consumes.

2.5.1.3 ARCH Fat-Tree Interconnect

The high performance of the Tianhe-1A is attributed to a customed-designed ARCH interconnect by the NUDT builder. This ARCH is built with the InfiniBand DDR 4X and 98 TB of memory. It assumes a fat-tree architecture as shown in Figure 2.27. The bidirectional bandwidth is 160 Gbps, about twice the bandwidth of the QDR InfiniBand network over the same number of nodes. The ARCH has a latency for a node hop of 1.57 microseconds, and an aggregated bandwidth of 61 Tb/second. At the first stage of the ARCH fat tree, 16 nodes are connected by a 16-port switching board. At the second stage, all parts are connects to eleven 384-port switches. The router and network interface chips are designed by the NUDT team.

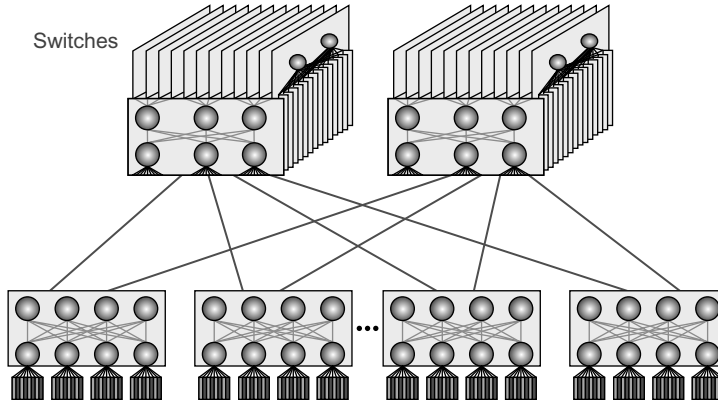


FIGURE 2.27

The ARCH fat-tree interconnect in two stages of high-bandwidth switches [11].

2.5.1.4 Software Stack

The software stack on the Tianhe-1A is typical of any high-performance system. It uses Kylin Linux, an operating system developed by NUDT and successfully approved by China's 863 Hi-tech Research and Development Program office in 2006. Kylin is based on Mach and FreeBSD, is compatible with other mainstream operating systems, and supports multiple microprocessors and computers of different structures. Kylin packages include standard open source and public packages, which have been brought onto one system for easy installation. Figure 2.28 depicts the Tianhe-1A software architecture.

The system features FORTRAN, C, C++, and Java compilers from Intel (icc 11.1), CUDA, OpenMP, and MPI based on MPICH2 with custom GLEX (Galaxy Express) Channel support. The NUDT builder developed a mathematics library, which is based on Intel's MKL 10.3.1.048 and BLAS for the GPU based on NVIDIA and optimized by NUDT. In addition, a High Productive Parallel Running Environment (HPPRE) was installed. This provides a parallel toolkit based on Eclipse, which is intended to integrate all the tools for editing, debugging, and performance analysis. In addition, the designers provide workflow support for Quality of Service (QoS) negotiations and resource reservations.

2.5.1.5 Power Consumption, Space, and Cost

The power consumption of the Tianhe-1A under load is 4.04 MWatt. The system has a footprint of 700 square meters and is cooled by a closed-coupled chilled water-cooling system with forced air. The hybrid architecture consumes less power—about one-third of the 12 MW that is needed to run the system entirely with the multicore CPUs. The budget for the system is 600 million RMB (approximately $90 million); 200 million RMB comes from the Ministry of Science and Technology (MOST) and 400 million RMB is from the Tianjin local government. It takes about $20 million annually to run, maintain, and keep the system cool in normal operations.

2.5.1.6 Linpack Benchmark Results and Planned Applications

The performance of the Linpack Benchmark on October 30, 2010 was 2.566 Pflops/second on a matrix of 3,600,000 and a $N_{1/2}$ = 1,000,000. The total time for the run was 3 hours and 22 minutes.

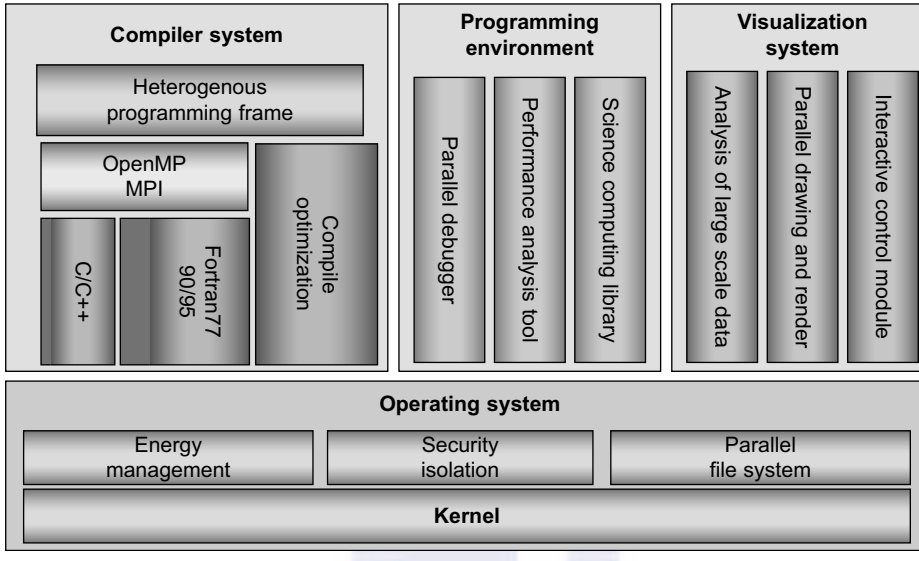


FIGURE 2.28

Software architecture of the Tianhe-1A supercomputer [11].

The system has an efficiency of 54.58 percent, which is much lower than the 75 percent efficiency achieved by Jaguar and Roadrunner. Listed below are some applications of Tianhe-1A. Most of them are specially tailored to satisfy China's national needs.

- Parallel AMR (Adaptive Mesh Refinement) method
- Parallel eigenvalue problems
- Parallel fast multipole methods
- Parallel computing models
- Gridmol computational chemistry
- ScGrid middleware, grid portal
- PSEPS parallel symmetric eigenvalue package solvers
- FMM-radar fast multipole methods on radar cross sections
- Transplant many open source software programs
- Sandstorm prediction, climate modeling, EM scattering, or cosmology
- CAD/CAE for automotive industry

2.5.2 Cray XT5 Jaguar: The Top Supercomputer in 2009

The Cray XT5 Jaguar was ranked the world's fastest supercomputer in the Top 500 list released at the ACM Supercomputing Conference in June 2010. This system became the second fastest supercomputer in the Top 500 list released in November 2010, when China's Tianhe-1A replaced the Jaguar as the No. 1 machine. This is a scalable MPP system built by Cray, Inc. The Jaguar belongs to Cray's system model XT5-HE. The system is installed at the Oak Ridge National Laboratory,

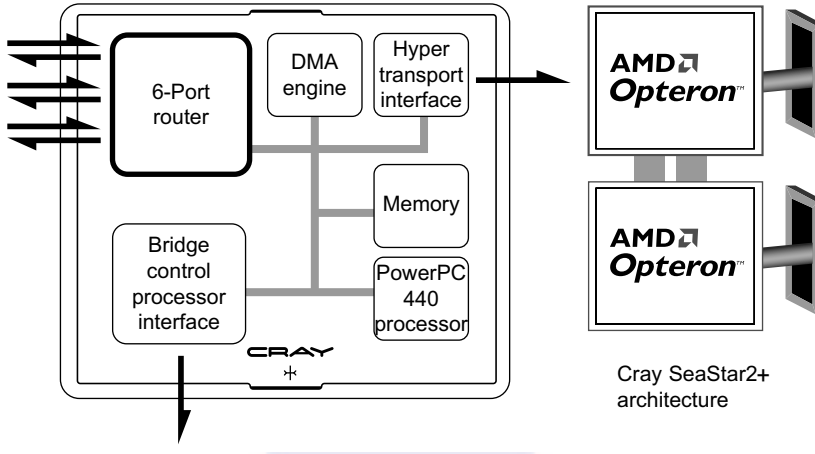


FIGURE 2.29

The interconnect SeaStar router chip design in the Cray XT5 Jaguar supercomputer.

(*Courtesy of Cray, Inc. [9] and Oak Ridge National Laboratory, United States, 2009*)

Department of Energy, in the United States. The entire Jaguar system is built with 86 cabinets. The following are some interesting architectural and operational features of the Jaguar system:

- Built with AMD six-core Opteron processors running Linux at a 2.6 GHz clock rate
- Has a total of 224,162 cores on more than 37,360 processors in 88 cabinets in four rows (there are 1,536 or 2,304 processor cores per cabinet)
- Features 8,256 compute nodes and 96 service nodes interconnected by a 3D torus network, built with Cray SeaStar2+ chips
- Attained a sustained speed, R_{max}, from the Linpack Benchmark test of 1.759 Pflops
- Largest Linpack matrix size tested recorded as N_{max} = 5,474,272 unknowns

The basic building blocks are the compute blades. The interconnect router in the SeaStar+ chip (Figure 2.29) provides six high-speed links to six neighbors in the 3D torus, as seen in Figure 2.30. The system is scalable by design from small to large configurations. The entire system has 129 TB of compute memory. In theory, the system was designed with a peak speed of $R_{peak} = 2.331$ Pflops. In other words, only 75 percent (=1.759/2.331) efficiency was achieved in Linpack experiments. The external I/O interface uses 10 Gbps Ethernet and InfiniBand links. MPI 2.1 was applied in message-passing programming. The system consumes 32–43 KW per cabinet. With 160 cabinets, the entire system consumes up to 6.950 MW. The system is cooled with forced cool air, which consumes a lot of electricity.

2.5.2.1 3D Torus Interconnect

Figure 2.30 shows the system's interconnect architecture. The Cray XT5 system incorporates a high-bandwidth, low-latency interconnect using the Cray SeaStar2+ router chips. The system is configured with XT5 compute blades with eight sockets supporting dual or quad-core Opterons. The XT5 applies a 3D torus network topology. This SeaStar2+ chip provides six high-speed network

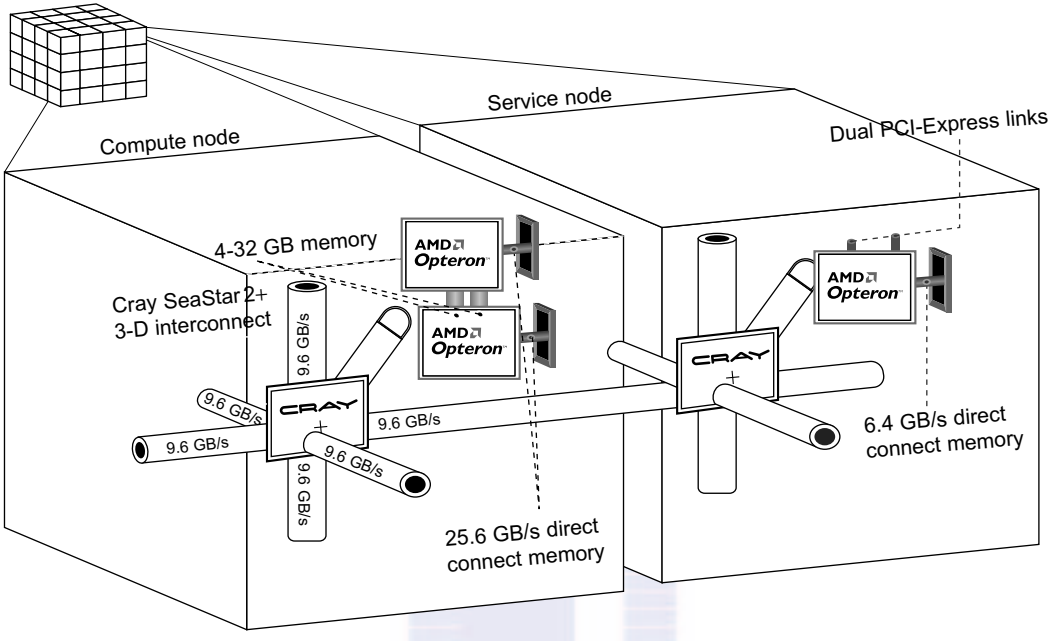


FIGURE 2.30

The 3D torus interconnect in the Cray XT5 Jaguar supercomputer.

(*Courtesy of Cray, Inc. [9] and Oak Ridge National Laboratory, United States, 2009*)

links which connect to six neighbors in the 3D torus. The peak bidirectional bandwidth of each link is 9.6 GB/second with sustained bandwidth in excess of 6 GB/second. Each port is configured with an independent router table, ensuring contention-free access for packets.

The router is designed with a reliable link-level protocol with error correction and retransmission, ensuring that message-passing traffic reliably reaches its destination without the costly timeout and retry mechanism used in typical clusters. The torus interconnect directly connects all the nodes in the Cray XT5 system, eliminating the cost and complexity of external switches and allowing for easy expandability. This allows systems to economically scale to tens of thousands of nodes—well beyond the capacity of fat-tree switches. The interconnect carries all message-passing and I/O traffic to the global file system.

2.5.2.2 Hardware Packaging

The Cray XT5 family employs an energy-efficient packaging technology, which reduces power use and thus lowers maintenance costs. The system's compute blades are packaged with only the necessary components for building an MPP with processors, memory, and interconnect. In a Cray XT5 cabinet, vertical cooling takes cold air straight from its source—the floor—and efficiently cools the processors on the blades, which are uniquely positioned for optimal airflow. Each processor also has a custom-designed heat sink depending on its position within the cabinet. Each Cray XT5 system cabinet is cooled with a single, high-efficiency ducted turbine fan. It takes 400/480VAC directly from the power grid without transformer and PDU loss.

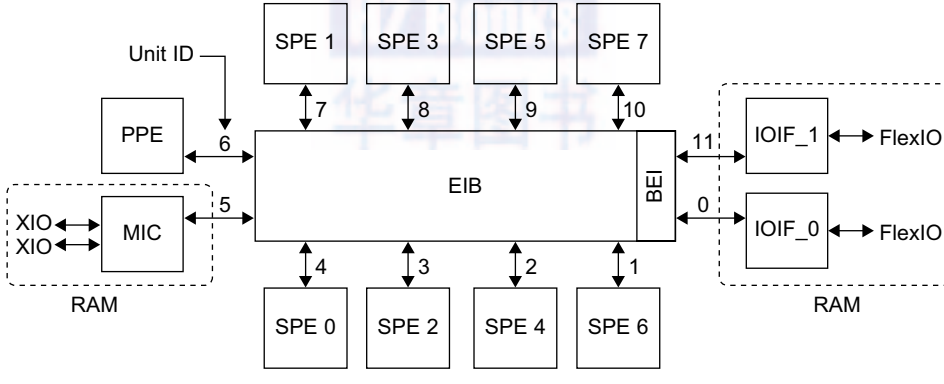
The Cray XT5 3D torus architecture is designed for superior MPI performance in HPC applications. This is accomplished by incorporating dedicated compute nodes and service nodes. Compute nodes are designed to run MPI tasks efficiently and reliably to completion. Each compute node is composed of one or two AMD Opteron microprocessors (dual or quad core) and direct attached memory, coupled with a dedicated communications resource. Service nodes are designed to provide system and I/O connectivity and also serve as login nodes from which jobs are compiled and launched. The I/O bandwidth of each compute node is designed for 25.6 GB/second performance.

2.5.3 IBM Roadrunner: The Top Supercomputer in 2008

In 2008, the IBM Roadrunner was the first general-purpose computer system in the world to reach petaflops performance. The system has a Linpack performance of 1.456 Pflops and is installed at the Los Alamos National Laboratory (LANL) in New Mexico. Subsequently, Cray's Jaguar topped the Roadrunner in late 2009. The system was used mainly to assess the decay of the U.S. nuclear arsenal. The system has a hybrid design with 12,960 IBM 3.2 GHz PowerXcell 8i CPUs (Figure 2.31) and 6,480 AMD 1.8 GHz Opteron 2210 dual-core processors. In total, the system has 122,400 cores. Roadrunner is an Opteron cluster accelerated by IBM Cell processors with eight floating-point cores.

2.5.3.1 Processor Chip and Compute Blade Design

The Cell/B.E. processors provide extraordinary compute power that can be harnessed from a single multicore chip. As shown in Figure 2.31, the Cell/B.E. architecture supports a very broad range of applications. The first implementation is a single-chip multiprocessor with nine processor elements



BEI	Cell broadband engine interface	MIC	Memory interface controller
EIB	Element interconnect bus	PPE	PowerPC processor element
FlexIO	Rambus flexIO bus	RAM	Resource allocation management
IOIF	I/O interface	SPE	Synergistic processor element
		XIO	Rambus XDR I/O (XIO) cell

FIGURE 2.31

Schematic of the IBMCell processor architecture.

(*Courtesy of IBM, http://www.redbooks.ibm.com/redpapers/pdfs/redp4477.pdf [28]*)

operating on a shared memory model. The rack is built with TriBlade servers, which are connected by an InfiniBand network. In order to sustain this compute power, the connectivity within each node consists of four PCI Express x8 links, each capable of 2 GB/s transfer rates, with a 2 μs latency. The expansion slot also contains the InfiniBand interconnect, which allows communications to the rest of the cluster. The capability of the InfiniBand interconnect is rated at 2 GB/s with a 2 μs latency.

2.5.3.2 InfiniBand Interconnect

The Roadrunner cluster was constructed hierarchically. The InfiniBand switches cluster together 18 connected units in 270 racks. In total, the cluster connects 12,960 IBM Power XCell 8i processors and 6,480 Opteron 2210 processors together with a total of 103.6 TB of RAM. This cluster complex delivers approximately 1.3 Pflops. In addition, the system's 18 Com/Service nodes deliver 4.5 Tflops using 18 InfiniBand switches. The second storage units are connected with eight InfiniBand switches. In total, 296 racks are installed in the system. The tiered architecture is constructed in two levels. The system consumes 2.35 MW power, and was the fourth most energy-efficient supercomputer built in 2009.

2.5.3.3 Message-Passing Performance

The Roadrunner uses MPI APIs to communicate with the other Opteron processors the application is running on in a typical single-program, multiple-data (SPMD) fashion. The number of compute nodes used to run the application is determined at program launch. The MPI implementation of Roadrunner is based on the open source Open MPI Project, and therefore is standard MPI. In this regard, Roadrunner applications are similar to other typical MPI applications such as those that run on the IBM Blue Gene solution. Where Roadrunner differs in the sphere of application architecture is how its Cell/B.E. accelerators are employed. At any point in the application flow, the MPI application running on each Opteron can offload computationally complex logic to its subordinate Cell/B.E. processor.

2.6 BIBLIOGRAPHIC NOTES AND HOMEWORK PROBLEMS

Cluster computing has been a hot research area since 1990. Cluster computing was pioneered by DEC and IBM as reported in Pfister [26]. His book provides a good introduction of several key concepts, including SSI and HA. Historically, milestone computer clusters include the VAXcluster running the VMS/OS in 1984, the Tandem Himalaya HA cluster (1994), and the IBM SP2 cluster in 1996. These earlier clusters were covered in [3,7,14,23,26]. In recent years, more than 85 percent of the Top 500 systems are built with cluster configurations [9,11,20,25,28,29].

Annually, IEEE and ACM hold several international conferences related to this topic. They include Cluster Computing (Cluster); Supercomputing Conference (SC); International Symposium on Parallel and Distributed Systems (IPDPS); International Conferences on Distributed Computing Systems (ICDCS); High-Performance Distributed Computing (HPDC); and Clusters, Clouds, and The Grids (CCGrid). There are also several journals related to this topic, including the Journal of Cluster Computing, Journal of Parallel and Distributed Computing (JPDC), and IEEE Transactions on Parallel and Distributed Systems (TPDS).

Cluster applications are assessed in Bader and Pennington [2]. Some figures and examples in this chapter are modified from the earlier book by Hwang and Xu [14]. Buyya has treated cluster

computing in two edited volumes [7]. Two books on Linux clusters are [20,23]. HA clusters are treated in [24]. Recent assessment of HPC interconnects can be found in [6,8,12,22]. The Google cluster interconnect was reported by Barroso, et al. [6]. GPUs for supercomputing was discussed in [10]. GPU clusters were studied in [19]. CUDA parallel programming for GPUs is treated in [31]. MOSIX/OS for cluster or grid computing is treated in [4,5,30].

Hwang, Jin, and Ho developed a distributed RAID system for achieving a single I/O space in a cluster of PCs or workstations [13–17]. More details of LSF can be found in Zhou [35]. The Top 500 list was cited from the release in June and November 2010 [25]. The material on the Tianhe-1A can be found in Dongarra [11] and on Wikipedia [29]. The IBM Blue Gene/L architecture was reported by Adiga, et al. [1] and subsequently upgraded to a newer model called the Blue Gene/P solution. The IBM Roadrunner was reported by Kevin, et al. [18] and also in Wikipedia [28]. The Cray XT5 and Jaguar systems are described in [9]. China's Nebulae supercomputer was reported in [27]. Specific cluster applications and checkpointing techniques can be found in [12,16,17,24,32,34]. Cluster applications can be found in [7,15,18,21,26,27,33,34].

Acknowledgments

This chapter is authored by Kai Hwang of USC and by Jack Dongarra of UTK jointly. Some cluster material are borrowed from the earlier book [14] by Kai Hwang of USC and Zhiwei Xu of the Chinese Academy of Sciences. Valuable suggestions to update the material were made by Rajkumar Buyya of the University of Melbourne.

References

[1] N. Adiga, et al., An overview of the blue gene/L supercomputer, in: ACM Supercomputing Conference 2002, November 2002, http://SC-2002.org/paperpdfs/pap.pap207.pdf.
[2] D. Bader, R. Pennington, Cluster computing applications, Int. J. High Perform. Comput. (May) (2001).
[3] M. Baker, et al., Cluster computing white paper. http://arxiv.org/abs/cs/0004014, January 2001.
[4] A. Barak, A. Shiloh, The MOSIX Management Systems for Linux Clusters, Multi-Clusters and Clouds. White paper, www.MOSIX.org//txt_pub.html, 2010.
[5] A. Barak, R. La'adan, The MOSIX multicomputer operating systems for high-performance cluster computing, Future Gener. Comput. Syst. 13 (1998) 361–372.
[6] L. Barroso, J. Dean, U. Holzle, Web search for a planet: The Google cluster architecture, IEEE Micro. 23 (2) (2003) 22–28.
[7] R. Buyya (Ed.), High-Performance Cluster Computing. Vols. 1 and 2, Prentice Hall, New Jersey, 1999.
[8] O. Celebioglu, R. Rajagopalan, R. Ali, Exploring InfiniBand as an HPC cluster interconnect, (October) (2004).
[9] Cray, Inc, CrayXT System Specifications. www.cray.com/Products/XT/Specifications.aspx, January 2010.
[10] B. Dally, GPU Computing to Exascale and Beyond, Keynote Address, ACM Supercomputing Conference, November 2010.
[11] J. Dongarra, Visit to the National Supercomputer Center in Tianjin, China, Technical Report, University of Tennessee and Oak Ridge National Laboratory, 20 February 2011.
[12] J. Dongarra, Survey of present and future supercomputer architectures and their interconnects, in: International Supercomputer Conference, Heidelberg, Germany, 2004.

[13] K. Hwang, H. Jin, R.S. Ho, Orthogonal striping and mirroring in distributed RAID for I/O-Centric cluster computing, IEEE Trans. Parallel Distrib. Syst. 13 (2) (2002) 26–44.
[14] K. Hwang, Z. Xu, Support of clustering and availability, in: Scalable Parallel Computing, McGraw-Hill, 1998, Chapter 9.
[15] K. Hwang, C.M. Wang, C.L. Wang, Z. Xu, Resource scaling effects on MPP performance: STAP benchmark implications, IEEE Trans. Parallel Distrib. Syst. (May) (1999) 509–527.
[16] K. Hwang, H. Jin, E. Chow, C.L. Wang, Z. Xu, Designing SSI clusters with hierarchical checkpointing and single-I/O space, IEEE Concurrency (January) (1999) 60–69.
[17] H. Jin, K. Hwang, Adaptive sector grouping to reduce false sharing of distributed RAID clusters, J. Clust. Comput. 4 (2) (2001) 133–143.
[18] J. Kevin, et al., Entering the petaflop era: the architecture of performance of Roadrunner, www.c3.lanl.gov/~kei/mypubbib/papers/SC08:Roadrunner.pdf, November 2008.
[19] V. Kindratenko, et al., GPU Clusters for High-Performance Computing, National Center for Supercomputing Applications, University of Illinois at Urban-Champaign, Urbana, IL, 2009.
[20] K. Kopper, The Linux Enterprise Cluster: Building a Highly Available Cluster with Commodity Hardware and Free Software, No Starch Press, San Francisco, CA, 2005.
[21] S.W. Lin, R.W. Lau, K. Hwang, X. Lin, P.Y. Cheung, Adaptive parallel Image rendering on multiprocessors and workstation clusters. IEEE Trans. Parallel Distrib. Syst. 12 (3) (2001) 241–258.
[22] J. Liu, D.K. Panda, et al., Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics, (2003).
[23] R. Lucke, Building Clustered Linux Systems, Prentice Hall, New Jersey, 2005.
[24] E. Marcus, H. Stern, Blueprints for High Availability: Designing Resilient Distributed Systems, Wiley.
[25] TOP500.org. Top-500 World's fastest supercomputers, www.top500.org, November 2010.
[26] G.F. Pfister, In Search of Clusters, second ed., Prentice-Hall, 2001.
[27] N.H. Sun, China's Nebulae Supercomputer, Institute of Computing Technology, Chinese Academy of Sciences, July 2010.
[28] Wikipedia, IBM Roadrunner. http://en.wikipedia.org/wiki/IBM_Roadrunner, 2010, (accessed 10.01.10).
[29] Wikipedia, Tianhe-1. http://en.wikipedia.org/wiki/Tianhe-1, 2011, (accessed 5.02.11).
[30] Wikipedia, MOSIX. http://en.wikipedia.org/wiki/MOSIX, 2011, (accessed 10.02.11).
[31] Wikipedia, CUDA. http://en.wikipedia.org/wiki/CUDA, 2011, (accessed 19.02.11).
[32] K. Wong, M. Franklin, Checkpointing in distributed computing systems, J. Parallel Distrib. Comput. (1996) 67–75.
[33] Z. Xu, K. Hwang, Designing superservers with clusters and commodity components. Annual Advances in Scalable Computing, World Scientific, Singapore, 1999.
[34] Z. Xu, K. Hwang, MPP versus clusters for scalable computing, in: Proceedings of the 2nd IEEE International Symposium on Parallel Architectures, Algorithms, and Networks, June 1996, pp. 117–123.
[35] S. Zhou, LSF: Load Sharing and Batch Queuing Software, Platform Computing Corp., Canada, 1996.

HOMEWORK PROBLEMS

Problem 2.1

Differentiate and exemplify the following terms related to clusters:

a. Compact versus slack clusters
b. Centralized versus decentralized clusters

c. Homogeneous versus heterogeneous clusters
d. Enclosed versus exposed clusters
e. Dedicated versus enterprise clusters

Problem 2.2

This problem refers to the redundancy technique. Assume that when a node fails, it takes 10 seconds to diagnose the fault and another 30 seconds for the workload to be switched over.

a. What is the availability of the cluster if planned downtime is ignored?
b. What is the availability of the cluster if the cluster is taken down one hour per week for maintenance, but one node at a time?

Problem 2.3

This is a research project to evaluate the cluster architectures of four supercomputers built in recent years. Study the details of the No. 1 supercomputer, the Tianhe-1A, which was announced in the Top 500 list released in November 2010. Your study should include the following:

a. Conduct an in-depth evaluation of the Tianhe-1A architecture, hardware components, operating system, software support, parallelizing compilers, packaging, cooling, and new applications.
b. Compare the relative strengths and limitations of the Tianhe-1A with respect to the three case-study systems: the Jaguar, Nebulae, and Roadrunner, studied in Section 2.5. Use tabulations or plot curves, if you find enough benchmark data to conduction the comparison study.

Problem 2.4

This problem consists of two parts related to cluster computing:

1. Define and distinguish among the following terms on scalability:
   a. Scalability over machine size
   b. Scalability over problem size
   c. Resource scalability
   d. Generation scalability
2. Explain the architectural and functional differences among three availability cluster configurations: *hot standby, active takeover*, and *fault-tolerant clusters*. Give two example commercial cluster systems in each availability cluster configuration. Comment on their relative strengths and weaknesses in commercial applications.

Problem 2.5

Distinguish between multiprocessors and multicomputers based on their structures, resource sharing, and interprocessor communications.

a. Explain the differences among UMA, NUMA, COMA, DSM, and NORMA memory models.
b. What are the additional functional features of a cluster that are not found in a conventional network of autonomous computers?
c. What are the advantages of a clustered system over a traditional SMP server?

Problem 2.6

Study the five research virtual cluster projects listed in Table 2.6 and answer the following questions regarding the coverage on COD and Violin experience given in Sections 2.5.3 and 2.5.4:

a. From the viewpoints of dynamic resource provisioning, evaluate the five virtual clusters and discuss their relative strengths and weaknesses based on the open literature.

b. Report on the unique contribution from each of the five virtual cluster projects in terms of the hardware setting, software tools, and experimental environments developed and performance results reported.

Problem 2.7

This problem is related to the use of high-end x86 processors in HPC system construction. Answer the following questions:

a. Referring to the latest Top 500 list of supercomputing systems, list all systems that have used x86 processors. Identify the processor models and key processor characteristics such as number of cores, clock frequency, and projected performance.

b. Some have used GPUs to complement the x86 CPUs. Identify those systems that have procured substantial GPUs. Discuss the roles of GPUs to provide peak or sustained flops per dollar.

Problem 2.8

Assume a sequential computer has 512 MB of main memory and enough disk space. The disk read/write bandwidth for a large data block is 1 MB/second. The following code needs to apply checkpointing:

```
do 1000 iterations
A = foo (C from last iteration)      /* this statement takes 10 minutes */
B = goo (A)                          /* this statement takes 10 minutes */
C = hoo (B)                          /* this statement takes 10 minutes */
end do
```

A, B, and C are arrays of 120 MB each. All other parts of the code, operating system, libraries take, at most, 16 MB of memory. Assume the computer fails exactly once, and the time to restore the computer is ignored.

a. What is the worst-case execution time for the successful completion of the code if checkpointing is performed?

b. What is the worst-case execution time for the successful completion of the code if plain transparent checkpointing is performed?

c. Is it beneficial to use forked checkpointing with (b)?

d. What is the worst-case execution time for the code if user-directed checkpointing is performed? Show the code where user directives are added.

e. What is the worst-case execution time of the code if forked checkpointing is used with (d)?

Problem 2.9

Compare the latest Top 500 list with the Top 500 Green List of HPC systems. Discuss a few top winners and losers in terms of energy efficiency in power and cooling costs. Reveal the green-energy winners' stories and report their special design features, packaging, cooling, and management policies that make them the winners. How different are the ranking orders in the two lists? Discuss their causes and implications based on publicly reported data.

Problem 2.10

This problem is related to processor selection and system interconnects used in building the top three clustered systems with commercial interconnects in the latest Top 500 list.

a. Compare the processors used in these clusters and identify their strengths and weaknesses in terms of potential peak floating-point performance.
b. Compare the commercial interconnects of these three clusters. Discuss their potential performance in terms of their topological properties, network latency, bisection bandwidth, and hardware used.

Problem 2.11

Study Example 2.6 and the original paper [14] reporting the distributed RAID-x architecture and performance results. Answer the following questions with technical justifications or evidence:

a. Explain how the RAID-x system achieved a single I/O address space across distributed disks attached to cluster nodes.
b. Explain the functionality of the cooperative disk drivers (CCDs) implemented in the RAID-x system. Comment on its application requirements and scalability based on current PC architecture, SCSI bus, and SCSI disk technology.
c. Explain why RAID-x has a fault-tolerance capability equal to that of the RAID-5 architecture.
d. Explain the strengths and limitations of RAID-x, compared with other RAID architectures.

Problem 2.12

Study the relevant material in Sections 2.2 and 2.5 and compare the system interconnects of the IBM Blue Gene/L, IBM Roadrunner, and Cray XT5 supercomputers released in the November 2009 Top 500 evaluation. Dig deeper to reveal the details of these systems. These systems may use custom-designed routers in interconnects. Some also use some commercial interconnects and components.

a. Compare the basic routers or switches used in the three system interconnects in terms of technology, chip design, routing scheme, and claimed message-passing performance.
b. Compare the topological properties, network latency, bisection bandwidth, and hardware packaging of the three system interconnects.

Problem 2.13

Study the latest and largest commercial HPC clustered system built by SGI, and report on the cluster architecture in the following technical and benchmark aspects:

a. What is the SGI system model and its specification? Illustrate the cluster architecture with a block diagram and describe the functionality of each building block.

b. Discuss the claimed peak performance and reported sustained performance from SGI.
c. What are the unique hardware, software, networking, or design features that contribute to the claimed performance in Part (b)? Describe or illustrate those system features.

Problem 2.14

Consider in Figure 2.32 a server-client cluster with an active-takeover configuration between two identical servers. The servers share a disk via a SCSI bus. The clients (PCs or workstations) and the Ethernet are fail-free. When a server fails, its workload is switched to the surviving server.

a. Assume that each server has an MTTF of 200 days and an MTTR of five days. The disk has an MTTF of 800 days and an MTTR of 20 days. In addition, each server is shut down for maintenance for one day every week, during which time that server is considered unavailable. Only one server is shut down for maintenance at a time. The failure rates cover both natural failures and scheduled maintenance. The SCSI bus has a failure rate of 2 percent. The servers and the disk fail independently. The disk and SCSI bus have no scheduled shutdown. The client machine will never fail.
1. The servers are considered available if at least one server is available. What is the combined availability of the two servers?
2. In normal operation, the cluster must have the SCSI bus, the disk, and at least one server available simultaneously. What are the possible single points of failure in this cluster?
b. The cluster is considered unacceptable if both servers fail at the same time. Furthermore, the cluster is declared unavailable when either the SCSI bus or the disk is down. Based on the aforementioned conditions, what is the system availability of the entire cluster?
c. Under the aforementioned failure and maintenance conditions, propose an improved architecture to eliminate all single points of failure identified in Part (a).

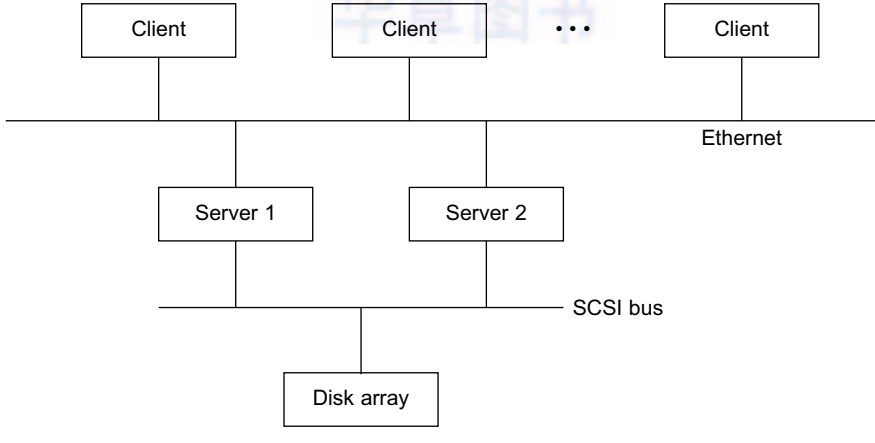


FIGURE 2.32
An HA cluster with redundant hardware components.

Problem 2.15

Study various cluster job scheduling policies in Table 2.6 and answer the following questions. You may need to gather more information from Wikipedia, Google, or other sources if any of the scheduling policies are new to you.

a. Explain the advantages and disadvantages of nonpreemptive and preemptive scheduling policies and suggest methods to amend the problems.
b. Repeat Part (a) for static and dynamic scheduling policies.
c. Repeat Part (a) for dedicated and space-sharing scheduling policies.
d. Compare the relative performance of time-sharing, independent, and gang scheduling policies.
e. Compare the relative performance in stay and migrating policies on local jobs against remote jobs.

Problem 2.16

Study various SSI features and HA support for clusters in Section 2.3 and answer the following questions, providing reasons for your answers. Identify some example cluster systems that are equipped with these features. Comment on their implementation requirements and discuss the operational obstacles to establish each SSI feature in a cluster system.

a. Single entry point in a cluster environment
b. Single memory space in a cluster system
c. Single file hierarchy in a cluster system
d. Single I/O space in a cluster system
e. Single network space in a cluster system
f. Single networking in a cluster system
g. Single point of control in a cluster system
h. Single job management in a cluster system
i. Single user interface in a cluster system
j. Single process space in a cluster system

Problem 2.17

Use examples to explain the following terms on cluster job management systems.

a. Serial jobs versus parallel jobs
b. Batch jobs versus interactive jobs
c. Cluster jobs versus foreign (local) jobs
d. Cluster processes, local processes, and kernel processes
e. Dedicated mode, space-sharing mode, and timesharing mode
f. Independent scheduling versus gang scheduling

Problem 2.18

This problem focuses on the concept of LSF:

a. Give an example of each of the four types of LSF jobs.
b. For a 1,000-server cluster, give two reasons why the LSF load-sharing policy is better if (1) the entire cluster has one master LIM or (2) all LIMs are masters.

c. In the LSF master-election scheme, a node in the "no master" state waits for a time period proportional to the node number before becoming a new master. Why is the wait time proportional to the node number?

Problem 2.19

This problem is related to the use of MOSIX for cluster computing. Check with the open literature on current features that have been claimed by designers and developers in supporting Linux clusters, GPU clusters, multiclusters, and even virtualized clouds. Discuss the advantages and shortcomings from the user's perspective.

Problem 2.20

Compare China's Tianhe-1A with the Cray Jaguar in terms of their relative strengths and weaknesses in architecture design, resource management, software environment, and reported applications. You may need to conduct some research to find the latest developments regarding these systems. Justify your assessment with reasoning and evidential information.

CHAPTER

Virtual Machines and Virtualization of Clusters and Data Centers

3

CHAPTER OUTLINE

SUMMARY

The reincarnation of *virtual machines* (VMs) presents a great opportunity for parallel, cluster, grid, cloud, and distributed computing. Virtualization technology benefits the computer and IT industries by enabling users to share expensive hardware resources by multiplexing VMs on the same set of hardware hosts. This chapter covers virtualization levels, VM architectures, virtual networking, virtual cluster construction, and virtualized data-center design and automation in cloud computing. In particular, the designs of dynamically structured clusters, grids, and clouds are presented with VMs and virtual clusters.

3.1 IMPLEMENTATION LEVELS OF VIRTUALIZATION

Virtualization is a computer architecture technology by which multiple *virtual machines* (VMs) are multiplexed in the same hardware machine. The idea of VMs can be dated back to the 1960s [53]. The purpose of a VM is to enhance resource sharing by many users and improve computer performance in terms of resource utilization and application flexibility. Hardware resources (CPU, memory, I/O devices, etc.) or software resources (operating system and software libraries) can be virtualized in various functional layers. This virtualization technology has been revitalized as the demand for distributed and cloud computing increased sharply in recent years [41].

The idea is to separate the hardware from the software to yield better system efficiency. For example, computer users gained access to much enlarged memory space when the concept of *virtual memory* was introduced. Similarly, virtualization techniques can be applied to enhance the use of compute engines, networks, and storage. In this chapter we will discuss VMs and their applications for building distributed systems. According to a 2009 Gartner Report, virtualization was the top strategic technology poised to change the computer industry. With sufficient storage, any computer platform can be installed in another host computer, even if they use processors with different instruction sets and run with distinct operating systems on the same hardware.

3.1.1 Levels of Virtualization Implementation

A traditional computer runs with a host operating system specially tailored for its hardware architecture, as shown in Figure 3.1(a). After virtualization, different user applications managed by their own operating systems (guest OS) can run on the same hardware, independent of the host OS. This is often done by adding additional software, called a *virtualization layer* as shown in Figure 3.1(b). This virtualization layer is known as *hypervisor* or *virtual machine monitor* (VMM) [54]. The VMs are shown in the upper boxes, where applications run with their own guest OS over the virtualized CPU, memory, and I/O resources.

The main function of the software layer for virtualization is to virtualize the physical hardware of a host machine into virtual resources to be used by the VMs, exclusively. This can be implemented at various operational levels, as we will discuss shortly. The virtualization software creates the abstraction of VMs by interposing a virtualization layer at various levels of a computer system. Common virtualization layers include the *instruction set architecture (ISA)* level, hardware level, operating system level, library support level, and application level (see Figure 3.2).

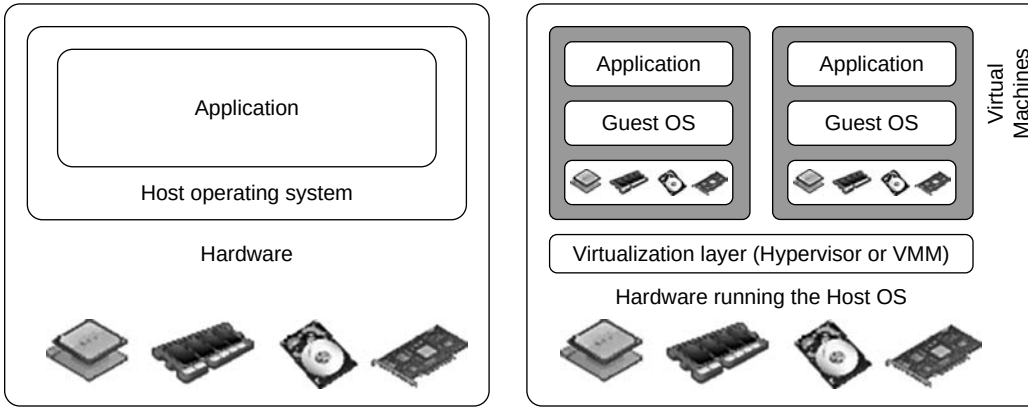


FIGURE 3.1

The architecture of a computer system before and after virtualization, where VMM stands for virtual machine monitor.

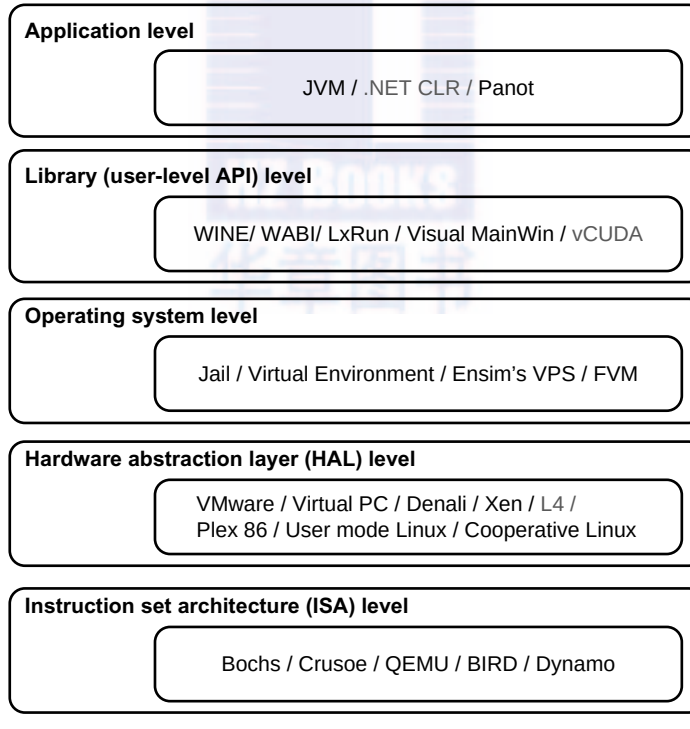


FIGURE 3.2

Virtualization ranging from hardware to applications in five abstraction levels.

3.1.1.1 Instruction Set Architecture Level

At the ISA level, virtualization is performed by emulating a given ISA by the ISA of the host machine. For example, MIPS binary code can run on an x86-based host machine with the help of ISA emulation. With this approach, it is possible to run a large amount of legacy binary code written for various processors on any given new hardware host machine. Instruction set emulation leads to virtual ISAs created on any hardware machine.

The basic emulation method is through *code interpretation*. An interpreter program interprets the source instructions to target instructions one by one. One source instruction may require tens or hundreds of native target instructions to perform its function. Obviously, this process is relatively slow. For better performance, *dynamic binary translation* is desired. This approach translates basic blocks of dynamic source instructions to target instructions. The basic blocks can also be extended to program traces or super blocks to increase translation efficiency. Instruction set emulation requires binary translation and optimization. A *virtual instruction set architecture (V-ISA)* thus requires adding a processor-specific software translation layer to the compiler.

3.1.1.2 Hardware Abstraction Level

Hardware-level virtualization is performed right on top of the bare hardware. On the one hand, this approach generates a virtual hardware environment for a VM. On the other hand, the process manages the underlying hardware through virtualization. The idea is to virtualize a computer's resources, such as its processors, memory, and I/O devices. The intention is to upgrade the hardware utilization rate by multiple users concurrently. The idea was implemented in the IBM VM/370 in the 1960s. More recently, the Xen hypervisor has been applied to virtualize x86-based machines to run Linux or other guest OS applications. We will discuss hardware virtualization approaches in more detail in Section 3.3.

3.1.1.3 Operating System Level

This refers to an abstraction layer between traditional OS and user applications. OS-level virtualization creates isolated *containers* on a single physical server and the OS instances to utilize the hardware and software in data centers. The containers behave like real servers. OS-level virtualization is commonly used in creating virtual hosting environments to allocate hardware resources among a large number of mutually distrusting users. It is also used, to a lesser extent, in consolidating server hardware by moving services on separate hosts into containers or VMs on one server. OS-level virtualization is depicted in Section 3.1.3.

3.1.1.4 Library Support Level

Most applications use APIs exported by user-level libraries rather than using lengthy system calls by the OS. Since most systems provide well-documented APIs, such an interface becomes another candidate for virtualization. Virtualization with library interfaces is possible by controlling the communication link between applications and the rest of a system through API hooks. The software tool WINE has implemented this approach to support Windows applications on top of UNIX hosts. Another example is the vCUDA which allows applications executing within VMs to leverage GPU hardware acceleration. This approach is detailed in Section 3.1.4.

3.1.1.5 User-Application Level

Virtualization at the application level virtualizes an application as a VM. On a traditional OS, an application often runs as a process. Therefore, *application-level virtualization* is also known as

process-level virtualization. The most popular approach is to deploy *high level language (HLL)* VMs. In this scenario, the virtualization layer sits as an application program on top of the operating system, and the layer exports an abstraction of a VM that can run programs written and compiled to a particular abstract machine definition. Any program written in the HLL and compiled for this VM will be able to run on it. The Microsoft .NET CLR and *Java Virtual Machine (JVM)* are two good examples of this class of VM.

Other forms of application-level virtualization are known as *application isolation*, *application sandboxing*, or *application streaming*. The process involves wrapping the application in a layer that is isolated from the host OS and other applications. The result is an application that is much easier to distribute and remove from user workstations. An example is the LANDesk application virtualization platform which deploys software applications as self-contained, executable files in an isolated environment without requiring installation, system modifications, or elevated security privileges.

3.1.1.6 Relative Merits of Different Approaches

Table 3.1 compares the relative merits of implementing virtualization at various levels. The column headings correspond to four technical merits. "Higher Performance" and "Application Flexibility" are self-explanatory. "Implementation Complexity" implies the cost to implement that particular virtualization level. "Application Isolation" refers to the effort required to isolate resources committed to different VMs. Each row corresponds to a particular level of virtualization.

The number of X's in the table cells reflects the advantage points of each implementation level. Five X's implies the best case and one X implies the worst case. Overall, hardware and OS support will yield the highest performance. However, the hardware and application levels are also the most expensive to implement. User isolation is the most difficult to achieve. ISA implementation offers the best application flexibility.

3.1.2 VMM Design Requirements and Providers

As mentioned earlier, hardware-level virtualization inserts a layer between real hardware and traditional operating systems. This layer is commonly called the *Virtual Machine Monitor (VMM)* and it manages the hardware resources of a computing system. Each time programs access the hardware the VMM captures the process. In this sense, the VMM acts as a traditional OS. One hardware component, such as the CPU, can be virtualized as several virtual copies. Therefore, several traditional operating systems which are the same or different can sit on the same set of hardware simultaneously.

Table 3.1 Relative Merits of Virtualization at Various Levels (More "X"'s Means Higher Merit, with a Maximum of 5 X's)

Level of Implementation	Higher Performance	Application Flexibility	Implementation Complexity	Application Isolation
ISA	X	XXXXX	XXX	XXX
Hardware-level virtualization	XXXXX	XXX	XXXXX	XXXX
OS-level virtualization	XXXXX	XX	XXX	XX
Runtime library support	XXX	XX	XX	XX
User application level	XX	XX	XXXXX	XXXXX

There are three requirements for a VMM. First, a VMM should provide an environment for programs which is essentially identical to the original machine. Second, programs run in this environment should show, at worst, only minor decreases in speed. Third, a VMM should be in complete control of the system resources. Any program run under a VMM should exhibit a function identical to that which it runs on the original machine directly. Two possible exceptions in terms of differences are permitted with this requirement: differences caused by the availability of system resources and differences caused by timing dependencies. The former arises when more than one VM is running on the same machine.

The hardware resource requirements, such as memory, of each VM are reduced, but the sum of them is greater than that of the real machine installed. The latter qualification is required because of the intervening level of software and the effect of any other VMs concurrently existing on the same hardware. Obviously, these two differences pertain to performance, while the function a VMM provides stays the same as that of a real machine. However, the identical environment requirement excludes the behavior of the usual time-sharing operating system from being classed as a VMM.

A VMM should demonstrate efficiency in using the VMs. Compared with a physical machine, no one prefers a VMM if its efficiency is too low. Traditional emulators and complete software interpreters (simulators) emulate each instruction by means of functions or macros. Such a method provides the most flexible solutions for VMMs. However, emulators or simulators are too slow to be used as real machines. To guarantee the efficiency of a VMM, a statistically dominant subset of the virtual processor's instructions needs to be executed directly by the real processor, with no software intervention by the VMM. Table 3.2 compares four hypervisors and VMMs that are in use today.

Complete control of these resources by a VMM includes the following aspects: (1) The VMM is responsible for allocating hardware resources for programs; (2) it is not possible for a program to access any resource not explicitly allocated to it; and (3) it is possible under certain circumstances for a VMM to regain control of resources already allocated. Not all processors satisfy these requirements for a VMM. A VMM is tightly related to the architectures of processors. It is difficult to

Table 3.2 Comparison of Four VMM and Hypervisor Software Packages

Provider and References	Host CPU	Host OS	Guest OS	Architecture
VMware Workstation [71]	x86, x86-64	Windows, Linux	Windows, Linux, Solaris, FreeBSD, Netware, OS/2, SCO, BeOS, Darwin	Full Virtualization
VMware ESX Server [71]	x86, x86-64	No host OS	The same as VMware Workstation	Para-Virtualization
Xen [7,13,42]	x86, x86-64, IA-64	NetBSD, Linux, Solaris	FreeBSD, NetBSD, Linux, Solaris, Windows XP and 2003 Server	Hypervisor
KVM [31]	x86, x86-64, IA-64, S390, PowerPC	Linux	Linux, Windows, FreeBSD, Solaris	Para-Virtualization

implement a VMM for some types of processors, such as the x86. Specific limitations include the inability to trap on some privileged instructions. If a processor is not designed to support virtualization primarily, it is necessary to modify the hardware to satisfy the three requirements for a VMM. This is known as hardware-assisted virtualization.

3.1.3 Virtualization Support at the OS Level

With the help of VM technology, a new computing mode known as cloud computing is emerging. Cloud computing is transforming the computing landscape by shifting the hardware and staffing costs of managing a computational center to third parties, just like banks. However, cloud computing has at least two challenges. The first is the ability to use a variable number of physical machines and VM instances depending on the needs of a problem. For example, a task may need only a single CPU during some phases of execution but may need hundreds of CPUs at other times. The second challenge concerns the slow operation of instantiating new VMs. Currently, new VMs originate either as fresh boots or as replicates of a template VM, unaware of the current application state. Therefore, to better support cloud computing, a large amount of research and development should be done.

3.1.3.1 Why OS-Level Virtualization?

As mentioned earlier, it is slow to initialize a hardware-level VM because each VM creates its own image from scratch. In a cloud computing environment, perhaps thousands of VMs need to be initialized simultaneously. Besides slow operation, storing the VM images also becomes an issue. As a matter of fact, there is considerable repeated content among VM images. Moreover, full virtualization at the hardware level also has the disadvantages of slow performance and low density, and the need for para-virtualization to modify the guest OS. To reduce the performance overhead of hardware-level virtualization, even hardware modification is needed. OS-level virtualization provides a feasible solution for these hardware-level virtualization issues.

Operating system virtualization inserts a virtualization layer inside an operating system to partition a machine's physical resources. It enables multiple isolated VMs within a single operating system kernel. This kind of VM is often called a *virtual execution environment (VE)*, *Virtual Private System (VPS)*, or simply *container*. From the user's point of view, VEs look like real servers. This means a VE has its own set of processes, file system, user accounts, network interfaces with IP addresses, routing tables, firewall rules, and other personal settings. Although VEs can be customized for different people, they share the same operating system kernel. Therefore, OS-level virtualization is also called single-OS image virtualization. Figure 3.3 illustrates operating system virtualization from the point of view of a machine stack.

3.1.3.2 Advantages of OS Extensions

Compared to hardware-level virtualization, the benefits of OS extensions are twofold: (1) VMs at the operating system level have minimal startup/shutdown costs, low resource requirements, and high scalability; and (2) for an OS-level VM, it is possible for a VM and its host environment to synchronize state changes when necessary. These benefits can be achieved via two mechanisms of OS-level virtualization: (1) All OS-level VMs on the same physical machine share a single operating system kernel; and (2) the virtualization layer can be designed in a way that allows processes in VMs to access as many resources of the host machine as possible, but never to modify them. In cloud

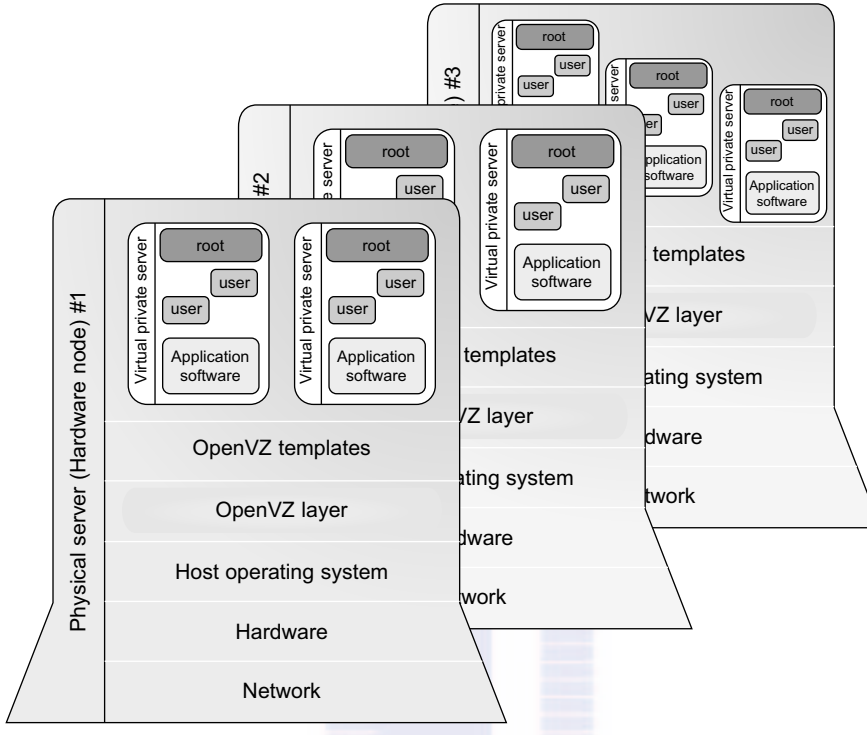


FIGURE 3.3

The OpenVZ virtualization layer inside the host OS, which provides some OS images to create VMs quickly. (*Courtesy of OpenVZ User's Guide [65]*)

computing, the first and second benefits can be used to overcome the defects of slow initialization of VMs at the hardware level, and being unaware of the current application state, respectively.

3.1.3.3 Disadvantages of OS Extensions

The main disadvantage of OS extensions is that all the VMs at operating system level on a single container must have the same kind of guest operating system. That is, although different OS-level VMs may have different operating system distributions, they must pertain to the same operating system family. For example, a Windows distribution such as Windows XP cannot run on a Linux-based container. However, users of cloud computing have various preferences. Some prefer Windows and others prefer Linux or other operating systems. Therefore, there is a challenge for OS-level virtualization in such cases.

Figure 3.3 illustrates the concept of OS-level virtualization. The virtualization layer is inserted inside the OS to partition the hardware resources for multiple VMs to run their applications in multiple virtual environments. To implement OS-level virtualization, isolated execution environments (VMs) should be created based on a single OS kernel. Furthermore, the access requests from a VM need to be redirected to the VM's local resource partition on the physical machine. For

example, the *chroot* command in a UNIX system can create several virtual root directories within a host OS. These virtual root directories are the root directories of all VMs created.

There are two ways to implement virtual root directories: duplicating common resources to each VM partition; or sharing most resources with the host environment and only creating private resource copies on the VM on demand. The first way incurs significant resource costs and overhead on a physical machine. This issue neutralizes the benefits of OS-level virtualization, compared with hardware-assisted virtualization. Therefore, OS-level virtualization is often a second choice.

3.1.3.4 Virtualization on Linux or Windows Platforms

By far, most reported OS-level virtualization systems are Linux-based. Virtualization support on the Windows-based platform is still in the research stage. The Linux kernel offers an abstraction layer to allow software processes to work with and operate on resources without knowing the hardware details. New hardware may need a new Linux kernel to support. Therefore, different Linux platforms use patched kernels to provide special support for extended functionality.

However, most Linux platforms are not tied to a special kernel. In such a case, a host can run several VMs simultaneously on the same hardware. Table 3.3 summarizes several examples of OS-level virtualization tools that have been developed in recent years. Two OS tools (Linux vServer and OpenVZ) support Linux platforms to run other platform-based applications through virtualization. These two OS-level tools are illustrated in Example 3.1. The third tool, FVM, is an attempt specifically developed for virtualization on the Windows NT platform.

Example 3.1 Virtualization Support for the Linux Platform

OpenVZ is an OS-level tool designed to support Linux platforms to create virtual environments for running VMs under different guest OSes. OpenVZ is an open source container-based virtualization solution built on Linux. To support virtualization and isolation of various subsystems, limited resource management, and checkpointing, OpenVZ modifies the Linux kernel. The overall picture of the OpenVZ system is illustrated in Figure 3.3. Several VPSes can run simultaneously on a physical machine. These VPSes look like normal

Table 3.3 Virtualization Support for Linux and Windows NT Platforms

Virtualization Support and Source of Information	Brief Introduction on Functionality and Application Platforms
Linux vServer for Linux platforms (http://linux-vserver.org/)	Extends Linux kernels to implement a security mechanism to help build VMs by setting resource limits and file attributes and changing the root environment for VM isolation
OpenVZ for Linux platforms [65]; http://ftp.openvz.org/doc/OpenVZ-Users-Guide.pdf)	Supports virtualization by creating *virtual private servers (VPSes)*; the VPS has its own files, users, process tree, and virtual devices, which can be isolated from other VPSes, and checkpointing and live migration are supported
FVM (Feather-Weight Virtual Machines) for virtualizing the Windows NT platforms [78])	Uses system call interfaces to create VMs at the NY kernel space; multiple VMs are supported by virtualized namespace and copy-on-write

Linux servers. Each VPS has its own files, users and groups, process tree, virtual network, virtual devices, and IPC through semaphores and messages.

The resource management subsystem of OpenVZ consists of three components: two-level disk allocation, a two-level CPU scheduler, and a resource controller. The amount of disk space a VM can use is set by the OpenVZ server administrator. This is the first level of disk allocation. Each VM acts as a standard Linux system. Hence, the VM administrator is responsible for allocating disk space for each user and group. This is the second-level disk quota. The first-level CPU scheduler of OpenVZ decides which VM to give the time slice to, taking into account the virtual CPU priority and limit settings.

The second-level CPU scheduler is the same as that of Linux. OpenVZ has a set of about 20 parameters which are carefully chosen to cover all aspects of VM operation. Therefore, the resources that a VM can use are well controlled. OpenVZ also supports checkpointing and live migration. The complete state of a VM can quickly be saved to a disk file. This file can then be transferred to another physical machine and the VM can be restored there. It only takes a few seconds to complete the whole process. However, there is still a delay in processing because the established network connections are also migrated.

3.1.4 Middleware Support for Virtualization

Library-level virtualization is also known as user-level *Application Binary Interface (ABI)* or API emulation. This type of virtualization can create execution environments for running alien programs on a platform rather than creating a VM to run the entire operating system. API call interception and remapping are the key functions performed. This section provides an overview of several library-level virtualization systems: namely the *Windows Application Binary Interface (WABI)*, lxrun, WINE, Visual MainWin, and vCUDA, which are summarized in Table 3.4.

Table 3.4 Middleware and Library Support for Virtualization

Middleware or Runtime Library and References or Web Link	Brief Introduction and Application Platforms
WABI (http://docs.sun.com/app/docs/doc/802-6306)	Middleware that converts Windows system calls running on x86 PCs to Solaris system calls running on SPARC workstations
Lxrun (Linux Run) (http://www.ugcs.caltech.edu/~steven/lxrun/)	A system call emulator that enables Linux applications written for x86 hosts to run on UNIX systems such as the SCO OpenServer
WINE (http://www.winehq.org/)	A library support system for virtualizing x86 processors to run Windows applications under Linux, FreeBSD, and Solaris
Visual MainWin (http://www.mainsoft.com/)	A compiler support system to develop Windows applications using Visual Studio to run on Solaris, Linux, and AIX hosts
vCUDA (Example 3.2) (IEEE *IPDPS* 2009 [57])	Virtualization support for using general-purpose GPUs to run data-intensive applications under a special guest OS

The WABI offers middleware to convert Windows system calls to Solaris system calls. Lxrun is really a system call emulator that enables Linux applications written for x86 hosts to run on UNIX systems. Similarly, Wine offers library support for virtualizing x86 processors to run Windows applications on UNIX hosts. Visual MainWin offers a compiler support system to develop Windows applications using Visual Studio to run on some UNIX hosts. The vCUDA is explained in Example 3.2 with a graphical illustration in Figure 3.4.

Example 3.2 The vCUDA for Virtualization of General-Purpose GPUs

CUDA is a programming model and library for general-purpose GPUs. It leverages the high performance of GPUs to run compute-intensive applications on host operating systems. However, it is difficult to run CUDA applications on hardware-level VMs directly. vCUDA virtualizes the CUDA library and can be installed on guest OSes. When CUDA applications run on a guest OS and issue a call to the CUDA API, vCUDA intercepts the call and redirects it to the CUDA API running on the host OS. Figure 3.4 shows the basic concept of the vCUDA architecture [57].

The vCUDA employs a client-server model to implement CUDA virtualization. It consists of three user space components: the vCUDA library, a virtual GPU in the guest OS (which acts as a client), and the vCUDA stub in the host OS (which acts as a server). The vCUDA library resides in the guest OS as a substitute for the standard CUDA library. It is responsible for intercepting and redirecting API calls from the client to the stub. Besides these tasks, vCUDA also creates vGPUs and manages them.

The functionality of a vGPU is threefold: It abstracts the GPU structure and gives applications a uniform view of the underlying hardware; when a CUDA application in the guest OS allocates a device's memory the vGPU can return a local virtual address to the application and notify the remote stub to allocate the real device memory, and the vGPU is responsible for storing the CUDA API flow. The vCUDA stub receives

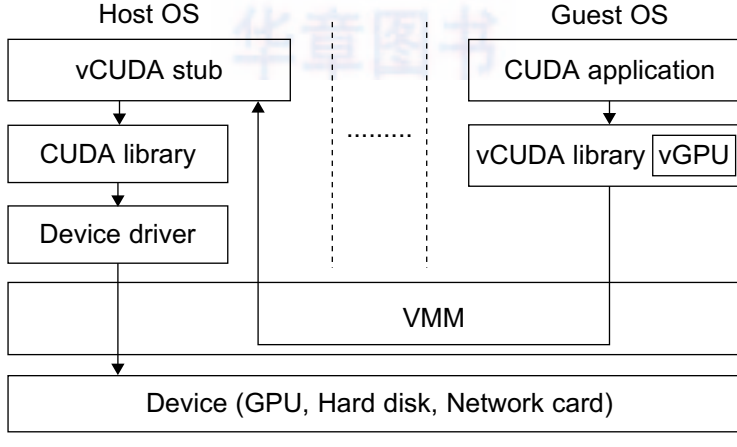


FIGURE 3.4

Basic concept of the vCUDA architecture.

(*Courtesy of Lin Shi, et al. [57]*)

and interprets remote requests and creates a corresponding execution context for the API calls from the guest OS, then returns the results to the guest OS. The vCUDA stub also manages actual physical resource allocation.

3.2 VIRTUALIZATION STRUCTURES/TOOLS AND MECHANISMS

In general, there are three typical classes of VM architecture. Figure 3.1 showed the architectures of a machine before and after virtualization. Before virtualization, the operating system manages the hardware. After virtualization, a virtualization layer is inserted between the hardware and the operating system. In such a case, the virtualization layer is responsible for converting portions of the real hardware into virtual hardware. Therefore, different operating systems such as Linux and Windows can run on the same physical machine, simultaneously. Depending on the position of the virtualization layer, there are several classes of VM architectures, namely the *hypervisor* architecture, *paravirtualization*, and *host-based virtualization*. The *hypervisor* is also known as the VMM (*Virtual Machine Monitor*). They both perform the same virtualization operations.

3.2.1 Hypervisor and Xen Architecture

The hypervisor supports hardware-level virtualization (see Figure 3.1(b)) on bare metal devices like CPU, memory, disk and network interfaces. The hypervisor software sits directly between the physical hardware and its OS. This virtualization layer is referred to as either the VMM or the hypervisor. The hypervisor provides *hypercalls* for the guest OSes and applications. Depending on the functionality, a hypervisor can assume a *micro-kernel architecture* like the Microsoft Hyper-V. Or it can assume a *monolithic hypervisor architecture* like the VMware ESX for server virtualization.

A micro-kernel hypervisor includes only the basic and unchanging functions (such as physical memory management and processor scheduling). The device drivers and other changeable components are outside the hypervisor. A monolithic hypervisor implements all the aforementioned functions, including those of the device drivers. Therefore, the size of the hypervisor code of a micro-kernel hypervisor is smaller than that of a monolithic hypervisor. Essentially, a hypervisor must be able to convert physical devices into virtual resources dedicated for the deployed VM to use.

3.2.1.1 The Xen Architecture

Xen is an open source hypervisor program developed by Cambridge University. Xen is a micro-kernel hypervisor, which separates the policy from the mechanism. The Xen hypervisor implements all the mechanisms, leaving the policy to be handled by Domain 0, as shown in Figure 3.5. Xen does not include any device drivers natively [7]. It just provides a mechanism by which a guest OS can have direct access to the physical devices. As a result, the size of the Xen hypervisor is kept rather small. Xen provides a virtual environment located between the hardware and the OS. A number of vendors are in the process of developing commercial Xen hypervisors, among them are Citrix XenServer [62] and Oracle VM [42].

The core components of a Xen system are the hypervisor, kernel, and applications. The organization of the three components is important. Like other virtualization systems, many guest OSes can run on top of the hypervisor. However, not all guest OSes are created equal, and one in

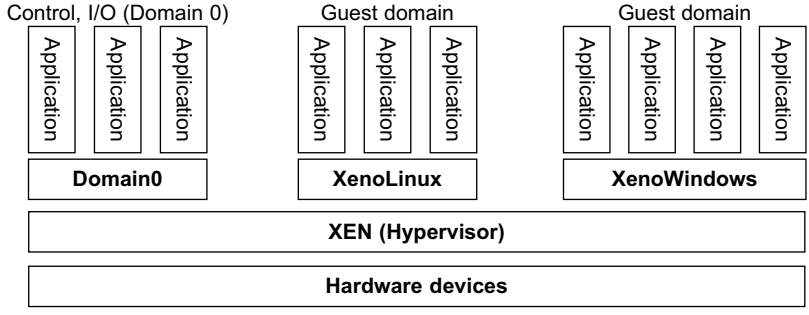


FIGURE 3.5

The Xen architecture's special domain 0 for control and I/O, and several guest domains for user applications. (*Courtesy of P. Barham, et al. [7]*)

particular controls the others. The guest OS, which has control ability, is called Domain 0, and the others are called Domain U. Domain 0 is a privileged guest OS of Xen. It is first loaded when Xen boots without any file system drivers being available. Domain 0 is designed to access hardware directly and manage devices. Therefore, one of the responsibilities of Domain 0 is to allocate and map hardware resources for the guest domains (the Domain U domains).

For example, Xen is based on Linux and its security level is C2. Its management VM is named Domain 0, which has the privilege to manage other VMs implemented on the same host. If Domain 0 is compromised, the hacker can control the entire system. So, in the VM system, security policies are needed to improve the security of Domain 0. Domain 0, behaving as a VMM, allows users to create, copy, save, read, modify, share, migrate, and roll back VMs as easily as manipulating a file, which flexibly provides tremendous benefits for users. Unfortunately, it also brings a series of security problems during the software life cycle and data lifetime.

Traditionally, a machine's lifetime can be envisioned as a straight line where the current state of the machine is a point that progresses monotonically as the software executes. During this time, configuration changes are made, software is installed, and patches are applied. In such an environment, the VM state is akin to a tree: At any point, execution can go into *N* different branches where multiple instances of a VM can exist at any point in this tree at any given time. VMs are allowed to roll back to previous states in their execution (e.g., to fix configuration errors) or rerun from the same point many times (e.g., as a means of distributing dynamic content or circulating a "live" system image).

3.2.2 Binary Translation with Full Virtualization

Depending on implementation technologies, hardware virtualization can be classified into two categories: *full virtualization* and *host-based virtualization*. Full virtualization does not need to modify the host OS. It relies on *binary translation* to trap and to virtualize the execution of certain sensitive, nonvirtualizable instructions. The guest OSes and their applications consist of noncritical and critical instructions. In a host-based system, both a host OS and a guest OS are used. A virtualization software layer is built between the host OS and guest OS. These two classes of VM architecture are introduced next.

3.2.2.1 Full Virtualization

With full virtualization, noncritical instructions run on the hardware directly while critical instructions are discovered and replaced with traps into the VMM to be emulated by software. Both the hypervisor and VMM approaches are considered full virtualization. Why are only critical instructions trapped into the VMM? This is because binary translation can incur a large performance overhead. Noncritical instructions do not control hardware or threaten the security of the system, but critical instructions do. Therefore, running noncritical instructions on hardware not only can promote efficiency, but also can ensure system security.

3.2.2.2 Binary Translation of Guest OS Requests Using a VMM

This approach was implemented by VMware and many other software companies. As shown in Figure 3.6, VMware puts the VMM at Ring 0 and the guest OS at Ring 1. The VMM scans the instruction stream and identifies the privileged, control- and behavior-sensitive instructions. When these instructions are identified, they are trapped into the VMM, which emulates the behavior of these instructions. The method used in this emulation is called *binary translation*. Therefore, full virtualization combines binary translation and direct execution. The guest OS is completely decoupled from the underlying hardware. Consequently, the guest OS is unaware that it is being virtualized.

The performance of full virtualization may not be ideal, because it involves binary translation which is rather time-consuming. In particular, the full virtualization of I/O-intensive applications is a really a big challenge. Binary translation employs a code cache to store translated hot instructions to improve performance, but it increases the cost of memory usage. At the time of this writing, the performance of full virtualization on the x86 architecture is typically 80 percent to 97 percent that of the host machine.

3.2.2.3 Host-Based Virtualization

An alternative VM architecture is to install a virtualization layer on top of the host OS. This host OS is still responsible for managing the hardware. The guest OSes are installed and run on top of the virtualization layer. Dedicated applications may run on the VMs. Certainly, some other applications can also run with the host OS directly. This host-based architecture has some distinct advantages, as enumerated next. First, the user can install this VM architecture without modifying the host OS. The virtualizing software can rely on the host OS to provide device drivers and other low-level services. This will simplify the VM design and ease its deployment.

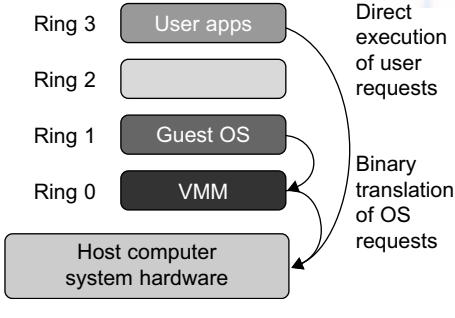


FIGURE 3.6

Indirect execution of complex instructions via binary translation of guest OS requests using the VMM plus direct execution of simple instructions on the same host. (*Courtesy of VM Ware [71]*)

Second, the host-based approach appeals to many host machine configurations. Compared to the hypervisor/VMM architecture, the performance of the host-based architecture may also be low. When an application requests hardware access, it involves four layers of mapping which downgrades performance significantly. When the ISA of a guest OS is different from the ISA of

the underlying hardware, binary translation must be adopted. Although the host-based architecture has flexibility, the performance is too low to be useful in practice.

3.2.3 Para-Virtualization with Compiler Support

Para-virtualization needs to modify the guest operating systems. A para-virtualized VM provides special APIs requiring substantial OS modifications in user applications. Performance degradation is a critical issue of a virtualized system. No one wants to use a VM if it is much slower than using a physical machine. The virtualization layer can be inserted at different positions in a machine software stack. However, para-virtualization attempts to reduce the virtualization overhead, and thus improve performance by modifying only the guest OS kernel.

Figure 3.7 illustrates the concept of a para-virtualized VM architecture. The guest operating systems are para-virtualized. They are assisted by an intelligent compiler to replace the nonvirtualizable OS instructions by hypercalls as illustrated in Figure 3.8. The traditional x86 processor offers four instruction execution rings: Rings 0, 1, 2, and 3. The lower the ring number, the higher the privilege of instruction being executed. The OS is responsible for managing the hardware and the privileged instructions to execute at Ring 0, while user-level applications run at Ring 3. The best example of para-virtualization is the KVM to be described below.

3.2.3.1 Para-Virtualization Architecture

When the x86 processor is virtualized, a virtualization layer is inserted between the hardware and the OS. According to the x86 ring definition, the virtualization layer should also be installed at Ring 0. Different instructions at Ring 0 may cause some problems. In Figure 3.8, we show that para-virtualization replaces nonvirtualizable instructions with *hypercalls* that communicate directly with the hypervisor or VMM. However, when the guest OS kernel is modified for virtualization, it can no longer run on the hardware directly.

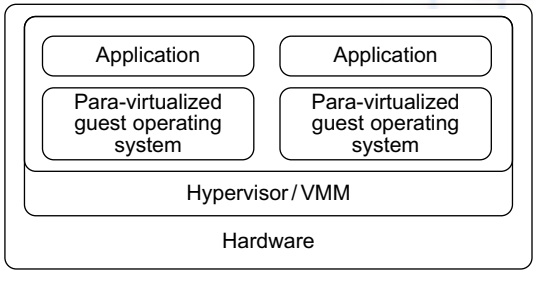


FIGURE 3.7

Para-virtualized VM architecture, which involves modifying the guest OS kernel to replace nonvirtualizable instructions with hypercalls for the hypervisor or the VMM to carry out the virtualization process (See Figure 3.8 for more details.)

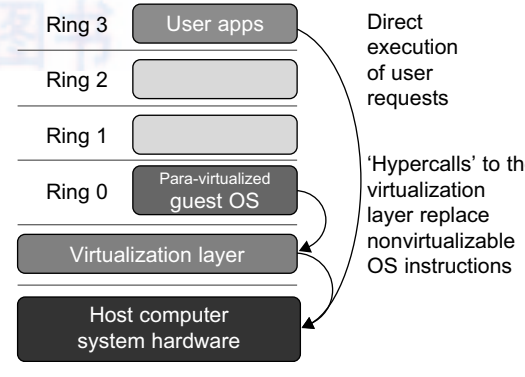


FIGURE 3.8

The use of a para-virtualized guest OS assisted by an intelligent compiler to replace nonvirtualizable OS instructions by hypercalls.

(*Courtesy of VMWare [71]*)

Although para-virtualization reduces the overhead, it has incurred other problems. First, its compatibility and portability may be in doubt, because it must support the unmodified OS as well. Second, the cost of maintaining para-virtualized OSes is high, because they may require deep OS kernel modifications. Finally, the performance advantage of para-virtualization varies greatly due to workload variations. Compared with full virtualization, para-virtualization is relatively easy and more practical. The main problem in full virtualization is its low performance in binary translation. To speed up binary translation is difficult. Therefore, many virtualization products employ the para-virtualization architecture. The popular Xen, KVM, and VMware ESX are good examples.

3.2.3.2 KVM (Kernel-Based VM)

This is a Linux para-virtualization system—a part of the Linux version 2.6.20 kernel. Memory management and scheduling activities are carried out by the existing Linux kernel. The KVM does the rest, which makes it simpler than the hypervisor that controls the entire machine. KVM is a hardware-assisted para-virtualization tool, which improves performance and supports unmodified guest OSes such as Windows, Linux, Solaris, and other UNIX variants.

3.2.3.3 Para-Virtualization with Compiler Support

Unlike the full virtualization architecture which intercepts and emulates privileged and sensitive instructions at runtime, para-virtualization handles these instructions at compile time. The guest OS kernel is modified to replace the privileged and sensitive instructions with hypercalls to the hypervisor or VMM. Xen assumes such a para-virtualization architecture.

The guest OS running in a guest domain may run at Ring 1 instead of at Ring 0. This implies that the guest OS may not be able to execute some privileged and sensitive instructions. The privileged instructions are implemented by hypercalls to the hypervisor. After replacing the instructions with hypercalls, the modified guest OS emulates the behavior of the original guest OS. On an UNIX system, a system call involves an interrupt or service routine. The hypercalls apply a dedicated service routine in Xen.

Example 3.3 VMware ESX Server for Para-Virtualization

VMware pioneered the software market for virtualization. The company has developed virtualization tools for desktop systems and servers as well as virtual infrastructure for large data centers. ESX is a VMM or a hypervisor for bare-metal x86 symmetric multiprocessing (SMP) servers. It accesses hardware resources such as I/O directly and has complete resource management control. An ESX-enabled server consists of four components: a virtualization layer, a resource manager, hardware interface components, and a service console, as shown in Figure 3.9. To improve performance, the ESX server employs a para-virtualization architecture in which the VM kernel interacts directly with the hardware without involving the host OS.

The VMM layer virtualizes the physical hardware resources such as CPU, memory, network and disk controllers, and human interface devices. Every VM has its own set of virtual hardware resources. The resource manager allocates CPU, memory disk, and network bandwidth and maps them to the virtual hardware resource set of each VM created. Hardware interface components are the device drivers and the

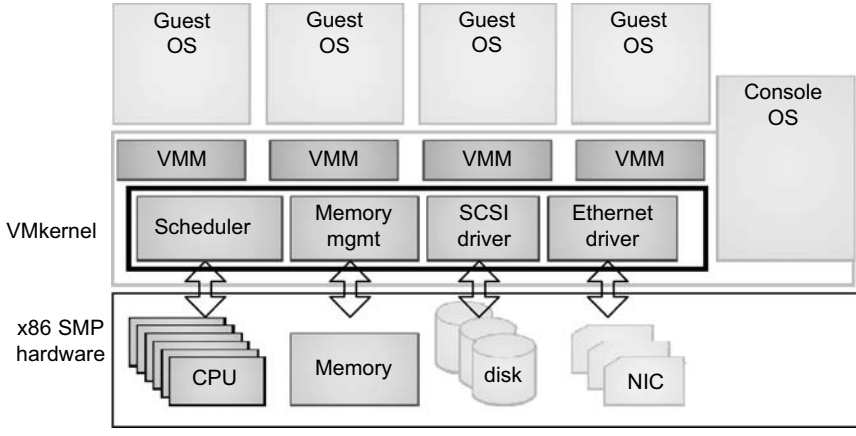


FIGURE 3.9

The VMware ESX server architecture using para-virtualization.

(*Courtesy of VMware [71]*)

VMware ESX Server File System. The service console is responsible for booting the system, initiating the execution of the VMM and resource manager, and relinquishing control to those layers. It also facilitates the process for system administrators.

■

3.3 VIRTUALIZATION OF CPU, MEMORY, AND I/O DEVICES

To support virtualization, processors such as the x86 employ a special running mode and instructions, known as *hardware-assisted virtualization*. In this way, the VMM and guest OS run in different modes and all sensitive instructions of the guest OS and its applications are trapped in the VMM. To save processor states, mode switching is completed by hardware. For the x86 architecture, Intel and AMD have proprietary technologies for hardware-assisted virtualization.

3.3.1 Hardware Support for Virtualization

Modern operating systems and processors permit multiple processes to run simultaneously. If there is no protection mechanism in a processor, all instructions from different processes will access the hardware directly and cause a system crash. Therefore, all processors have at least two modes, user mode and supervisor mode, to ensure controlled access of critical hardware. Instructions running in supervisor mode are called privileged instructions. Other instructions are unprivileged instructions. In a virtualized environment, it is more difficult to make OSes and applications run correctly because there are more layers in the machine stack. Example 3.4 discusses Intel's hardware support approach.

At the time of this writing, many hardware virtualization products were available. The VMware Workstation is a VM software suite for x86 and x86-64 computers. This software suite allows users to set up multiple x86 and x86-64 virtual computers and to use one or more of these VMs simultaneously with the host operating system. The VMware Workstation assumes the host-based virtualization. Xen is a hypervisor for use in IA-32, x86-64, Itanium, and PowerPC 970 hosts. Actually, Xen modifies Linux as the lowest and most privileged layer, or a hypervisor.

One or more guest OS can run on top of the hypervisor. KVM (*Kernel-based Virtual Machine*) is a Linux kernel virtualization infrastructure. KVM can support hardware-assisted virtualization and paravirtualization by using the Intel VT-x or AMD-v and VirtIO framework, respectively. The VirtIO framework includes a paravirtual Ethernet card, a disk I/O controller, a balloon device for adjusting guest memory usage, and a VGA graphics interface using VMware drivers.

Example 3.4 Hardware Support for Virtualization in the Intel x86 Processor

Since software-based virtualization techniques are complicated and incur performance overhead, Intel provides a hardware-assist technique to make virtualization easy and improve performance. Figure 3.10 provides an overview of Intel's full virtualization techniques. For processor virtualization, Intel offers the VT-x or VT-i technique. VT-x adds a privileged mode (VMX Root Mode) and some instructions to processors. This enhancement traps all sensitive instructions in the VMM automatically. For memory virtualization, Intel offers the EPT, which translates the virtual address to the machine's physical addresses to improve performance. For I/O virtualization, Intel implements VT-d and VT-c to support this.

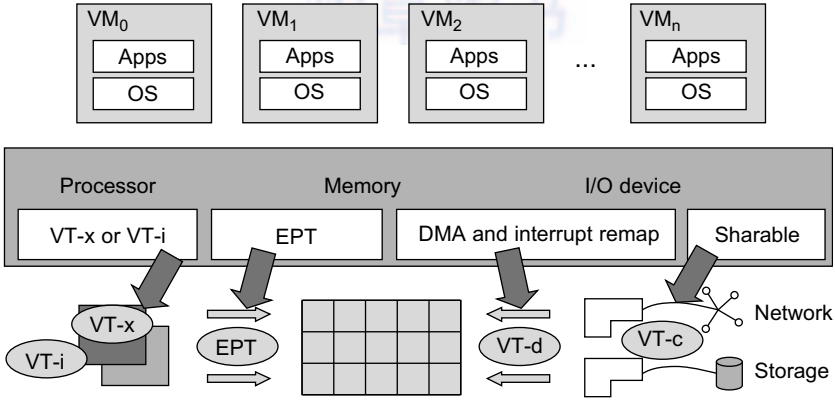


FIGURE 3.10

Intel hardware support for virtualization of processor, memory, and I/O devices.

(*Modified from [68], Courtesy of Lizhong Chen, USC*)

3.3.2 CPU Virtualization

A VM is a duplicate of an existing computer system in which a majority of the VM instructions are executed on the host processor in native mode. Thus, unprivileged instructions of VMs run directly on the host machine for higher efficiency. Other critical instructions should be handled carefully for correctness and stability. The critical instructions are divided into three categories: *privileged instructions*, *control-sensitive instructions*, and *behavior-sensitive instructions*. Privileged instructions execute in a privileged mode and will be trapped if executed outside this mode. Control-sensitive instructions attempt to change the configuration of resources used. Behavior-sensitive instructions have different behaviors depending on the configuration of resources, including the load and store operations over the virtual memory.

A CPU architecture is virtualizable if it supports the ability to run the VM's privileged and unprivileged instructions in the CPU's user mode while the VMM runs in supervisor mode. When the privileged instructions including control- and behavior-sensitive instructions of a VM are executed, they are trapped in the VMM. In this case, the VMM acts as a unified mediator for hardware access from different VMs to guarantee the correctness and stability of the whole system. However, not all CPU architectures are virtualizable. RISC CPU architectures can be naturally virtualized because all control- and behavior-sensitive instructions are privileged instructions. On the contrary, x86 CPU architectures are not primarily designed to support virtualization. This is because about 10 sensitive instructions, such as *SGDT* and *SMSW*, are not privileged instructions. When these instructions execute in virtualization, they cannot be trapped in the VMM.

On a native UNIX-like system, a system call triggers the *80h* interrupt and passes control to the OS kernel. The interrupt handler in the kernel is then invoked to process the system call. On a para-virtualization system such as Xen, a system call in the guest OS first triggers the *80h* interrupt normally. Almost at the same time, the *82h* interrupt in the hypervisor is triggered. Incidentally, control is passed on to the hypervisor as well. When the hypervisor completes its task for the guest OS system call, it passes control back to the guest OS kernel. Certainly, the guest OS kernel may also invoke the hypercall while it's running. Although paravirtualization of a CPU lets unmodified applications run in the VM, it causes a small performance penalty.

3.3.2.1 Hardware-Assisted CPU Virtualization

This technique attempts to simplify virtualization because full or paravirtualization is complicated. Intel and AMD add an additional mode called privilege mode level (some people call it Ring-1) to x86 processors. Therefore, operating systems can still run at Ring 0 and the hypervisor can run at Ring -1. All the privileged and sensitive instructions are trapped in the hypervisor automatically. This technique removes the difficulty of implementing binary translation of full virtualization. It also lets the operating system run in VMs without modification.

Example 3.5 Intel Hardware-Assisted CPU Virtualization

Although x86 processors are not virtualizable primarily, great effort is taken to virtualize them. They are used widely in comparing RISC processors that the bulk of x86-based legacy systems cannot discard easily. Virtualization of x86 processors is detailed in the following sections. Intel's VT-x technology is an example of hardware-assisted virtualization, as shown in Figure 3.11. Intel calls the privilege level of x86 processors the VMX Root Mode. In order to control the start and stop of a VM and allocate a memory page to maintain the

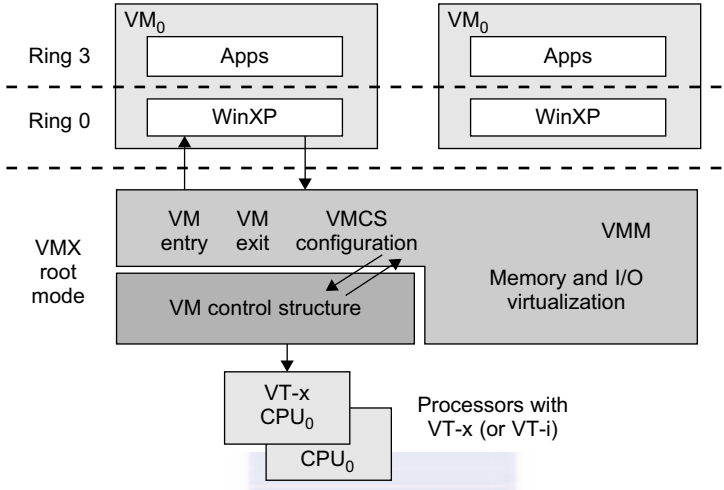


FIGURE 3.11

Intel hardware-assisted CPU virtualization.

(*Modified from [68], Courtesy of Lizhong Chen, USC*)

CPU state for VMs, a set of additional instructions is added. At the time of this writing, Xen, VMware, and the Microsoft Virtual PC all implement their hypervisors by using the VT-x technology.

Generally, hardware-assisted virtualization should have high efficiency. However, since the transition from the hypervisor to the guest OS incurs high overhead switches between processor modes, it sometimes cannot outperform binary translation. Hence, virtualization systems such as VMware now use a hybrid approach, in which a few tasks are offloaded to the hardware but the rest is still done in software. In addition, para-virtualization and hardware-assisted virtualization can be combined to improve the performance further. ■

3.3.3 Memory Virtualization

Virtual memory virtualization is similar to the virtual memory support provided by modern operating systems. In a traditional execution environment, the operating system maintains mappings of *virtual memory* to *machine memory* using page tables, which is a one-stage mapping from virtual memory to machine memory. All modern x86 CPUs include a *memory management unit (MMU)* and a *translation lookaside buffer (TLB)* to optimize virtual memory performance. However, in a virtual execution environment, virtual memory virtualization involves sharing the physical system memory in RAM and dynamically allocating it to the *physical memory* of the VMs.

That means a two-stage mapping process should be maintained by the guest OS and the VMM, respectively: virtual memory to physical memory and physical memory to machine memory. Furthermore, MMU virtualization should be supported, which is transparent to the guest OS. The guest OS continues to control the mapping of virtual addresses to the physical memory addresses of VMs. But the guest OS cannot directly access the actual machine memory. The VMM is responsible for mapping the guest physical memory to the actual machine memory. Figure 3.12 shows the two-level memory mapping procedure.

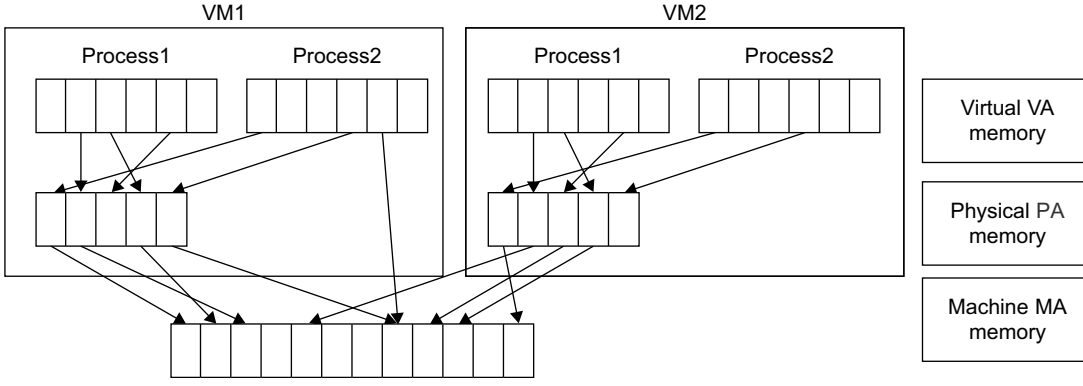


FIGURE 3.12

Two-level memory mapping procedure.

(*Courtesy of R. Rblig, et al. [68]*)

Since each page table of the guest OSes has a separate page table in the VMM corresponding to it, the VMM page table is called the shadow page table. Nested page tables add another layer of indirection to virtual memory. The MMU already handles virtual-to-physical translations as defined by the OS. Then the physical memory addresses are translated to machine addresses using another set of page tables defined by the hypervisor. Since modern operating systems maintain a set of page tables for every process, the shadow page tables will get flooded. Consequently, the performance overhead and cost of memory will be very high.

VMware uses shadow page tables to perform virtual-memory-to-machine-memory address translation. Processors use TLB hardware to map the virtual memory directly to the machine memory to avoid the two levels of translation on every access. When the guest OS changes the virtual memory to a physical memory mapping, the VMM updates the shadow page tables to enable a direct lookup. The AMD Barcelona processor has featured hardware-assisted memory virtualization since 2007. It provides hardware assistance to the two-stage address translation in a virtual execution environment by using a technology called nested paging.

Example 3.6 Extended Page Table by Intel for Memory Virtualization

Since the efficiency of the software shadow page table technique was too low, Intel developed a hardware-based EPT technique to improve it, as illustrated in Figure 3.13. In addition, Intel offers a Virtual Processor ID (VPID) to improve use of the TLB. Therefore, the performance of memory virtualization is greatly improved. In Figure 3.13, the page tables of the guest OS and EPT are all four-level.

When a virtual address needs to be translated, the CPU will first look for the L4 page table pointed to by Guest CR3. Since the address in Guest CR3 is a physical address in the guest OS, the CPU needs to convert the Guest CR3 GPA to the host physical address (HPA) using EPT. In this procedure, the CPU will check the EPT TLB to see if the translation is there. If there is no required translation in the EPT TLB, the CPU will look for it in the EPT. If the CPU cannot find the translation in the EPT, an EPT violation exception will be raised.

When the GPA of the L4 page table is obtained, the CPU will calculate the GPA of the L3 page table by using the GVA and the content of the L4 page table. If the entry corresponding to the GVA in the L4

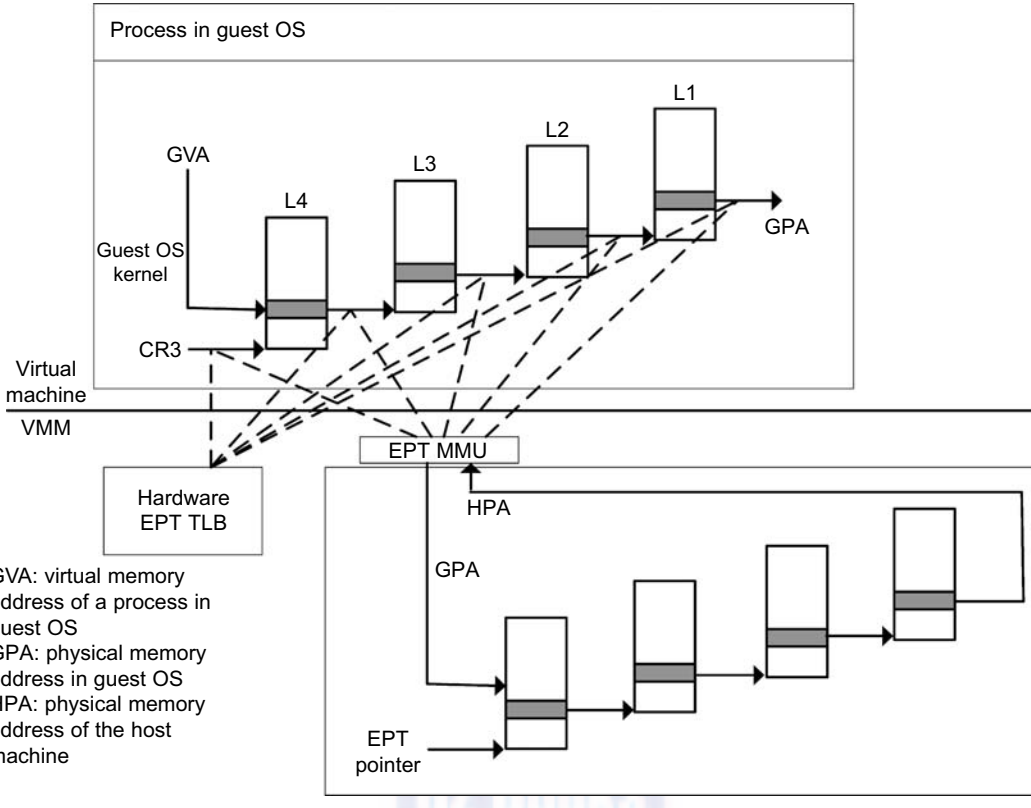


FIGURE 3.13

Memory virtualization using EPT by Intel (the EPT is also known as the shadow page table [68]).

page table is a page fault, the CPU will generate a page fault interrupt and will let the guest OS kernel handle the interrupt. When the PGA of the L3 page table is obtained, the CPU will look for the EPT to get the HPA of the L3 page table, as described earlier. To get the HPA corresponding to a GVA, the CPU needs to look for the EPT five times, and each time, the memory needs to be accessed four times. Therefore, there are 20 memory accesses in the worst case, which is still very slow. To overcome this shortcoming, Intel increased the size of the EPT TLB to decrease the number of memory accesses.

3.3.4 I/O Virtualization

I/O virtualization involves managing the routing of I/O requests between virtual devices and the shared physical hardware. At the time of this writing, there are three ways to implement I/O virtualization: full device emulation, para-virtualization, and direct I/O. Full device emulation is the first approach for I/O virtualization. Generally, this approach emulates well-known, real-world devices.

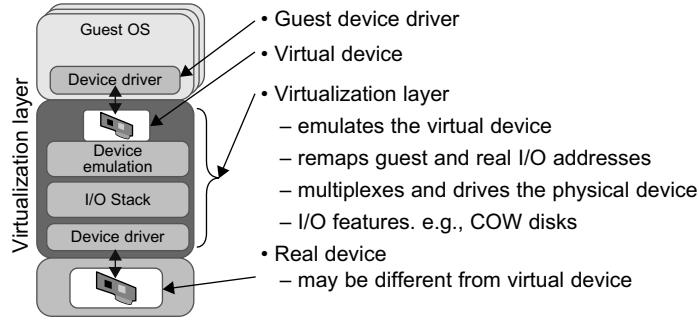


FIGURE 3.14

Device emulation for I/O virtualization implemented inside the middle layer that maps real I/O devices into the virtual devices for the guest device driver to use.

(*Courtesy of V. Chadha, et al. [10] and Y. Dong, et al. [15]*)

All the functions of a device or bus infrastructure, such as device enumeration, identification, interrupts, and DMA, are replicated in software. This software is located in the VMM and acts as a virtual device. The I/O access requests of the guest OS are trapped in the VMM which interacts with the I/O devices. The full device emulation approach is shown in Figure 3.14.

A single hardware device can be shared by multiple VMs that run concurrently. However, software emulation runs much slower than the hardware it emulates [10,15]. The para-virtualization method of I/O virtualization is typically used in Xen. It is also known as the split driver model consisting of a frontend driver and a backend driver. The frontend driver is running in Domain U and the backend driver is running in Domain 0. They interact with each other via a block of shared memory. The frontend driver manages the I/O requests of the guest OSes and the backend driver is responsible for managing the real I/O devices and multiplexing the I/O data of different VMs. Although para-I/O-virtualization achieves better device performance than full device emulation, it comes with a higher CPU overhead.

Direct I/O virtualization lets the VM access devices directly. It can achieve close-to-native performance without high CPU costs. However, current direct I/O virtualization implementations focus on networking for mainframes. There are a lot of challenges for commodity hardware devices. For example, when a physical device is reclaimed (required by workload migration) for later reassignment, it may have been set to an arbitrary state (e.g., DMA to some arbitrary memory locations) that can function incorrectly or even crash the whole system. Since software-based I/O virtualization requires a very high overhead of device emulation, hardware-assisted I/O virtualization is critical. Intel VT-d supports the remapping of I/O DMA transfers and device-generated interrupts. The architecture of VT-d provides the flexibility to support multiple usage models that may run unmodified, special-purpose, or "virtualization-aware" guest OSes.

Another way to help I/O virtualization is via self-virtualized I/O (SV-IO) [47]. The key idea of SV-IO is to harness the rich resources of a multicore processor. All tasks associated with virtualizing an I/O device are encapsulated in SV-IO. It provides virtual devices and an associated access API to VMs and a management API to the VMM. SV-IO defines one virtual interface (VIF) for every kind of virtualized I/O device, such as virtual network interfaces, virtual block devices (disk), virtual camera devices,

and others. The guest OS interacts with the VIFs via VIF device drivers. Each VIF consists of two message queues. One is for outgoing messages to the devices and the other is for incoming messages from the devices. In addition, each VIF has a unique ID for identifying it in SV-IO.

Example 3.7 VMware Workstation for I/O Virtualization

The VMware Workstation runs as an application. It leverages the I/O device support in guest OSes, host OSes, and VMM to implement I/O virtualization. The application portion (VMApp) uses a driver loaded into the host operating system (VMDriver) to establish the privileged VMM, which runs directly on the hardware. A given physical processor is executed in either the host world or the VMM world, with the VMDriver facilitating the transfer of control between the two worlds. The VMware Workstation employs full device emulation to implement I/O virtualization. Figure 3.15 shows the functional blocks used in sending and receiving packets via the emulated virtual NIC.

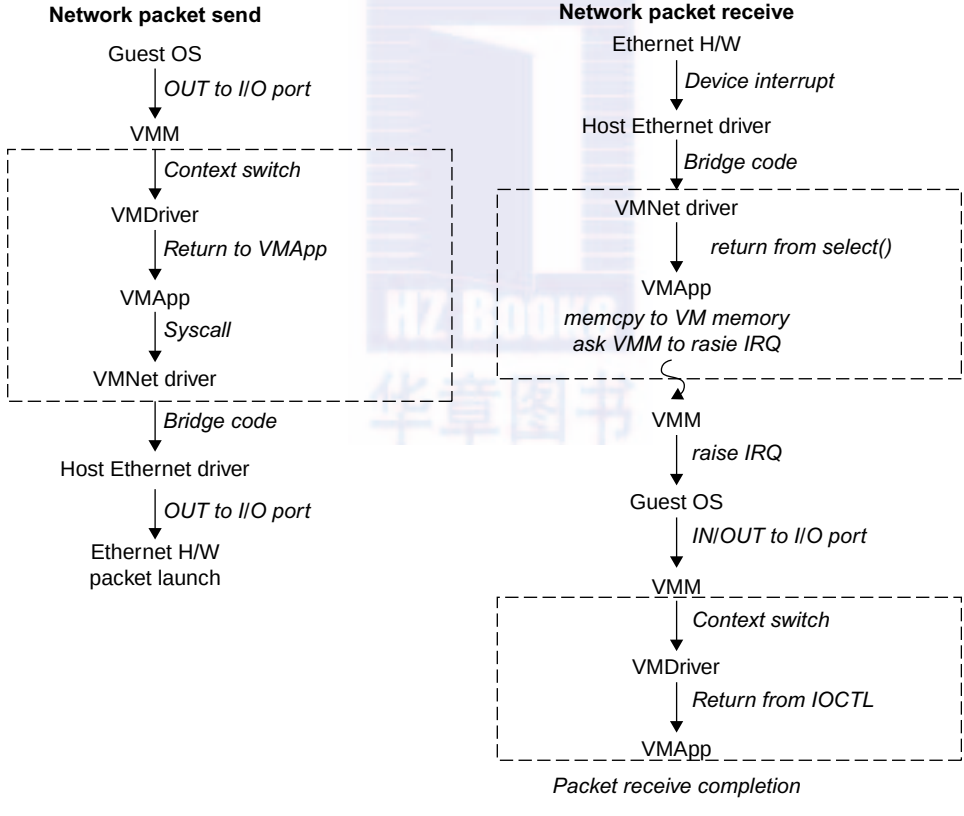


FIGURE 3.15

Functional blocks involved in sending and receiving network packets.

(*Courtesy of VMWare [71]*)

The virtual NIC models an AMD Lance Am79C970A controller. The device driver for a Lance controller in the guest OS initiates packet transmissions by reading and writing a sequence of virtual I/O ports; each read or write switches back to the VMApp to emulate the Lance port accesses. When the last OUT instruction of the sequence is encountered, the Lance emulator calls a normal *write()* to the VMNet driver. The VMNet driver then passes the packet onto the network via a host NIC and then the VMApp switches back to the VMM. The switch raises a virtual interrupt to notify the guest device driver that the packet was sent. Packet receives occur in reverse.

3.3.5 Virtualization in Multi-Core Processors

Virtualizing a multi-core processor is relatively more complicated than virtualizing a uni-core processor. Though multicore processors are claimed to have higher performance by integrating multiple processor cores in a single chip, muti-core virtualiuzation has raised some new challenges to computer architects, compiler constructors, system designers, and application programmers. There are mainly two difficulties: Application programs must be parallelized to use all cores fully, and software must explicitly assign tasks to the cores, which is a very complex problem.

Concerning the first challenge, new programming models, languages, and libraries are needed to make parallel programming easier. The second challenge has spawned research involving scheduling algorithms and resource management policies. Yet these efforts cannot balance well among performance, complexity, and other issues. What is worse, as technology scales, a new challenge called *dynamic heterogeneity* is emerging to mix the fat CPU core and thin GPU cores on the same chip, which further complicates the multi-core or many-core resource management. The dynamic heterogeneity of hardware infrastructure mainly comes from less reliable transistors and increased complexity in using the transistors [33,66].

3.3.5.1 Physical versus Virtual Processor Cores

Wells, et al. [74] proposed a multicore virtualization method to allow hardware designers to get an abstraction of the low-level details of the processor cores. This technique alleviates the burden and inefficiency of managing hardware resources by software. It is located under the ISA and remains unmodified by the operating system or VMM (hypervisor). Figure 3.16 illustrates the technique of a software-visible VCPU moving from one core to another and temporarily suspending execution of a VCPU when there are no appropriate cores on which it can run.

3.3.5.2 Virtual Hierarchy

The emerging many-core *chip multiprocessors* (CMPs) provides a new computing landscape. Instead of supporting time-sharing jobs on one or a few cores, we can use the abundant cores in a space-sharing, where single-threaded or multithreaded jobs are simultaneously assigned to separate groups of cores for long time intervals. This idea was originally suggested by Marty and Hill [39]. To optimize for space-shared workloads, they propose using *virtual hierarchies* to overlay a coherence and caching hierarchy onto a physical processor. Unlike a fixed physical hierarchy, a virtual hierarchy can adapt to fit how the work is space shared for improved performance and performance isolation.

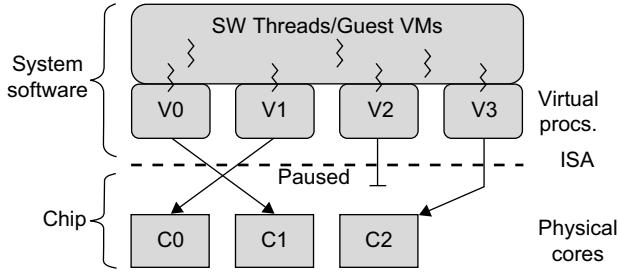


FIGURE 3.16

Multicore virtualization method that exposes four VCPUs to the software, when only three cores are actually present.

(*Courtesy of Wells, et al. [74]*)

Today's many-core CMPs use a physical hierarchy of two or more cache levels that statically determine the cache allocation and mapping. A *virtual hierarchy* is a cache hierarchy that can adapt to fit the workload or mix of workloads [39]. The hierarchy's first level locates data blocks close to the cores needing them for faster access, establishes a shared-cache domain, and establishes a point of coherence for faster communication. When a miss leaves a tile, it first attempts to locate the block (or sharers) within the first level. The first level can also provide isolation between independent workloads. A miss at the L1 cache can invoke the L2 access.

The idea is illustrated in Figure 3.17(a). Space sharing is applied to assign three workloads to three clusters of virtual cores: namely VM0 and VM3 for database workload, VM1 and VM2 for web server workload, and VM4–VM7 for middleware workload. The basic assumption is that each workload runs in its own VM. However, space sharing applies equally within a single operating system. Statically distributing the directory among tiles can do much better, provided operating systems or hypervisors carefully map virtual pages to physical frames. Marty and Hill suggested a two-level virtual coherence and caching hierarchy that harmonizes with the assignment of tiles to the virtual clusters of VMs.

Figure 3.17(b) illustrates a logical view of such a virtual cluster hierarchy in two levels. Each VM operates in a isolated fashion at the first level. This will minimize both miss access time and performance interference with other workloads or VMs. Moreover, the shared resources of cache capacity, inter-connect links, and miss handling are mostly isolated between VMs. The second level maintains a globally shared memory. This facilitates dynamically repartitioning resources without costly cache flushes. Furthermore, maintaining globally shared memory minimizes changes to existing system software and allows virtualization features such as content-based page sharing. A virtual hierarchy adapts to space-shared workloads like multiprogramming and server consolidation. Figure 3.17 shows a case study focused on consolidated server workloads in a tiled architecture. This many-core mapping scheme can also optimize for space-shared multiprogrammed workloads in a single-OS environment.

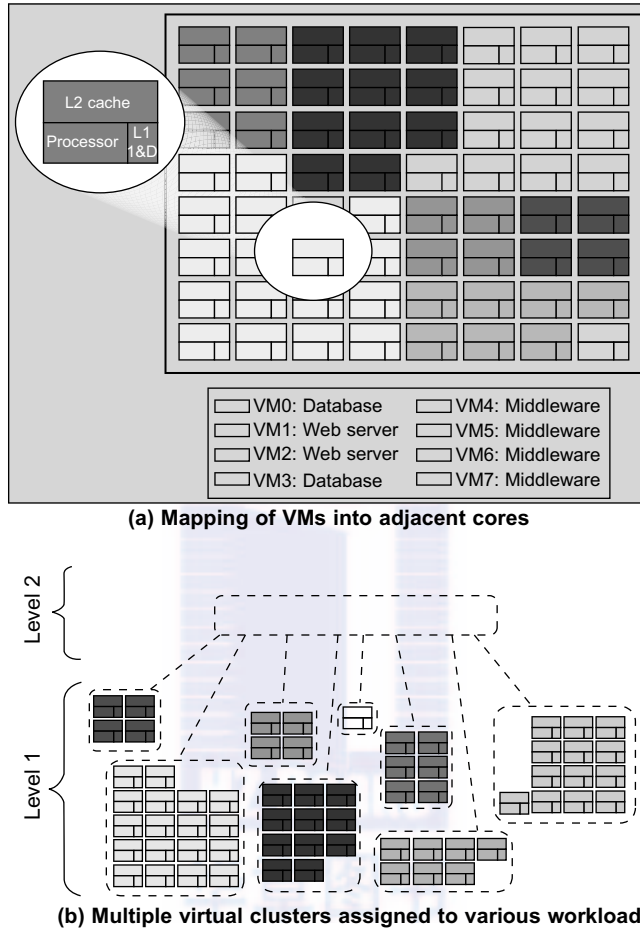


FIGURE 3.17

CMP server consolidation by space-sharing of VMs into many cores forming multiple virtual clusters to execute various workloads.

(*Courtesy of Marty and Hill [39]*)

3.4 VIRTUAL CLUSTERS AND RESOURCE MANAGEMENT

A *physical cluster* is a collection of servers (physical machines) interconnected by a physical network such as a LAN. In Chapter 2, we studied various clustering techniques on physical machines. Here, we introduce virtual clusters and study its properties as well as explore their potential applications. In this section, we will study three critical design issues of virtual clusters: *live migration* of VMs, *memory and file migrations*, and *dynamic deployment* of virtual clusters.

When a traditional VM is initialized, the administrator needs to manually write configuration information or specify the configuration sources. When more VMs join a network, an inefficient configuration always causes problems with overloading or underutilization. Amazon's *Elastic Compute Cloud (EC2)* is a good example of a web service that provides elastic computing power in a cloud. EC2 permits customers to create VMs and to manage user accounts over the time of their use. Most virtualization platforms, including XenServer and VMware ESX Server, support a bridging mode which allows all domains to appear on the network as individual hosts. By using this mode, VMs can communicate with one another freely through the virtual network interface card and configure the network automatically.

3.4.1 Physical versus Virtual Clusters

Virtual clusters are built with VMs installed at distributed servers from one or more physical clusters. The VMs in a virtual cluster are interconnected logically by a virtual network across several physical networks. Figure 3.18 illustrates the concepts of virtual clusters and physical clusters. Each virtual cluster is formed with physical machines or a VM hosted by multiple physical clusters. The virtual cluster boundaries are shown as distinct boundaries.

The provisioning of VMs to a virtual cluster is done dynamically to have the following interesting properties:

- The virtual cluster nodes can be either physical or virtual machines. Multiple VMs running with different OSes can be deployed on the same physical node.
- A VM runs with a guest OS, which is often different from the host OS, that manages the resources in the physical machine, where the VM is implemented.
- The purpose of using VMs is to consolidate multiple functionalities on the same server. This will greatly enhance server utilization and application flexibility.

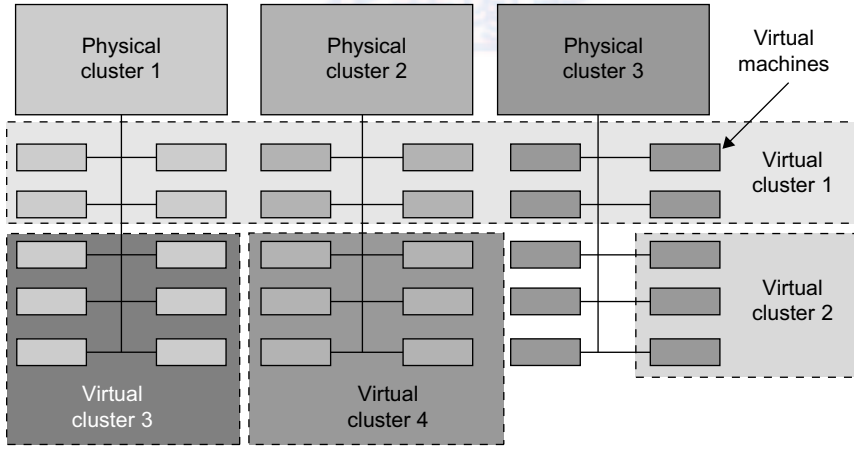


FIGURE 3.18

A cloud platform with four virtual clusters over three physical clusters shaded differently.

(*Courtesy of Fan Zhang, Tsinghua University*)

- VMs can be colonized (replicated) in multiple servers for the purpose of promoting distributed parallelism, fault tolerance, and disaster recovery.
- The size (number of nodes) of a virtual cluster can grow or shrink dynamically, similar to the way an overlay network varies in size in a peer-to-peer (P2P) network.
- The failure of any physical nodes may disable some VMs installed on the failing nodes. But the failure of VMs will not pull down the host system.

Since system virtualization has been widely used, it is necessary to effectively manage VMs running on a mass of physical computing nodes (also called virtual clusters) and consequently build a high-performance virtualized computing environment. This involves virtual cluster deployment, monitoring and management over large-scale clusters, as well as resource scheduling, load balancing, server consolidation, fault tolerance, and other techniques. The different node colors in Figure 3.18 refer to different virtual clusters. In a virtual cluster system, it is quite important to store the large number of VM images efficiently.

Figure 3.19 shows the concept of a virtual cluster based on application partitioning or customization. The different colors in the figure represent the nodes in different virtual clusters. As a large number of VM images might be present, the most important thing is to determine how to store those images in the system efficiently. There are common installations for most users or applications, such as operating systems or user-level programming libraries. These software packages can be preinstalled as templates (called template VMs). With these templates, users can build their own software stacks. New OS instances can be copied from the template VM. User-specific components such as programming libraries and applications can be installed to those instances.

Three physical clusters are shown on the left side of Figure 3.18. Four virtual clusters are created on the right, over the physical clusters. The physical machines are also called *host systems*. In contrast, the VMs are *guest systems*. The host and guest systems may run with different operating

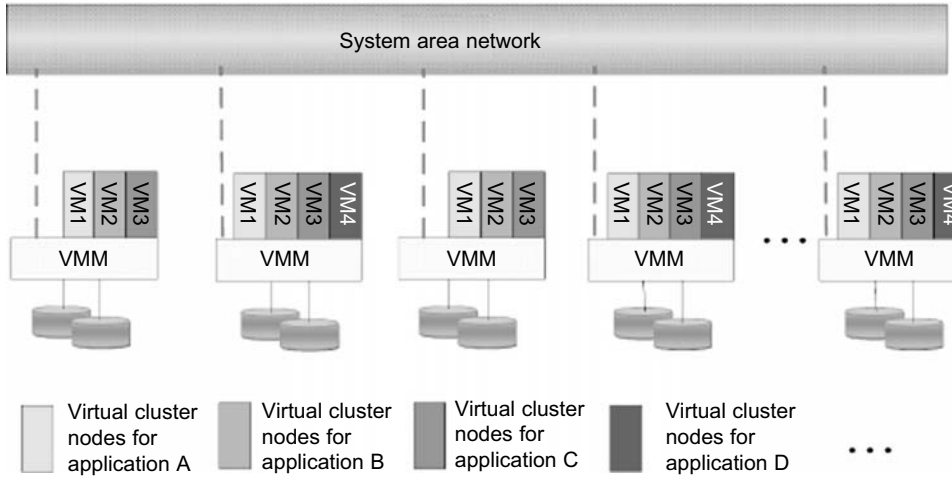


FIGURE 3.19

The concept of a virtual cluster based on application partitioning.

(*Courtesy of Kang Chen, Tsinghua University 2008*)

systems. Each VM can be installed on a remote server or replicated on multiple servers belonging to the same or different physical clusters. The boundary of a virtual cluster can change as VM nodes are added, removed, or migrated dynamically over time.

3.4.1.1 Fast Deployment and Effective Scheduling

The system should have the capability of fast deployment. Here, deployment means two things: to construct and distribute software stacks (OS, libraries, applications) to a physical node inside clusters as fast as possible, and to quickly switch runtime environments from one user's virtual cluster to another user's virtual cluster. If one user finishes using his system, the corresponding virtual cluster should shut down or suspend quickly to save the resources to run other VMs for other users.

The concept of "green computing" has attracted much attention recently. However, previous approaches have focused on saving the energy cost of components in a single workstation without a global vision. Consequently, they do not necessarily reduce the power consumption of the whole cluster. Other cluster-wide energy-efficient techniques can only be applied to homogeneous workstations and specific applications. The live migration of VMs allows workloads of one node to transfer to another node. However, it does not guarantee that VMs can randomly migrate among themselves. In fact, the potential overhead caused by live migrations of VMs cannot be ignored.

The overhead may have serious negative effects on cluster utilization, throughput, and QoS issues. Therefore, the challenge is to determine how to design migration strategies to implement green computing without influencing the performance of clusters. Another advantage of virtualization is load balancing of applications in a virtual cluster. Load balancing can be achieved using the load index and frequency of user logins. The automatic scale-up and scale-down mechanism of a virtual cluster can be implemented based on this model. Consequently, we can increase the resource utilization of nodes and shorten the response time of systems. Mapping VMs onto the most appropriate physical node should promote performance. Dynamically adjusting loads among nodes by live migration of VMs is desired, when the loads on cluster nodes become quite unbalanced.

3.4.1.2 High-Performance Virtual Storage

The template VM can be distributed to several physical hosts in the cluster to customize the VMs. In addition, existing software packages reduce the time for customization as well as switching virtual environments. It is important to efficiently manage the disk spaces occupied by template software packages. Some storage architecture design can be applied to reduce duplicated blocks in a distributed file system of virtual clusters. Hash values are used to compare the contents of data blocks. Users have their own profiles which store the identification of the data blocks for corresponding VMs in a user-specific virtual cluster. New blocks are created when users modify the corresponding data. Newly created blocks are identified in the users' profiles.

Basically, there are four steps to deploy a group of VMs onto a target cluster: *preparing the disk image*, *configuring the VMs*, *choosing the destination nodes*, and *executing the VM deployment command* on every host. Many systems use templates to simplify the disk image preparation process. A template is a disk image that includes a preinstalled operating system with or without certain application software. Users choose a proper template according to their requirements and make a duplicate of it as their own disk image. Templates could implement the *COW (Copy on Write)* format. A new COW backup file is very small and easy to create and transfer. Therefore, it definitely reduces disk space consumption. In addition, VM deployment time is much shorter than that of copying the whole raw image file.

Every VM is configured with a name, disk image, network setting, and allocated CPU and memory. One needs to record each VM configuration into a file. However, this method is inefficient when managing a large group of VMs. VMs with the same configurations could use preedited profiles to simplify the process. In this scenario, the system configures the VMs according to the chosen profile. Most configuration items use the same settings, while some of them, such as UUID, VM name, and IP address, are assigned with automatically calculated values. Normally, users do not care which host is running their VM. A strategy to choose the proper destination host for any VM is needed. The deployment principle is to fulfill the VM requirement and to balance workloads among the whole host network.

3.4.2 Live VM Migration Steps and Performance Effects

In a cluster built with mixed nodes of host and guest systems, the normal method of operation is to run everything on the physical machine. When a VM fails, its role could be replaced by another VM on a different node, as long as they both run with the same guest OS. In other words, a physical node can fail over to a VM on another host. This is different from physical-to-physical failover in a traditional physical cluster. The advantage is enhanced failover flexibility. The potential drawback is that a VM must stop playing its role if its residing host node fails. However, this problem can be mitigated with VM life migration. Figure 3.20 shows the process of life migration of a VM from host A to host B. The migration copies the VM state file from the storage area to the host machine.

There are four ways to manage a virtual cluster. First, you can use a *guest-based manager*, by which the cluster manager resides on a guest system. In this case, multiple VMs form a virtual cluster. For example, openMosix is an open source Linux cluster running different guest systems on top of the Xen hypervisor. Another example is Sun's cluster Oasis, an experimental Solaris cluster of VMs supported by a VMware VMM. Second, you can build a cluster manager on the host systems. The *host-based manager* supervises the guest systems and can restart the guest system on another physical machine. A good example is the VMware HA system that can restart a guest system after failure.

These two cluster management systems are either guest-only or host-only, but they do not mix. A third way to manage a virtual cluster is to use an *independent cluster manager* on both the host and guest systems. This will make infrastructure management more complex, however. Finally, you can use an *integrated cluster* on the guest and host systems. This means the manager must be designed to distinguish between virtualized resources and physical resources. Various cluster management schemes can be greatly enhanced when VM life migration is enabled with minimal overhead.

VMs can be live-migrated from one physical machine to another; in case of failure, one VM can be replaced by another VM. Virtual clusters can be applied in computational grids, cloud platforms, and high-performance computing (HPC) systems. The major attraction of this scenario is that virtual clustering provides dynamic resources that can be quickly put together upon user demand or after a node failure. In particular, virtual clustering plays a key role in cloud computing. When a VM runs a live service, it is necessary to make a trade-off to ensure that the migration occurs in a manner that minimizes all three metrics. The motivation is to design a live VM migration scheme with negligible downtime, the lowest network bandwidth consumption possible, and a reasonable total migration time.

Furthermore, we should ensure that the migration will not disrupt other active services residing in the same host through resource contention (e.g., CPU, network bandwidth). A VM can be in one of the following four states. An *inactive state* is defined by the virtualization platform, under which

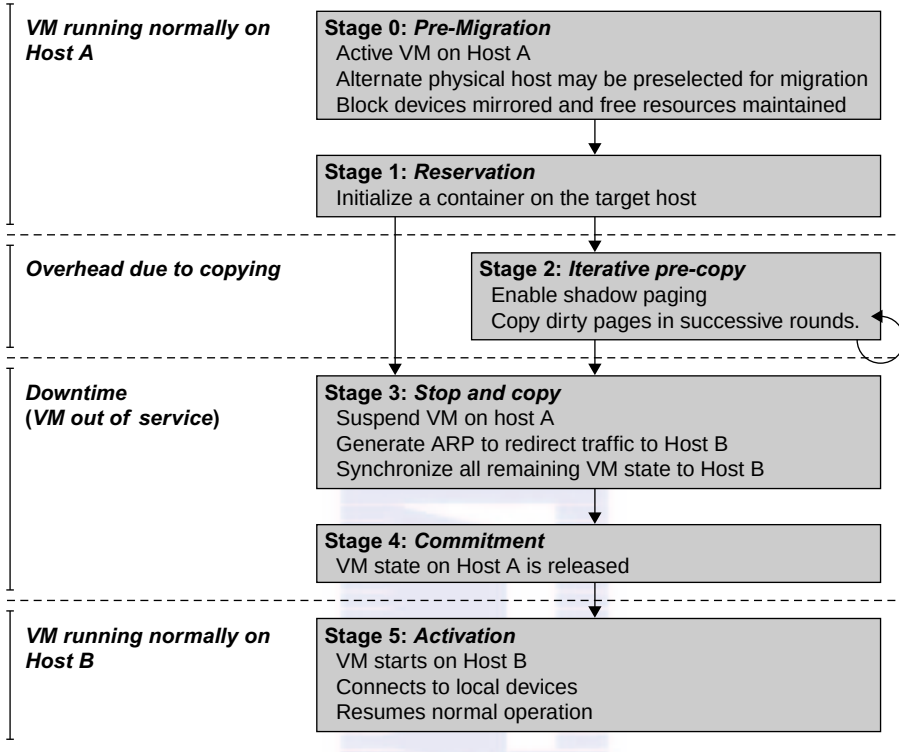


FIGURE 3.20

Live migration process of a VM from one host to another.

(*Courtesy of C. Clark, et al. [14]*)

the VM is not enabled. An *active state* refers to a VM that has been instantiated at the virtualization platform to perform a real task. A *paused state* corresponds to a VM that has been instantiated but disabled to process a task or paused in a waiting state. A VM enters the *suspended state* if its machine file and virtual resources are stored back to the disk. As shown in Figure 3.20, live migration of a VM consists of the following six steps:

Steps 0 and 1: Start migration. This step makes preparations for the migration, including determining the migrating VM and the destination host. Although users could manually make a VM migrate to an appointed host, in most circumstances, the migration is automatically started by strategies such as load balancing and server consolidation.

Steps 2: Transfer memory. Since the whole execution state of the VM is stored in memory, sending the VM's memory to the destination node ensures continuity of the service provided by the VM. All of the memory data is transferred in the first round, and then the migration controller recopies the memory data which is changed in the last round. These steps keep iterating until the dirty portion of the memory is small enough to handle the final copy. Although precopying memory is performed iteratively, the execution of programs is not obviously interrupted.

Step 3: Suspend the VM and copy the last portion of the data. The migrating VM's execution is suspended when the last round's memory data is transferred. Other nonmemory data such as CPU and network states should be sent as well. During this step, the VM is stopped and its applications will no longer run. This "service unavailable" time is called the "downtime" of migration, which should be as short as possible so that it can be negligible to users.

Steps 4 and 5: Commit and activate the new host. After all the needed data is copied, on the destination host, the VM reloads the states and recovers the execution of programs in it, and the service provided by this VM continues. Then the network connection is redirected to the new VM and the dependency to the source host is cleared. The whole migration process finishes by removing the original VM from the source host.

Figure 3.21 shows the effect on the data transmission rate (Mbit/second) of live migration of a VM from one host to another. Before copying the VM with 512 KB files for 100 clients, the data throughput was 870 MB/second. The first precopy takes 63 seconds, during which the rate is reduced to 765 MB/second. Then the data rate reduces to 694 MB/second in 9.8 seconds for more iterations of the copying process. The system experiences only 165 ms of downtime, before the VM is restored at the destination host. This experimental result shows a very small migration overhead in live transfer of a VM between host nodes. This is critical to achieve dynamic cluster reconfiguration and disaster recovery as needed in cloud computing. We will study these techniques in more detail in Chapter 4.

With the emergence of widespread cluster computing more than a decade ago, many cluster configuration and management systems have been developed to achieve a range of goals. These goals naturally influence individual approaches to cluster management. VM technology has become a popular method for simplifying management and sharing of physical computing resources. Platforms

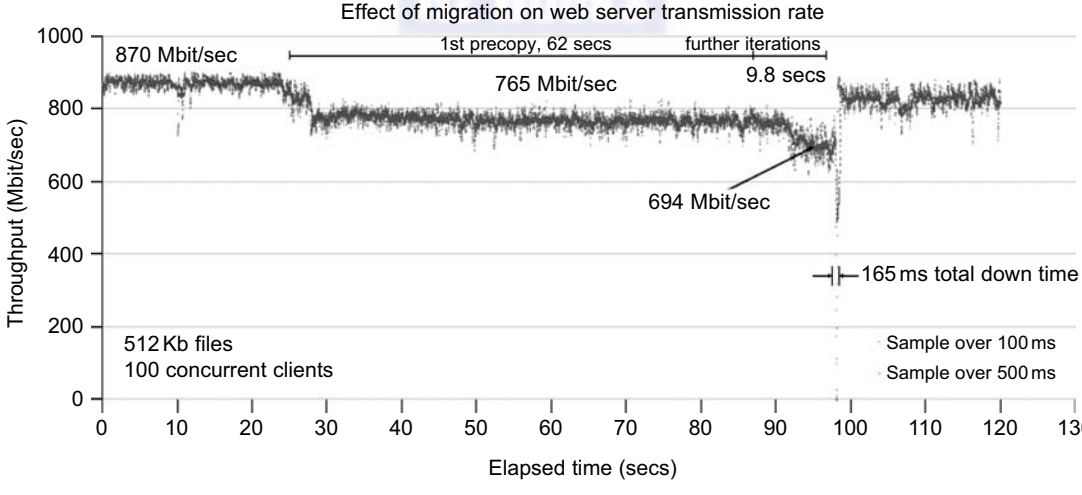


FIGURE 3.21

Effect on data transmission rate of a VM migrated from one failing web server to another.

(*Courtesy of C. Clark, et al. [14]*)

such as VMware and Xen allow multiple VMs with different operating systems and configurations to coexist on the same physical host in mutual isolation. Clustering inexpensive computers is an effective way to obtain reliable, scalable computing power for network services and compute-intensive applications

3.4.3 Migration of Memory, Files, and Network Resources

Since clusters have a high initial cost of ownership, including space, power conditioning, and cooling equipment, leasing or sharing access to a common cluster is an attractive solution when demands vary over time. Shared clusters offer economies of scale and more effective utilization of resources by multiplexing. Early configuration and management systems focus on expressive and scalable mechanisms for defining clusters for specific types of service, and physically partition cluster nodes among those types. When one system migrates to another physical node, we should consider the following issues.

3.4.3.1 Memory Migration

This is one of the most important aspects of VM migration. Moving the memory instance of a VM from one physical host to another can be approached in any number of ways. But traditionally, the concepts behind the techniques tend to share common implementation paradigms. The techniques employed for this purpose depend upon the characteristics of application/workloads supported by the guest OS.

Memory migration can be in a range of hundreds of megabytes to a few gigabytes in a typical system today, and it needs to be done in an efficient manner. The *Internet Suspend-Resume (ISR)* technique exploits temporal locality as memory states are likely to have considerable overlap in the suspended and the resumed instances of a VM. Temporal locality refers to the fact that the memory states differ only by the amount of work done since a VM was last suspended before being initiated for migration.

To exploit temporal locality, each file in the file system is represented as a tree of small subfiles. A copy of this tree exists in both the suspended and resumed VM instances. The advantage of using a tree-based representation of files is that the caching ensures the transmission of only those files which have been changed. The ISR technique deals with situations where the migration of live machines is not a necessity. Predictably, the downtime (the period during which the service is unavailable due to there being no currently executing instance of a VM) is high, compared to some of the other techniques discussed later.

3.4.3.2 File System Migration

To support VM migration, a system must provide each VM with a consistent, location-independent view of the file system that is available on all hosts. A simple way to achieve this is to provide each VM with its own virtual disk which the file system is mapped to and transport the contents of this virtual disk along with the other states of the VM. However, due to the current trend of high-capacity disks, migration of the contents of an entire disk over a network is not a viable solution. Another way is to have a global file system across all machines where a VM could be located. This way removes the need to copy files from one machine to another because all files are network-accessible.

A distributed file system is used in ISR serving as a transport mechanism for propagating a suspended VM state. The actual file systems themselves are not mapped onto the distributed file system. Instead, the VMM only accesses its local file system. The relevant VM files are explicitly copied into the local file system for a resume operation and taken out of the local file system for a suspend operation. This approach relieves developers from the complexities of implementing several different file system calls for different distributed file systems. It also essentially disassociates the VMM from any particular distributed file system semantics. However, this decoupling means that the VMM has to store the contents of each VM's virtual disks in its local files, which have to be moved around with the other state information of that VM.

In smart copying, the VMM exploits spatial locality. Typically, people often move between the same small number of locations, such as their home and office. In these conditions, it is possible to transmit only the difference between the two file systems at suspending and resuming locations. This technique significantly reduces the amount of actual physical data that has to be moved. In situations where there is no locality to exploit, a different approach is to synthesize much of the state at the resuming site. On many systems, user files only form a small fraction of the actual data on disk. Operating system and application software account for the majority of storage space. The proactive state transfer solution works in those cases where the resuming site can be predicted with reasonable confidence.

3.4.3.3 Network Migration

A migrating VM should maintain all open network connections without relying on forwarding mechanisms on the original host or on support from mobility or redirection mechanisms. To enable remote systems to locate and communicate with a VM, each VM must be assigned a virtual IP address known to other entities. This address can be distinct from the IP address of the host machine where the VM is currently located. Each VM can also have its own distinct virtual MAC address. The VMM maintains a mapping of the virtual IP and MAC addresses to their corresponding VMs. In general, a migrating VM includes all the protocol states and carries its IP address with it.

If the source and destination machines of a VM migration are typically connected to a single switched LAN, an unsolicited ARP reply from the migrating host is provided advertising that the IP has moved to a new location. This solves the open network connection problem by reconfiguring all the peers to send future packets to a new location. Although a few packets that have already been transmitted might be lost, there are no other problems with this mechanism. Alternatively, on a switched network, the migrating OS can keep its original Ethernet MAC address and rely on the network switch to detect its move to a new port.

Live migration means moving a VM from one physical node to another while keeping its OS environment and applications unbroken. This capability is being increasingly utilized in today's enterprise environments to provide efficient online system maintenance, reconfiguration, load balancing, and proactive fault tolerance. It provides desirable features to satisfy requirements for computing resources in modern computing systems, including server consolidation, performance isolation, and ease of management. As a result, many implementations are available which support the feature using disparate functionalities. Traditional migration suspends VMs before the transportation and then resumes them at the end of the process. By importing the precopy mechanism, a VM could be live-migrated without stopping the VM and keep the applications running during the migration.

Live migration is a key feature of system virtualization technologies. Here, we focus on VM migration within a cluster environment where a network-accessible storage system, such as *storage*

area network (SAN) or *network attached storage* (NAS), is employed. Only memory and CPU status needs to be transferred from the source node to the target node. Live migration techniques mainly use the precopy approach, which first transfers all memory pages, and then only copies modified pages during the last round iteratively. The VM service downtime is expected to be minimal by using iterative copy operations. When applications' writable working set becomes small, the VM is suspended and only the CPU state and dirty pages in the last round are sent out to the destination.

In the precopy phase, although a VM service is still available, much performance degradation will occur because the migration daemon continually consumes network bandwidth to transfer dirty pages in each round. An adaptive rate limiting approach is employed to mitigate this issue, but total migration time is prolonged by nearly 10 times. Moreover, the maximum number of iterations must be set because not all applications' dirty pages are ensured to converge to a small writable working set over multiple rounds.

In fact, these issues with the precopy approach are caused by the large amount of transferred data during the whole migration process. A checkpointing/recovery and trace/replay approach (CR/TR-Motion) is proposed to provide fast VM migration. This approach transfers the execution trace file in iterations rather than dirty pages, which is logged by a trace daemon. Apparently, the total size of all log files is much less than that of dirty pages. So, total migration time and downtime of migration are drastically reduced. However, CR/TR-Motion is valid only when the log replay rate is larger than the log growth rate. The inequality between source and target nodes limits the application scope of live migration in clusters.

Another strategy of postcopy is introduced for live migration of VMs. Here, all memory pages are transferred only once during the whole migration process and the baseline total migration time is reduced. But the downtime is much higher than that of precopy due to the latency of fetching pages from the source node before the VM can be resumed on the target. With the advent of multicore or many-core machines, abundant CPU resources are available. Even if several VMs reside on a same multicore machine, CPU resources are still rich because physical CPUs are frequently amenable to multiplexing. We can exploit these copious CPU resources to compress page frames and the amount of transferred data can be significantly reduced. Memory compression algorithms typically have little memory overhead. Decompression is simple and very fast and requires no memory for decompression.

3.4.3.4 Live Migration of VM Using Xen

In Section 3.2.1, we studied Xen as a VMM or hypervisor, which allows multiple commodity OSes to share x86 hardware in a safe and orderly fashion. The following example explains how to perform live migration of a VM between two Xen-enabled host machines. Domain 0 (or Dom0) performs tasks to create, terminate, or migrate to another host. Xen uses a send/recv model to transfer states across VMs.

Example 3.8 Live Migration of VMs between Two Xen-Enabled Hosts

Xen supports live migration. It is a useful feature and natural extension to virtualization platforms that allows for the transfer of a VM from one physical machine to another with little or no downtime of the services hosted by the VM. Live migration transfers the working state and memory of a VM across a network when it is running. Xen also supports VM migration by using a mechanism called *Remote Direct Memory Access (RDMA)*.

RDMA speeds up VM migration by avoiding TCP/IP stack processing overhead. RDMA implements a different transfer protocol whose origin and destination VM buffers must be registered before any transfer

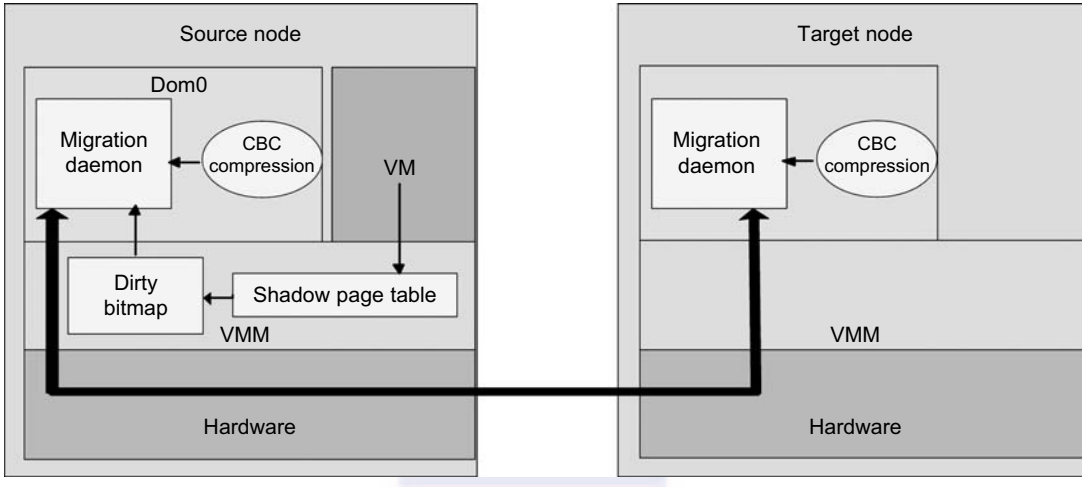


FIGURE 3.22

Live migration of VM from the Dom0 domain to a Xen-enabled target host.

operations occur, reducing it to a "one-sided" interface. Data communication over RDMA does not need to involve the CPU, caches, or context switches. This allows migration to be carried out with minimal impact on guest operating systems and hosted applications. Figure 3.22 shows the a compression scheme for VM migration.

This design requires that we make trade-offs between two factors. If an algorithm embodies expectations about the kinds of regularities in the memory footprint, it must be very fast and effective. A single compression algorithm for all memory data is difficult to achieve the win-win status that we expect. Therefore, it is necessary to provide compression algorithms to pages with different kinds of regularities. The structure of this live migration system is presented in Dom0.

Migration daemons running in the management VMs are responsible for performing migration. Shadow page tables in the VMM layer trace modifications to the memory page in migrated VMs during the precopy phase. Corresponding flags are set in a dirty bitmap. At the start of each precopy round, the bitmap is sent to the migration daemon. Then, the bitmap is cleared and the shadow page tables are destroyed and re-created in the next round. The system resides in Xen's management VM. Memory pages denoted by bitmap are extracted and compressed before they are sent to the destination. The compressed data is then decompressed on the target.

3.4.4 Dynamic Deployment of Virtual Clusters

Table 3.5 summarizes four virtual cluster research projects. We briefly introduce them here just to identify their design objectives and reported results. The Cellular Disco at Stanford is a virtual cluster built in a shared-memory multiprocessor system. The INRIA virtual cluster was built to test parallel algorithm performance. The COD and VIOLIN clusters are studied in forthcoming examples.

Table 3.5 Experimental Results on Four Research Virtual Clusters

Project Name	Design Objectives	Reported Results and References
Cluster-on-Demand at Duke Univ.	Dynamic resource allocation with a virtual cluster management system	Sharing of VMs by multiple virtual clusters using Sun GridEngine [12]
Cellular Disco at Stanford Univ.	To deploy a virtual cluster on a shared-memory multiprocessor	VMs deployed on multiple processors under a VMM called Cellular Disco [8]
VIOLIN at Purdue Univ.	Multiple VM clustering to prove the advantage of dynamic adaptation	Reduce execution time of applications running VIOLIN with adaptation [25,55]
GRAAL Project at INRIA in France	Performance of parallel algorithms in Xen-enabled virtual clusters	75% of max. performance achieved with 30% resource slacks over VM clusters

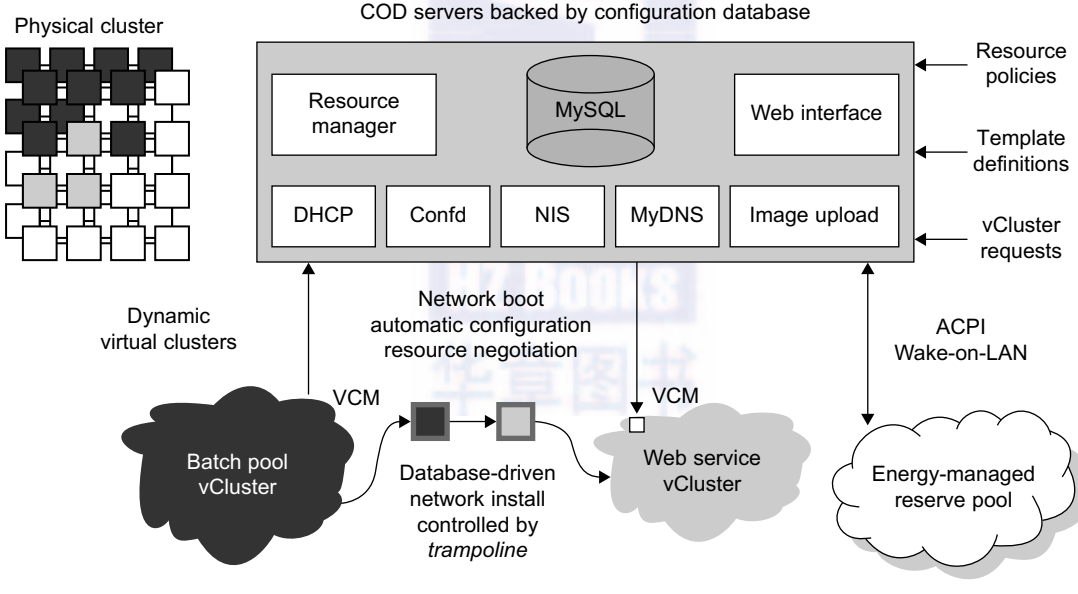


FIGURE 3.23

COD partitioning a physical cluster into multiple virtual clusters.

(*Courtesy of Jeff Chase, et al., HPDC-2003 [12]*)

Example 3.9 The Cluster-on-Demand (COD) Project at Duke University

Developed by researchers at Duke University, the COD (*Cluster-on-Demand*) project is a virtual cluster management system for dynamic allocation of servers from a computing pool to multiple virtual clusters [12]. The idea is illustrated by the prototype implementation of the COD shown in Figure 3.23. The COD

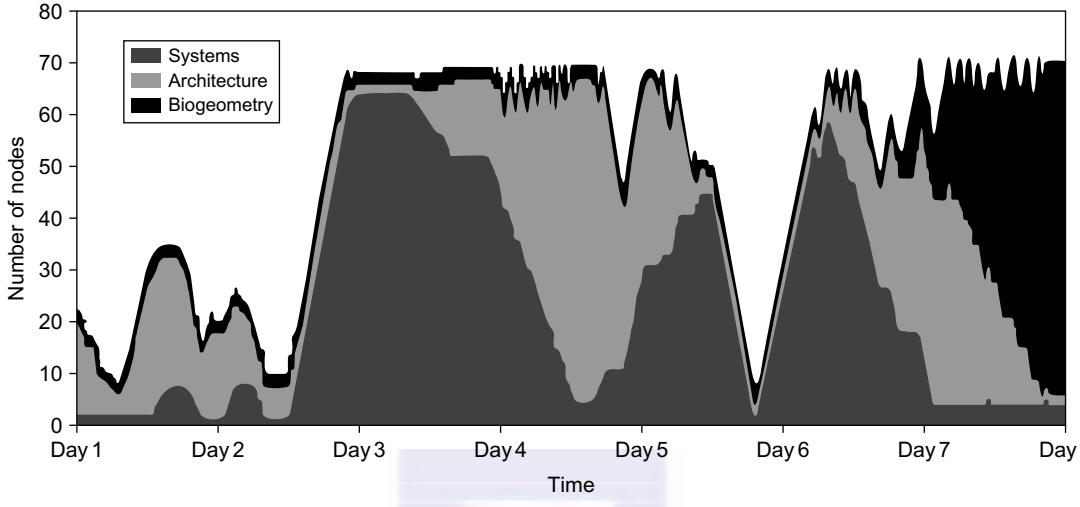


FIGURE 3.24

Cluster size variations in COD over eight days at Duke University.

(*Courtesy of J. Chase, et al. [12]*)

partitions a physical cluster into multiple virtual clusters (*vClusters*). vCluster owners specify the operating systems and software for their clusters through an XML-RPC interface. The vClusters run a batch schedule from Sun's GridEngine on a web server cluster. The COD system can respond to load changes in restructuring the virtual clusters dynamically.

The Duke researchers used the Sun GridEngine scheduler to demonstrate that dynamic virtual clusters are an enabling abstraction for advanced resource management in computing utilities such as grids. The system supports dynamic, policy-based cluster sharing between local users and hosted grid services. Attractive features include resource reservation, adaptive provisioning, scavenging of idle resources, and dynamic instantiation of grid services. The COD servers are backed by a configuration database. This system provides resource policies and template definition in response to user requests.

Figure 3.24 shows the variation in the number of nodes in each of three virtual clusters during eight days of a live deployment. Three application workloads requested by three user groups are labeled "Systems," "Architecture," and "BioGeometry" in the trace plot. The experiments were performed with multiple SGE batch pools on a test bed of 80 rack-mounted IBM xSeries-335 servers within the Duke cluster. This trace plot clearly shows the sharp variation in cluster size (number of nodes) over the eight days. Dynamic provisioning and deprovisioning of virtual clusters are needed in real-life cluster applications.

Example 3.10 The VIOLIN Project at Purdue University

The Purdue VIOLIN Project applies live VM migration to reconfigure a virtual cluster environment. Its purpose is to achieve better resource utilization in executing multiple cluster jobs on multiple cluster

domains. The project leverages the maturity of VM migration and environment adaptation technology. The approach is to enable mutually isolated virtual environments for executing parallel applications on top of a shared physical infrastructure consisting of multiple domains. Figure 3.25 illustrates the idea with five concurrent virtual environments, labeled as VIOLIN 1–5, sharing two physical clusters.

The squares of various shadings represent the VMs deployed in the physical server nodes. The major contribution by the Purdue group is to achieve autonomic adaptation of the virtual computation environments as active, integrated entities. A virtual execution environment is able to relocate itself across the infrastructure, and can scale its share of infrastructural resources. The adaptation is transparent to both users of virtual environments and administrations of infrastructures. The adaptation overhead is maintained at 20 sec out of 1,200 sec in solving a large NEMO3D problem of 1 million particles.

The message being conveyed here is that the virtual environment adaptation can enhance resource utilization significantly at the expense of less than 1 percent of an increase in total execution time. The

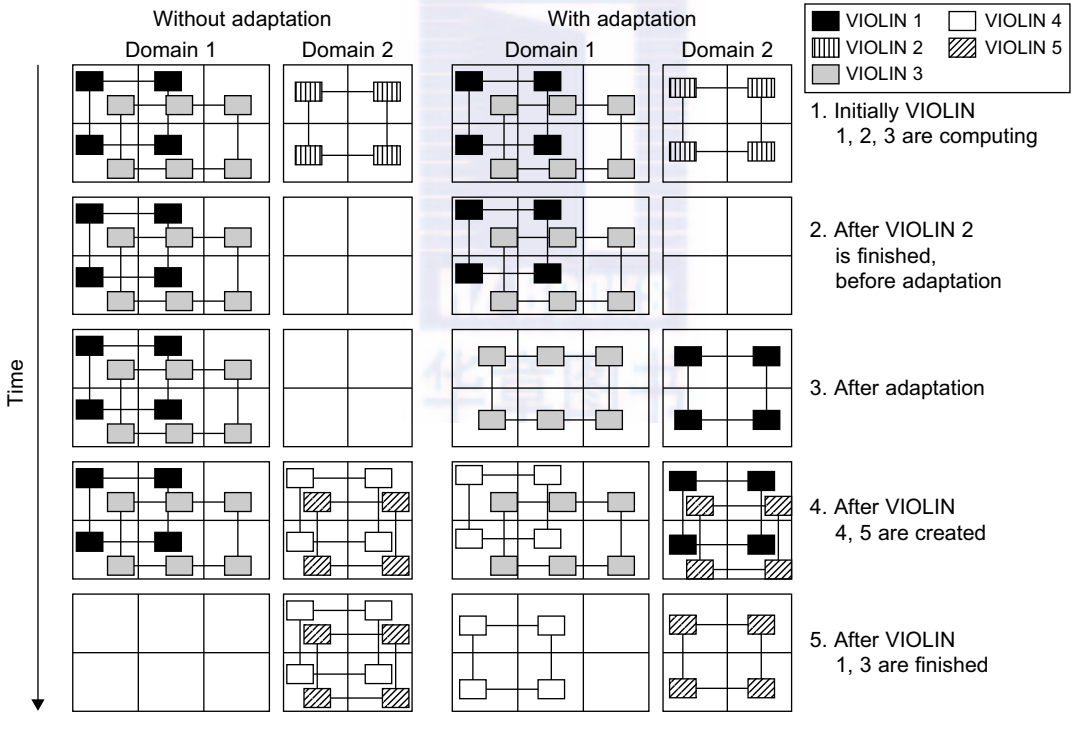


FIGURE 3.25

VIOLIN adaptation scenario of five virtual environments sharing two hosted clusters. Note that there are more idle squares (blank nodes) before and after the adaptation.

(*Courtesy of P. Ruth, et al. [55]*)

migration of VIOLIN environments does pay off. Of course, the gain in shared resource utilization will benefit many users, and the performance gain varies with different adaptation scenarios. We leave readers to trace the execution of another scenario in Problem 3.17 at the end of this chapter to tell the differences. Virtual networking is a fundamental component of the VIOLIN system.

3.5 VIRTUALIZATION FOR DATA-CENTER AUTOMATION

Data centers have grown rapidly in recent years, and all major IT companies are pouring their resources into building new data centers. In addition, Google, Yahoo!, Amazon, Microsoft, HP, Apple, and IBM are all in the game. All these companies have invested billions of dollars in data-center construction and automation. Data-center automation means that huge volumes of hardware, software, and database resources in these data centers can be allocated dynamically to millions of Internet users simultaneously, with guaranteed QoS and cost-effectiveness.

This automation process is triggered by the growth of virtualization products and cloud computing services. From 2006 to 2011, according to an IDC 2007 report on the growth of virtualization and its market distribution in major IT sectors. In 2006, virtualization has a market share of $1,044 million in business and enterprise opportunities. The majority was dominated by production consolidation and software development. Virtualization is moving towards enhancing mobility, reducing planned downtime (for maintenance), and increasing the number of virtual clients.

The latest virtualization development highlights *high availability* (HA), backup services, workload balancing, and further increases in client bases. IDC projected that automation, service orientation, policy-based, and variable costs in the virtualization market. The total business opportunities may increase to $3.2 billion by 2011. The major market share moves to the areas of HA, utility computing, production consolidation, and client bases. In what follows, we will discuss server consolidation, virtual storage, OS support, and trust management in automated data-center designs.

3.5.1 Server Consolidation in Data Centers

In data centers, a large number of heterogeneous workloads can run on servers at various times. These heterogeneous workloads can be roughly divided into two categories: chatty workloads and noninteractive workloads. Chatty workloads may burst at some point and return to a silent state at some other point. A web video service is an example of this, whereby a lot of people use it at night and few people use it during the day. Noninteractive workloads do not require people's efforts to make progress after they are submitted. High-performance computing is a typical example of this. At various stages, the requirements for resources of these workloads are dramatically different. However, to guarantee that a workload will always be able to cope with all demand levels, the workload is statically allocated enough resources so that peak demand is satisfied.

Therefore, it is common that most servers in data centers are underutilized. A large amount of hardware, space, power, and management cost of these servers is wasted. Server consolidation is an approach to improve the low utility ratio of hardware resources by reducing the number of physical servers. Among several server consolidation techniques such as centralized and physical consolidation, virtualization-based server consolidation is the most powerful. Data centers need to optimize their resource management. Yet these techniques are performed with the granularity of a full server machine, which makes resource management far from well optimized. Server virtualization enables smaller resource allocation than a physical machine.

In general, the use of VMs increases resource management complexity. This causes a challenge in terms of how to improve resource utilization as well as guarantee QoS in data centers. In detail, server virtualization has the following side effects:

- Consolidation enhances hardware utilization. Many underutilized servers are consolidated into fewer servers to enhance resource utilization. Consolidation also facilitates backup services and disaster recovery.
- This approach enables more agile provisioning and deployment of resources. In a virtual environment, the images of the guest OSes and their applications are readily cloned and reused.
- The total cost of ownership is reduced. In this sense, server virtualization causes deferred purchases of new servers, a smaller data-center footprint, lower maintenance costs, and lower power, cooling, and cabling requirements.
- This approach improves availability and business continuity. The crash of a guest OS has no effect on the host OS or any other guest OS. It becomes easier to transfer a VM from one server to another, because virtual servers are unaware of the underlying hardware.

To automate data-center operations, one must consider resource scheduling, architectural support, power management, automatic or autonomic resource management, performance of analytical models, and so on. In virtualized data centers, an efficient, on-demand, fine-grained scheduler is one of the key factors to improve resource utilization. Scheduling and reallocations can be done in a wide range of levels in a set of data centers. The levels match at least at the VM level, server level, and data-center level. Ideally, scheduling and resource reallocations should be done at all levels. However, due to the complexity of this, current techniques only focus on a single level or, at most, two levels.

Dynamic CPU allocation is based on VM utilization and application-level QoS metrics. One method considers both CPU and memory flowing as well as automatically adjusting resource overhead based on varying workloads in hosted services. Another scheme uses a two-level resource management system to handle the complexity involved. A local controller at the VM level and a global controller at the server level are designed. They implement autonomic resource allocation via the interaction of the local and global controllers. Multicore and virtualization are two cutting techniques that can enhance each other.

However, the use of CMP is far from well optimized. The memory system of CMP is a typical example. One can design a virtual hierarchy on a CMP in data centers. One can consider protocols that minimize the memory access time, inter-VM interferences, facilitating VM reassignment, and supporting inter-VM sharing. One can also consider a VM-aware power budgeting scheme using multiple managers integrated to achieve better power management. The power budgeting policies cannot ignore the heterogeneity problems. Consequently, one must address the trade-off of power saving and data-center performance.

3.5.2 Virtual Storage Management

The term "storage virtualization" was widely used before the renaissance of system virtualization. Yet the term has a different meaning in a system virtualization environment. Previously, storage virtualization was largely used to describe the aggregation and repartitioning of disks at very coarse time scales for use by physical machines. In system virtualization, virtual storage includes the storage managed by VMMs and guest OSes. Generally, the data stored in this environment can be classified into two categories: VM images and application data. The VM images are special to the virtual environment, while application data includes all other data which is the same as the data in traditional OS environments.

The most important aspects of system virtualization are encapsulation and isolation. Traditional operating systems and applications running on them can be encapsulated in VMs. Only one operating system runs in a virtualization while many applications run in the operating system. System virtualization allows multiple VMs to run on a physical machine and the VMs are completely isolated. To achieve encapsulation and isolation, both the system software and the hardware platform, such as CPUs and chipsets, are rapidly updated. However, storage is lagging. The storage systems become the main bottleneck of VM deployment.

In virtualization environments, a virtualization layer is inserted between the hardware and traditional operating systems or a traditional operating system is modified to support virtualization. This procedure complicates storage operations. On the one hand, storage management of the guest OS performs as though it is operating in a real hard disk while the guest OSes cannot access the hard disk directly. On the other hand, many guest OSes contest the hard disk when many VMs are running on a single physical machine. Therefore, storage management of the underlying VMM is much more complex than that of guest OSes (traditional OSes).

In addition, the storage primitives used by VMs are not nimble. Hence, operations such as remapping volumes across hosts and checkpointing disks are frequently clumsy and esoteric, and sometimes simply unavailable. In data centers, there are often thousands of VMs, which cause the VM images to become flooded. Many researchers tried to solve these problems in virtual storage management. The main purposes of their research are to make management easy while enhancing performance and reducing the amount of storage occupied by the VM images. Parallax is a distributed storage system customized for virtualization environments. Content Addressable Storage (CAS) is a solution to reduce the total size of VM images, and therefore supports a large set of VM-based systems in data centers.

Since traditional storage management techniques do not consider the features of storage in virtualization environments, Parallax designs a novel architecture in which storage features that have traditionally been implemented directly on high-end storage arrays and switchers are relocated into a federation of storage VMs. These storage VMs share the same physical hosts as the VMs that they serve. Figure 3.26 provides an overview of the Parallax system architecture. It supports all popular system virtualization techniques, such as paravirtualization and full virtualization. For each physical machine, Parallax customizes a special storage appliance VM. The storage appliance VM acts as a block virtualization layer between individual VMs and the physical storage device. It provides a virtual disk for each VM on the same physical machine.

Example 3.11 Parallax Providing Virtual Disks to Client VMs from a Large Common Shared Physical Disk

The architecture of Parallax is scalable and especially suitable for use in cluster-based environments. Figure 3.26 shows a high-level view of the structure of a Parallax-based cluster. A cluster-wide administrative domain manages all storage appliance VMs, which makes storage management easy. The storage appliance

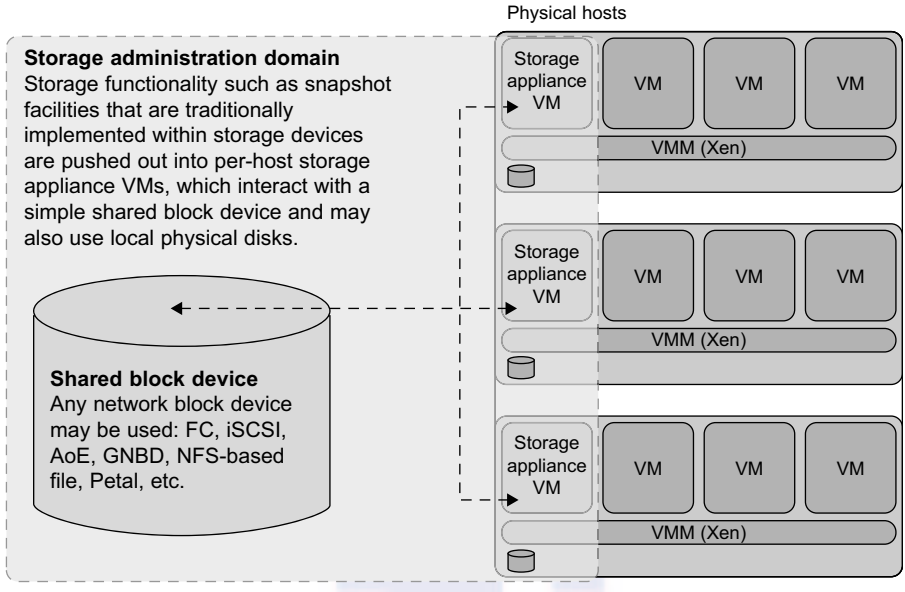


FIGURE 3.26

Parallax is a set of per-host storage appliances that share access to a common block device and presents virtual disks to client VMs.

(*Courtesy of D. Meyer, et al. [43]*)

VM also allows functionality that is currently implemented within data-center hardware to be pushed out and implemented on individual hosts. This mechanism enables advanced storage features such as snapshot facilities to be implemented in software and delivered above commodity network storage targets.

Parallax itself runs as a user-level application in the storage appliance VM. It provides *virtual disk images (VDIs)* to VMs. A VDI is a single-writer virtual disk which may be accessed in a location-transparent manner from any of the physical hosts in the Parallax cluster. The VDIs are the core abstraction provided by Parallax. Parallax uses Xen's block tap driver to handle block requests and it is implemented as a tapdisk library. This library acts as a single block virtualization service for all client VMs on the same physical host. In the Parallax system, it is the storage appliance VM that connects the physical hardware device for block and network access. As shown in Figure 3.30, physical device drivers are included in the storage appliance VM. This implementation enables a storage administrator to live-upgrade the block device drivers in an active cluster.

3.5.3 Cloud OS for Virtualized Data Centers

Data centers must be virtualized to serve as cloud providers. Table 3.6 summarizes four *virtual infrastructure (VI)* managers and OSes. These VI managers and OSes are specially tailored for virtualizing data centers which often own a large number of servers in clusters. Nimbus, Eucalyptus,

Table 3.6 VI Managers and Operating Systems for Virtualizing Data Centers [9]

Manager/ OS, Platforms, License	Resources Being Virtualized, Web Link	Client API, Language	Hypervisors Used	Public Cloud Interface	Special Features
Nimbus Linux, Apache v2	VM creation, virtual cluster, www .nimbusproject.org/	EC2 WS, WSRF, CLI	Xen, KVM	EC2	Virtual networks
Eucalyptus Linux, BSD	Virtual networking (Example 3.12 and [41]), www .eucalyptus.com/	EC2 WS, CLI	Xen, KVM	EC2	Virtual networks
OpenNebula Linux, Apache v2	Management of VM, host, virtual network, and scheduling tools, www.opennebula.org/	XML-RPC, CLI, Java	Xen, KVM	EC2, Elastic Host	Virtual networks, dynamic provisioning
vSphere 4 Linux, Windows, proprietary	Virtualizing OS for data centers (Example 3.13), www .vmware.com/ products/vsphere/ [66]	CLI, GUI, Portal, WS	VMware ESX, ESXi	VMware vCloud partners	Data protection, vStorage, VMFS, DRM, HA

and OpenNebula are all open source software available to the general public. Only vSphere 4 is a proprietary OS for cloud resource virtualization and management over data centers.

These VI managers are used to create VMs and aggregate them into virtual clusters as elastic resources. Nimbus and Eucalyptus support essentially virtual networks. OpenNebula has additional features to provision dynamic resources and make advance reservations. All three public VI managers apply Xen and KVM for virtualization. vSphere 4 uses the hypervisors ESX and ESXi from VMware. Only vSphere 4 supports virtual storage in addition to virtual networking and data protection. We will study Eucalyptus and vSphere 4 in the next two examples.

Example 3.12 Eucalyptus for Virtual Networking of Private Cloud

Eucalyptus is an open source software system (Figure 3.27) intended mainly for supporting Infrastructure as a Service (IaaS) clouds. The system primarily supports virtual networking and the management of VMs; virtual storage is not supported. Its purpose is to build private clouds that can interact with end users through Ethernet or the Internet. The system also supports interaction with other private clouds or public clouds over the Internet. The system is short on security and other desired features for general-purpose grid or cloud applications.

The designers of Eucalyptus [45] implemented each high-level system component as a stand-alone web service. Each web service exposes a well-defined language-agnostic API in the form of a WSDL document containing both operations that the service can perform and input/output data structures.

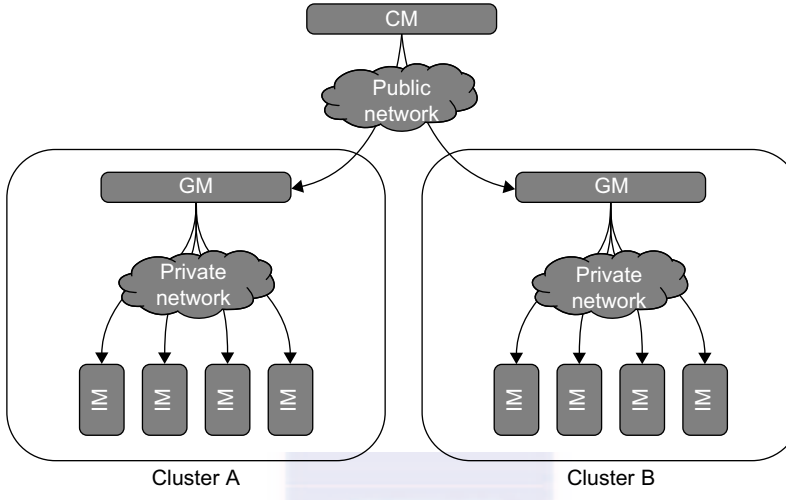


FIGURE 3.27

Eucalyptus for building private clouds by establishing virtual networks over the VMs linking through Ethernet and the Internet.

(*Courtesy of D. Nurmi, et al. [45]*)

Furthermore, the designers leverage existing web-service features such as WS-Security policies for secure communication between components. The three resource managers in Figure 3.27 are specified below:

- **Instance Manager** controls the execution, inspection, and terminating of VM instances on the host where it runs.
- **Group Manager** gathers information about and schedules VM execution on specific instance managers, as well as manages virtual instance network.
- **Cloud Manager** is the entry-point into the cloud for users and administrators. It queries node managers for information about resources, makes scheduling decisions, and implements them by making requests to group managers.

In terms of functionality, Eucalyptus works like AWS APIs. Therefore, it can interact with EC2. It does provide a storage API to emulate the Amazon S3 API for storing user data and VM images. It is installed on Linux-based platforms, is compatible with EC2 with SOAP and Query, and is S3-compatible with SOAP and REST. CLI and web portal services can be applied with Eucalyptus.

Example 3.13 VMware vSphere 4 as a Commercial Cloud OS [66]

The vSphere 4 offers a hardware and software ecosystem developed by VMware and released in April 2009. vSphere extends earlier virtualization software products by VMware, namely the VMware Workstation, ESX for server virtualization, and Virtual Infrastructure for server clusters. Figure 3.28 shows vSphere's

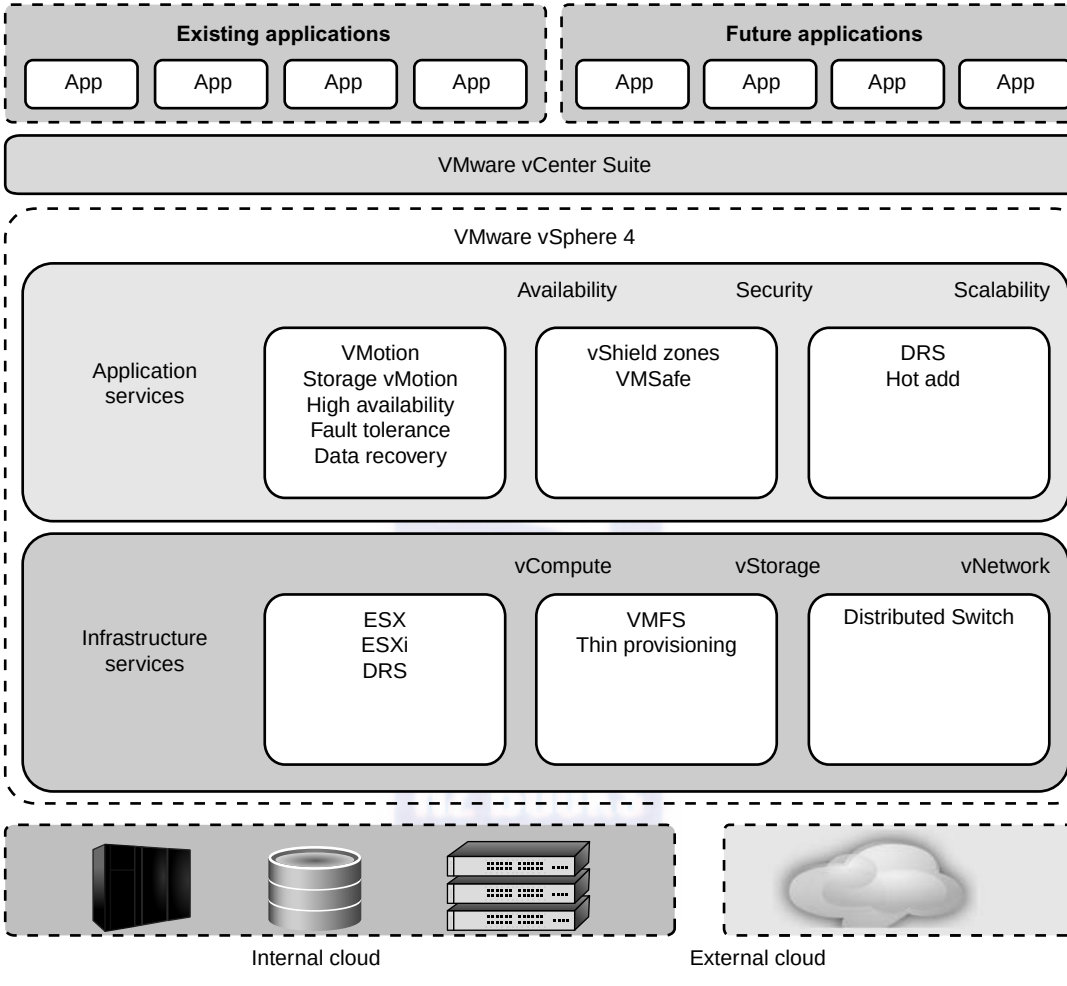


FIGURE 3.28

vSphere/4, a cloud operating system that manages compute, storage, and network resources over virtualized data centers.

(*Courtesy of VMware, April 2010 [72]*)

overall architecture. The system interacts with user applications via an interface layer, called *vCenter*. vSphere is primarily intended to offer virtualization support and resource management of data-center resources in building private clouds. VMware claims the system is the first cloud OS that supports availability, security, and scalability in providing cloud computing services.

The vSphere 4 is built with two functional software suites: *infrastructure services* and *application services*. It also has three component packages intended mainly for virtualization purposes: *vCompute* is supported by ESX, ESXi, and DRS virtualization libraries from VMware; *vStorage* is supported by VMS and

thin provisioning libraries; and *vNetwork* offers distributed switching and networking functions. These packages interact with the hardware servers, disks, and networks in the data center. These infrastructure functions also communicate with other external clouds.

The application services are also divided into three groups: *availability*, *security*, and *scalability*. Availability support includes VMotion, Storage VMotion, HA, Fault Tolerance, and Data Recovery from VMware. The security package supports vShield Zones and VMsafe. The scalability package was built with DRS and Hot Add. Interested readers should refer to the vSphere 4 web site for more details regarding these component software functions. To fully understand the use of vSphere 4, users must also learn how to use the vCenter interfaces in order to link with existing applications or to develop new applications.

3.5.4 Trust Management in Virtualized Data Centers

A VMM changes the computer architecture. It provides a layer of software between the operating systems and system hardware to create one or more VMs on a single physical platform. A VM entirely encapsulates the state of the guest operating system running inside it. Encapsulated machine state can be copied and shared over the network and removed like a normal file, which proposes a challenge to VM security. In general, a VMM can provide secure isolation and a VM accesses hardware resources through the control of the VMM, so the VMM is the base of the security of a virtual system. Normally, one VM is taken as a management VM to have some privileges such as creating, suspending, resuming, or deleting a VM.

Once a hacker successfully enters the VMM or management VM, the whole system is in danger. A subtler problem arises in protocols that rely on the "freshness" of their random number source for generating session keys. Considering a VM, rolling back to a point after a random number has been chosen, but before it has been used, resumes execution; the random number, which must be "fresh" for security purposes, is reused. With a stream cipher, two different plaintexts could be encrypted under the same key stream, which could, in turn, expose both plaintexts if the plaintexts have sufficient redundancy. Noncryptographic protocols that rely on freshness are also at risk. For example, the reuse of TCP initial sequence numbers can raise TCP hijacking attacks.

3.5.4.1 VM-Based Intrusion Detection

Intrusions are unauthorized access to a certain computer from local or network users and intrusion detection is used to recognize the unauthorized access. An intrusion detection system (IDS) is built on operating systems, and is based on the characteristics of intrusion actions. A typical IDS can be classified as a *host-based IDS (HIDS)* or a *network-based IDS (NIDS)*, depending on the data source. A HIDS can be implemented on the monitored system. When the monitored system is attacked by hackers, the HIDS also faces the risk of being attacked. A NIDS is based on the flow of network traffic which can't detect fake actions.

Virtualization-based intrusion detection can isolate guest VMs on the same hardware platform. Even some VMs can be invaded successfully; they never influence other VMs, which is similar to the way in which a NIDS operates. Furthermore, a VMM monitors and audits access requests for hardware and system software. This can avoid fake actions and possess the merit of a HIDS. There are two different methods for implementing a VM-based IDS: Either the IDS is an independent process in each VM or a high-privileged VM on the VMM; or the IDS is integrated into the VMM

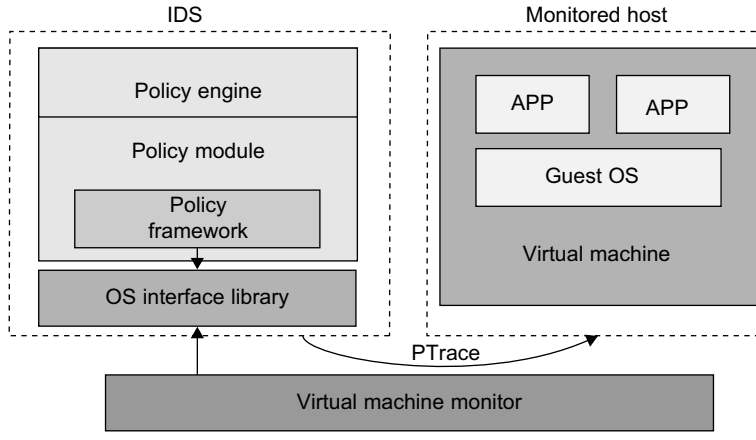


FIGURE 3.29

The architecture of livewire for intrusion detection using a dedicated VM.

(*Courtesy of Garfinkel and Rosenblum, 2002 [17]*)

and has the same privilege to access the hardware as well as the VMM. Garfinkel and Rosenblum [17] have proposed an IDS to run on a VMM as a high-privileged VM. Figure 3.29 illustrates the concept.

The VM-based IDS contains a policy engine and a policy module. The policy framework can monitor events in different guest VMs by operating system interface library and PTrace indicates trace to secure policy of monitored host. It's difficult to predict and prevent all intrusions without delay. Therefore, an analysis of the intrusion action is extremely important after an intrusion occurs. At the time of this writing, most computer systems use logs to analyze attack actions, but it is hard to ensure the credibility and integrity of a log. The IDS log service is based on the operating system kernel. Thus, when an operating system is invaded by attackers, the log service should be unaffected.

Besides IDS, honeypots and honeynets are also prevalent in intrusion detection. They attract and provide a fake system view to attackers in order to protect the real system. In addition, the attack action can be analyzed, and a secure IDS can be built. A honeypot is a purposely defective system that simulates an operating system to cheat and monitor the actions of an attacker. A honeypot can be divided into physical and virtual forms. A guest operating system and the applications running on it constitute a VM. The host operating system and VMM must be guaranteed to prevent attacks from the VM in a virtual honeypot.

Example 3.14 EMC Establishment of Trusted Zones for Protection of Virtual Clusters Provided to Multiple Tenants

EMC and VMware have joined forces in building security middleware for trust management in distributed systems and private clouds. The concept of *trusted zones* was established as part of the virtual infrastructure. Figure 3.30 illustrates the concept of creating trusted zones for virtual clusters (multiple

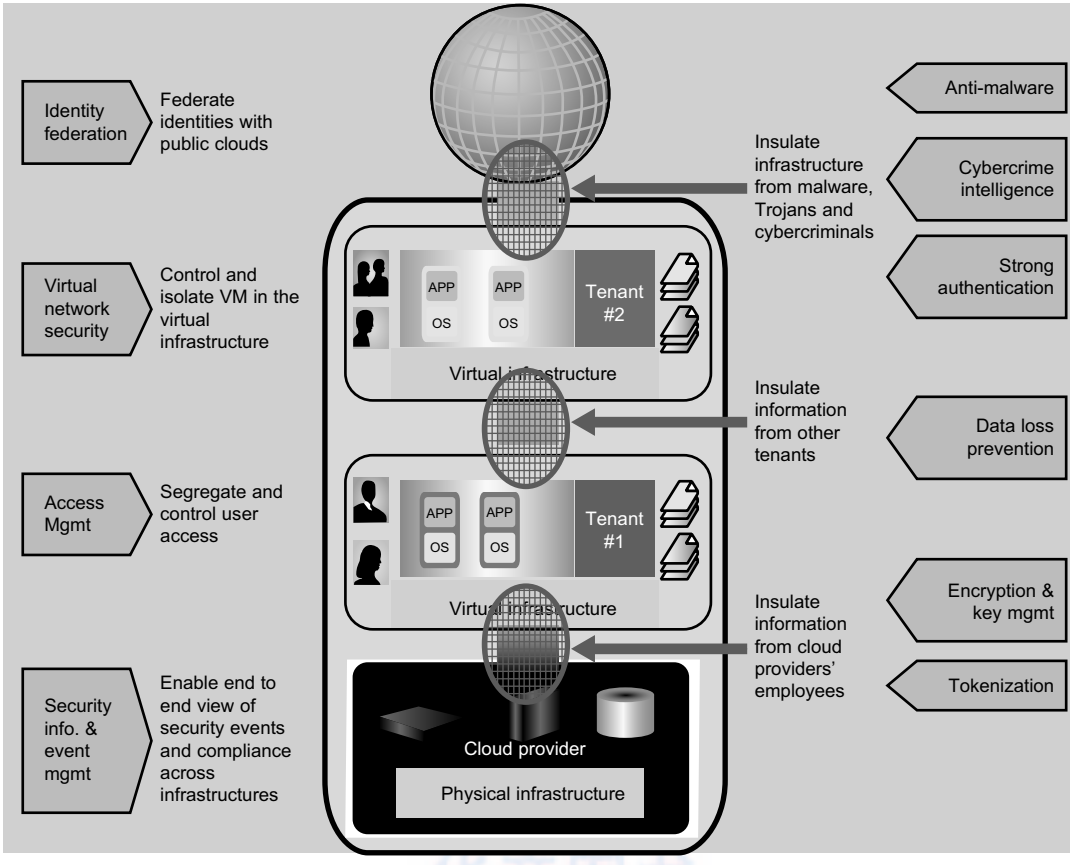


FIGURE 3.30

Techniques for establishing trusted zones for virtual cluster insulation and VM isolation.

(*Courtesy of L. Nick, EMC [40]*)

applications and OSes for each tenant) provisioned in separate virtual environments. The physical infrastructure is shown at the bottom, and marked as a cloud provider. The virtual clusters or infrastructures are shown in the upper boxes for two tenants. The public cloud is associated with the global user communities at the top.

The arrowed boxes on the left and the brief description between the arrows and the zoning boxes are security functions and actions taken at the four levels from the users to the providers. The small circles between the four boxes refer to interactions between users and providers and among the users themselves. The arrowed boxes on the right are those functions and actions applied between the tenant environments, the provider, and the global communities.

Almost all available countermeasures, such as anti-virus, worm containment, intrusion detection, encryption and decryption mechanisms, are applied here to insulate the trusted zones and isolate the VMs for private tenants. The main innovation here is to establish the trust zones among the virtual clusters.

The end result is to enable an end-to-end view of security events and compliance across the virtual clusters dedicated to different tenants. We will discuss security and trust issues in Chapter 7 when we study clouds in more detail.

3.6 BIBLIOGRAPHIC NOTES AND HOMEWORK PROBLEMS

Good reviews on virtualization technology can be found in Rosenblum, et al. [53,54] and in Smith and Nair [58,59]. White papers at [71,72] report on virtualization products at VMware, including the vSphere 4 cloud operating system. The Xen hypervisor was introduced in [7,13,42] and KVM in [31]. Qian, et al. [50] have given some theoretical treatment of the subject. ISA-level virtualization and binary translation techniques are treated in [3] and in Smith and Nair [58]. Some comparison of virtualizing software environments can be found in Buyya, et al. [9]. Hardware-level virtualization is treated in [1,7,8,13,37,43,54,67]. Intel's support of hardware-level virtualization is treated in [62]. Some entries in Table 3.5 are taken from Buyya, et al. [9].

For GPU computing on VMs, the readers are referred to [57]. The x86 host virtualization is treated in [2]. Pentium virtualization is treated in [53]. I/O virtualzation is treated in [11,24,26,52].

Sun Microsystems reports on OS-level virtualization in [64]. OpenVZ is introduced in its user's guide [65]. Virtualization in Windows NT machines is described in [77,78]. For GPU computing on VMs, readers are referred to [53]. The x86 host virtualization is treated in [2]. Pentium virtualization is treated in [53]. The book [24] have a good coverage of hardware support for virtualization. Virtual clusters are treated in [8,12,20,25,49,55,56]. In particular, Duke's COD is reported in [12] and Purdue's Violin in [25,55]. Memory virtualization is treated in [1,10,13,51,58,68].

Hardware-level virtualization is treated in [1,7,8,13,41,47,58,73]. Intel's support of hardware-level virtualization is treated in [68]. Wells, et al. [74] have studied multi-core virtualization. The integration of multi-core and virtualization on future CPU/GPU chips posts a very hot research area, called asymmetric CMP as reported in [33,39,63,66,74]. Architectural supports for virtualized chip multiprocessors have been also studied in [17,28,30,39,66]. The maturity of this co-design CMP approach will greatly impact the future development of HPC and HTC systems.

Server consolidation in virtualized data centers is treated in [29,39,75]. Power consumption in virtualized data centers is treated in [44,46]. Literatures [46,60,61,62] discussed the virtual resource management in data centers. Kochut gives an analytical model for virtualized data centers [32]. Security protection and trust management for virtualized data centers are treated in [18,45]. For virtual storage, readers are referred to the literature [27,36,43,48,76,79]. Specific references for the examples are identified in the figure captions or in the text description of the example. The Eucalyptus was reported in [45], vSphere in [72], Parallax in [43]. The vCUDA was reported in [57] for CUDA programming on the VMs.

Acknowledgments

This chapter is coauthored by Zhibin Yu of Huazhong University of Science and Technology (HUST), China and by Kai Hwang of USC. Dr. Hai Jin and Dr. Xiaofei Liao of HUST have extended significant technical support to this work. The authors would like to thank Dr. Zhong-Yuan Qin

of Southeast University, China, Fan Zhang of Tsinghua University, and Lizhong Chen and Zhou Zhao of USC for their assistance in drawing several sketched figures and updating the references in this chapter.

References

[1] Advanced Micro Devices. AMD Secure Virtual Machine Architecture Reference Manual, 2008.
[2] K. Adams, O. Agesen, A comparison of software and hardware techniques for x86 virtualization, in: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 2006, pp. 21–25.
[3] V. Adve, C. Lattner, et al., LLVA: A low-level virtual instruction set architecture, in: Proceedings of the 36th International Symposium on Micro-architecture (MICRO-36 '03), 2003.
[4] J. Alonso, L. Silva, A. Andrzejak, P. Silva, J. Torres, High-available grid services through the use of virtualized clustering, in: Proceedings of the 8th Grid Computing Conference, 2007.
[5] P. Anedda, M. Gaggero, et al., A general service-oriented approach for managing virtual machine allocation, in: Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC 2009), ACM Press, March 2009, pp. 9–12.
[6] H. Andre Lagar-Cavilla, J.A. Whitney, A. Scannell, et al., SnowFlock: rapid virtual machine cloning for cloud computing, in: Proceedings of EuroSystems, 2009.
[7] P. Barham, B. Dragovic, K. Fraser, et al., Xen and the art of virtualization, in: Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP19), ACM Press, 2003, pp. 164–177.
[8] E. Bugnion, S. Devine, M. Rosenblum, Disco: running commodity OS on scalable multiprocessors, in: Proceedings of SOSP, 1997.
[9] R. Buyya, J. Broberg, A. Goscinski (Eds.), Cloud Computing: Principles and Paradigms, Wiley Press, New York, 2011.
[10] V. Chadha, R. Illikkal, R. Iyer, I/O Processing in a virtualized platform: a simulation-driven approach, in: Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE), 2007.
[11] R. Chandra, N. Zeldovich, C. Sapuntzakis, M.S. Lam, The collective: a cache-based system management architecture, in: Proceedings of the Second Symposium on Networked Systems Design and Implementation (NSDI '05), USENIX, Boston, May 2005, pp. 259–272.
[12] J. Chase, L. Grit, D. Irwin, J. Moore, S. Sprenkle, Dynamic virtual cluster in a grid site manager, in: IEEE Int'l Symp. on High Performance Distributed Computing, (HPDC-12), 2003.
[13] D. Chisnall, The Definitive Guide to the Xen Hypervisor, Prentice Hall, International, 2007.
[14] C. Clark, K. Fraser, S. Hand, et al., Live migration of virtual machines, in: Proceedings of the Second Symposium on Networked Systems Design and Implementation (NSDI '05), 2005, pp. 273–286.
[15] Y. Dong, J. Dai, et al., Towards high-quality I/O virtualization, in: Proceedings of SYSTOR 2009, The Israeli Experimental Systems Conference, 2009.
[16] E. Elnozahy, M. Kistler, R. Rajamony, Energy-efficient server clusters, in: Proceedings of the 2nd Workshop on Power-Aware Computing Systems, February 2002.
[17] J. Frich, et al., On the potential of NoC virtualization for multicore chips, in: IEEE Int'l Conf. on Complex, Intelligent and Software-Intensive Systems, 2008, pp. 801–807.
[18] T. Garfinkel, M. Rosenblum, A virtual machine introspection-based architecture for intrusion detection, 2002.
[19] L. Grit, D. Irwin, A. Yumerefendi, J. Chase, Virtual machine hosting for networked clusters: building the foundations for autonomic orchestration, in: First International Workshop on Virtualization Technology in Distributed Computing (VTDC), November 2006.

[20] D. Gupta, S. Lee, M. Vrable, et al., Difference engine: Harnessing memory redundancy in virtual machines, in: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '08), 2008, pp. 309–322.
[21] M. Hines, K. Gopalan, Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning, in: Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments (VEE '09), 2009, pp. 51–60.
[22] T. Hirofuchi, H. Nakada, et al., A live storage migration mechanism over WAN and its performance evaluation, in: Proceedings of the 4th International Workshop on Virtualization Technologies in Distributed Computing, 15 June, ACM Press, Barcelona, Spain, 2009.
[23] K. Hwang, D. Li, Trusted cloud computing with secure resources and data coloring, IEEE Internet Comput., (September/October) (2010) 30–39.
[24] Intel Open Source Technology Center, System Virtualization—Principles and Implementation, Tsinghua University Press, Beijing, China, 2009.
[25] X. Jiang, D. Xu, VIOLIN: Virtual internetworking on overlay infrastructure, in: Proceedings of the International Symposium on Parallel and Distributed Processing and Applications, 2004, pp. 937–946.
[26] H. Jin, L. Deng, S. Wu, X. Shi, X. Pan, Live virtual machine migration with adaptive memory compression, in: Proceedings of the IEEE International Conference on Cluster Computing, 2009.
[27] K. Jin, E. Miller, The effectiveness of deduplication on virtual machine disk images, in: Proceedings of SYSTOR, 2009, The Israeli Experimental Systems Conference, 2009.
[28] S. Jones, A. Arpaci-Disseau, R. Arpaci-Disseau, Geiger: Monitoring the buffer cache in a virtual machine environment, in: ACM ASPLOS, San Jose, CA, October 2006, pp. 13–14.
[29] F. Kamoun, Virtualizing the datacenter without compromising server performance, ACM Ubiquity 2009, (9) (2009).
[30] D. Kim, H. Kim, J. Huh, Virtual snooping: Filtering snoops in virtualized multi-coures, in: 43rd Annual IEEE/ACM Int'l Symposium on Mcrosrchitecture (MICRO-43).
[31] A. Kivity, et al., KVM: The linux virtual machine monitor, in: Proceedings of the Linux Symposium, Ottawa, Canada, 2007, p. 225.
[32] A. Kochut, On impact of dynamic virtual machine reallocation on data center efficiency, in: Proceedings of the IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems (MASCOTS), 2008.
[33] R. Kumar, et al., Heterogeneous chip multiptiprocessors, IEEE Comput. Mag. 38 (November) (2005) 32–38.
[34] B. Kyrre, Managing large networks of virtual machines, in: Proceedings of the 20th Large Installation System Administration Conference, 2006, pp. 205–214.
[35] J. Lange, P. Dinda, Transparent network services via a virtual traffic layer for virtual machines, in: Proceedings of High Performance Distributed Computing, ACM Press, Monterey, CA, pp. 25–29, June 2007.
[36] A. Liguori, E. Hensbergen, Experiences with content addressable storage and virtual disks, in: Proceedings of the Workshop on I/O Virtualization (WIOV '08), 2008.
[37] H. Liu, H. Jin, X. Liao, L. Hu, C. Yu, Live migration of virtual machine based on full system trace and replay, in: Proceedings of the 18th International Symposium on High Performance Distributed Computing (HPDC '09), 2009, pp. 101–110.
[38] A. Mainwaring, D. Culler, Design challenges of virtual networks: Fast, general-purpose communication, in: Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, 1999.
[39] M. Marty, M. Hill, Virtual hierarchies to support server consolidation, in: Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA), 2007.

[40] M. McNett, D. Gupta, A. Vahdat, G.M. Voelker, Usher: An extensible framework for managing clusters of virtual machines, in: 21st Large Installation System Administration Conference (LISA) 2007.
[41] D. Menasce, Virtualization: Concepts, applications., performance modeling, in: Proceedings of the 31st International Computer Measurement Group Conference, 2005, pp. 407–414.
[42] A. Menon, J. Renato, Y. Turner, Diagnosing performance overheads in the Xen virtual machine environment, in: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, 2005.
[43] D. Meyer, et al., Parallax: Virtual disks for virtual machines, in: Proceedings of EuroSys, 2008.
[44] J. Nick, Journey to the private cloud: Security and compliance, in: Technical presentation by EMC Visiting Team, May 25, Tsinghua University, Beijing, 2010.
[45] D. Nurmi, et al., The eucalyptus open-source cloud computing system, in: Proceedings of the 9th IEEE ACM International Symposium on Cluster Computing and The Grid (CCGrid), Shanghai, China, September 2009, pp. 124–131.
[46] P. Padala, et al., Adaptive control of virtualized resources in utility computing environments, in: Proceedings of EuroSys 2007.
[47] L. Peterson, A. Bavier, M.E. Fiuczynski, S. Muir, Experiences Building PlanetLab, in: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI2006), 6–8 November 2006.
[48] B. Pfaff, T. Garfinkel, M. Rosenblum, Virtualization aware file systems: Getting beyond the limitations of virtual disks, in: Proceedings of USENIX Networked Systems Design and Implementation (NSDI 2006), May 2006, pp. 353–366.
[49] E. Pinheiro, R. Bianchini, E. Carrera, T. Heath, Dynamic cluster reconfiguration for power and performance, in: L. Benini (Ed.), Compilers and Operating Systems for Low Power, Kluwer Academic Publishers, 2003.
[50] H. Qian, E. Miller, et al., Agility in virtualized utility computing, in: Proceedings of the Third International Workshop on Virtualization Technology in Distributed Computing (VTDC 2007), 12 November 2007.
[51] H. Raj, I. Ganev, K. Schwan, Self-Virtualized I/O: High Performance, Scalable I/O Virtualization in Multi-core Systems, Technical Report GIT-CERCS-06-02, CERCS, Georgia Tech, 2006, www.cercs.gatech.edu/tech-reports/tr2006/git-cercs-06-02.pdf.
[52] J. Robin, C. Irvine, Analysis of the Intel pentium's ability to support a secure virtual machine monitor, in: Proceedings of the 9th USENIX Security Symposium Vol. 9, 2000.
[53] M. Rosenblum, The reincarnation of virtual machines, ACM QUEUE, (July/August) (2004).
[54] M. Rosenblum, T. Garfinkel, Virtual machine monitors: current technology and future trends, IEEE Comput 38 (5) (2005) 39–47.
[55] P. Ruth, et al., Automatic Live Migration of Virtual Computational Environments in a Multi-domain Infrastructure, Purdue University, 2006.
[56] C. Sapuntzakis, R. Chandra, B. Pfaff, et al., Optimizing the migration of virtual computers, in: Proceedings of the 5th Symposium on Operating Systems Design and Implementation, Boston, 9–11 December 2002.
[57] L. Shi, H. Chen, J. Sun, vCUDA: GPU accelerated high performance computing in virtual machines, in: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, 2009.
[58] J. Smith, R. Nair, Virtual Machines: Versatile Platforms for Systems and Processes, Morgan Kaufmann, 2005.
[59] J. Smith, R. Nair, The architecture of virtual machines, IEEE Comput., (May) (2005).
[60] Y. Song, H. Wang, et al., Multi-tiered on-demand resource scheduling for VM-based data center, in: Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, 2009.
[61] B. Sotomayor, K. Keahey, I. Foster, Combining batch execution and leasing using virtual machines, in: Proceedings of the 17th International Symposium on High-Performance Distributed Computing, 2008.

[62] M. Steinder, I. Whalley, et al., Server virtualization in autonomic management of heterogeneous workloads, ACM SIGOPS Oper. Syst. Rev. 42 (1) (2008) 94–95.
[63] M. Suleman, Y. Patt, E. Sprangle, A. Rohillah, Asymmetric chip multiprocessors: balancing hardware efficiency and programming efficiency, (2007).
[64] Sun Microsystems. Solaris Containers: Server Virtualization and Manageability, Technical white paper, September 2004.
[65] SWsoft, Inc. OpenVZ User's Guide, http://ftp.openvz.org/doc/OpenVZ-Users-Guide.pdf, 2005.
[66] F. Trivino, et al., Virtualizing netwoirk on chip resources in chip multiprocessors, J. Microprocess. Microsyst. 35 (2010). 245–230 http://www.elsevier.com/locate/micro.
[67] J. Xu, M. Zhao, et al., On the use of fuzzy modeling in virtualized datacenter management, in: Proceedings of the 4th International Conference on Autonomic Computing (ICAC07), 2007.
[68] R. Ublig, et al., Intel virtualization technology, IEEE Comput., (May) (2005).
[69] H. Van, F. Tran, Autonomic virtual resource management for service hosting platforms, CLOUD (2009).
[70] A. Verma, P. Ahuja, A. Neogi, pMapper: Power and migration cost aware application placement in virtualized systems, in: Proceedings of the 9th International Middleware Conference, 2008, pp. 243–264.
[71] VMware (white paper). Understanding Full Virtualization, Paravirtualization, and Hardware Assist, www.vmware.com/files/pdf/VMware_paravirtualization.pdf.
[72] VMware (white paper). The vSphere 4 Operating System for Virtualizing Datacenters, News release, February 2009, www.vmware.com/products/vsphere/, April 2010.
[73] J. Walters, et al., A comparison of virtualization technologies for HPC, in: Proceedings of Advanced Information Networking and Applications (AINA), 2008.
[74] P. Wells, K. Chakraborty, G.S. Sohi, Dynamic heterogeneity and the need for multicore virtualization, ACM SIGOPS Operat. Syst. Rev. 43 (2) (2009) 5–14.
[75] T. Wood, G. Levin, P. Shenoy, Memory buddies: Exploiting page sharing for smart collocation in virtualized data centers, in: Proceedings of the 5th International Conference on Virtual Execution Environments (VEE), 2009.
[76] J. Xun, K. Chen, W. Zheng, Amigo file system: CAS based storage management for a virtual cluster system, in: Proceedings of IEEE 9th International Conference on Computer and Information Technology (CIT), 2009.
[77] Y. Yu, OS-level Virtualization and Its Applications, Ph.D. dissertation, Computer Science Department, SUNY, Stony Brook, New York, December 2007.
[78] Y. Yu, F. Guo, et al., A feather-weight virtual machine for windows applications, in: Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE), Ottawa, Canada, 14–16 June 2006.
[79] M. Zhao, J. Zhang, et al., Distributed file system support for virtual machines in grid computing, in: Proceedings of High Performance Distributed Computing, 2004.

HOMEWORK PROBLEMS

Problem 3.1

Briefly answer the following questions on virtualization levels. Highlight the key points and identify the distinctions in different approaches. Discuss their relative advantages, shortcomings and limitations. Also identify example systems implemented at each level.

Problem 3.2

Explain the differences between hypervisor and para-virtualization and give one example VMM (virtual machine monitor), that was built in each of the two categories.

Problem 3.3

Install the VMware Workstation on a Windows XP or Vista personal computer or laptop, and then install Red Hat Linux and Windows XP in the VMware Workstation. Configure the network settings of Red Hat Linux and Windows XP to get on the Internet. Write an installation and configuration guide for the VMware Workstation, Red Hat Linux, and Windows XP systems. Include any trouble-shooting tips in the guide.

Problem 3.4

Download a new kernel package from www.kernel.org/. Compile it in Red Hat Linux in the VMware Workstation installed in Problem 3.3 with Red Hat Linux on a real computer. Compare the time required for the two compilations. Which one takes longer to compile? What are their major differences?

Problem 3.5

Install Xen on a Red Hat Linux machine in two methods from the binary code or from the source code. Compile installation guides for the two methods used. Describe the dependencies of utilities and packages along with troubleshooting tips.

Problem 3.6

Install Red Hat Linux on the Xen you installed in Problem 3.5. Download nbench from www.tux.org/~mayer/linux/bmark.html. Run the nbench on the VM using Xen and on a real machine. Compare the performance of the programs on the two platforms.

Problem 3.7

Use the utilities for easing deployment of Google enterprise applications in VMs. The Google-vm-deployment tool can be downloaded from http://code.google.com/p/google-vm-deployment/.

Problem 3.8

Describe the approaches used to exchange data among the domains of Xen and design experiments to compare the performance of data communication between the domains. This is designed to familiarize you with the Xen programming environment. It may require a longer period of time to port the Xen code, implement the application code, perform the experiments, collect the performance data, and interpret the results.

Problem 3.9

Build your own LAN by using the VMware Workstation. The topological structure of the LAN is specified in Figure 3.31. Machine A is required to install Red Hat Linux while machine B is required to install Windows XP.

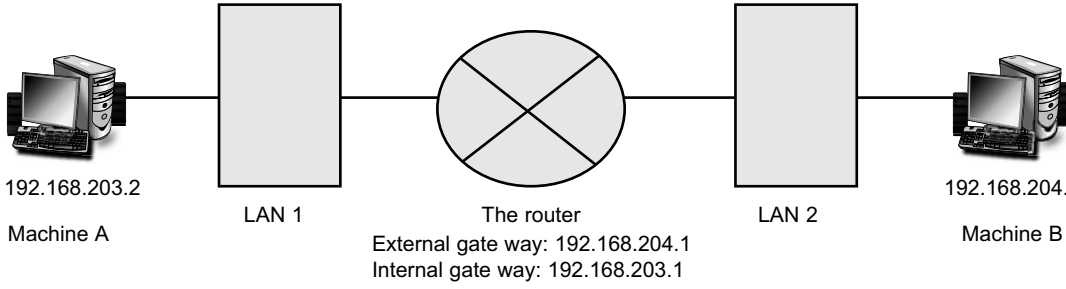


FIGURE 3.31

The topological structure of the virtual LAN.

Problem 3.10

Study the relevant papers [33,63,74] on asymmetric or heterogeneous chip multitprocessors (CMP). Write a study report to survey the area, identify the key research issues, review the current development and open research challenges lying ahead.

Problem 3.11

Study the relevant papers [17,28,30,66] on network on chip (NoC) and virtualization of NoC resources for multi-core CMP design and applications. Repeat Problem 3.10 with a survey report after the research study.

Problem 3.12

Hardware and software resource deployment are 4 often complicated and time-consuming. Automatic VM deployment can significantly reduce the time to instantiate new services or reallocate resources depending on user needs. Visit the following web site for more information. http://wiki.systemimager.org/index.php/Automating_Xen_VM_deployment_with_SystemImager. Report your experience with automatic deployment using the SystemImager and Xen-tools.

Problem 3.13

Design an experiment to analyze the performance of Xen live migration for I/O read-intensive applications. The performance merits include the time consumed by the precopy phase, the downtime, the time used by the pull phase, and the total migration time.

Problem 3.14

Design an experiment to test the performance of Xen live migration for I/O write-intensive applications. The performance metrics include the time consumed by the precopy phase, the downtime, the time used by the pull phase, and the total migration time. Compare the results with those from Problem 3.13.

Problem 3.15

Design and implement a VM execution environment for grid computing based on VMware Server. The environment should enable grid users and resource providers to use services that are unique to a VM-based approach to distributed computing. Users can define customized execution environments which can then be archived, copied, shared, and instantiated as multiple runtime clones.

Problem 3.16

Design a large-scale virtual cluster system. This problem may require three students to work together for a semester. Assume that users can create multiple VMs at one time. Users can also manipulate and configure multiple VMs at the same time. Common software such as OS or libraries are preinstalled as templates. These templates enable users to create a new execution environment rapidly. Finally, you can assume that users have their own profiles which store the identification of data blocks.

Problem 3.17

Figure 3.32 shows another VIOLIN adaptation scenario for changes in virtual environments. There are four VIOLIN applications running in two cluster domains. Trace the three steps of VIOLIN job execution and discuss the gains in resource utilization after live migration of the virtual execution

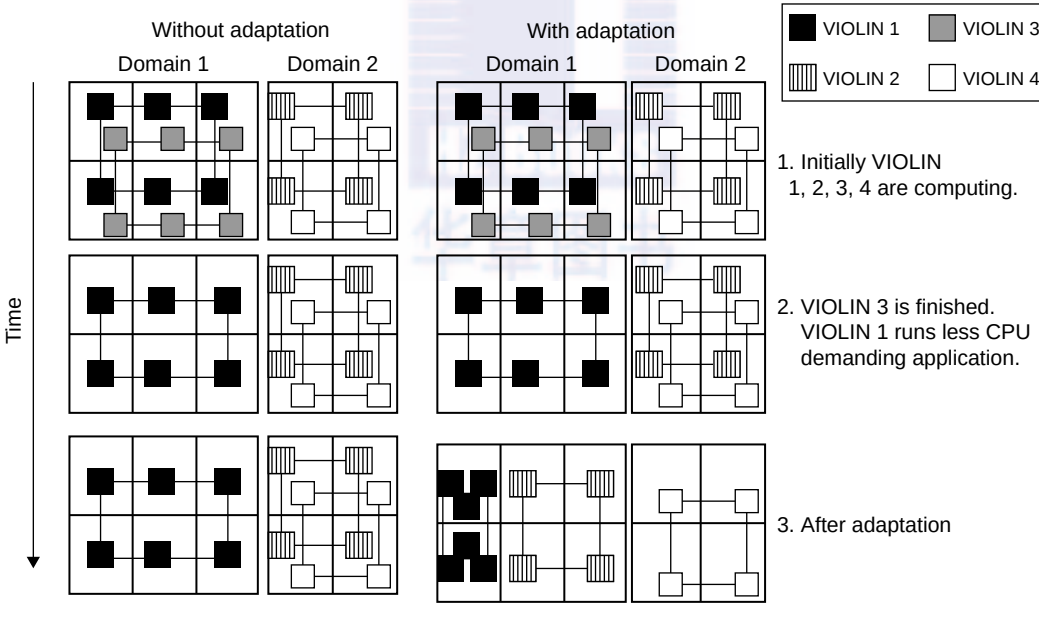


FIGURE 3.32

An adaptation scenario with four VIOLINs running in two cluster domains in the VIOLIN Virtual clustering experiments.

(*Courtesy of P. Ruth, et al. [55]*)

environment in the two cluster domains. You can check your results against the cited paper to compare your observations.

Problem 3.18

After studying the material presented in Section 3.3.5, plus reading the papers by Wells, et al. [74] and by Marty and Hill in [39] answer the following two questions:

a. Distinguish virtual cores from physical cores and discuss the mapping technique in Wells's paper on upgrading resource utilization and fault tolerance in using virtualized multicore processors.

b. Study the cache coherence protocol presented in the Marty and Hill paper and discuss its feasibility and advantages to implement on many-core CMP in the future.

HZ BOOKS
华章图书