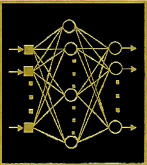
ELSEVIER
ACADEMIC
PRESS

Advances in COMPUTERS

Volume 63

Parallel, Distributed, and Pervasive Computing



Guest Editor

ALI R. HURSON

This page intentionally left blank

Advances

in COMPUTERS

VOLUME 63

This page intentionally left blank

Advances in COMPUTERS

Parallel, Distributed, and Pervasive Computing

GUEST EDITOR

ALI R. HURSON

Department of Computer Science and Engineering
The Pennsylvania State University
University Park, Pennsylvania

SERIES EDITOR

MARVIN V. ZELKOWITZ

Department of Computer Science
and Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland

VOLUME 63

ELSEVIER
ACADEMIC
PRESS

Amsterdam Boston Heidelberg London New York Oxford
Paris San Diego San Francisco Singapore Sydney Tokyo

ELSEVIER B.V.
Radarweg 29
P.O. Box 211, 1000 AE Amsterdam
The Netherlands

ELSEVIER Inc.
525 B Street, Suite 1900
San Diego, CA 92101-4495
USA

ELSEVIER Ltd
The Boulevard, Langford Lane
Kidlington, Oxford OX5 1GB, UK

ELSEVIER Ltd
84 Theobalds Road
London WC1X 8RR, UK

First edition 2005

ISBN: 0-12-012163-8
ISSN (Series): 0065-2458

⊗ The paper used in this publication meets the requirements of ANSI/NISO Z39.48-1992 (Permanence of Paper).

Printed in the USA.

Contents

Contributors

Shoukat Ali is an Assistant Professor in the Department of Electrical and Computer Engineering at the University of Missouri–Rolla. From 1998 to 2003, he was at the School of Electrical and Computer Engineering at Purdue where he earned his M.S.E.E. (1999) and Ph.D. (2003) degrees. He received a B.S. degree (1996) in electrical engineering from the University of Engineering and Technology (UET), Lahore, Pakistan. His research interests include heterogeneous parallel and distributed computing systems, grid computing, sensor networks, and robust resource allocation. He has co-authored 18 published technical papers in these and related areas.

Noah Beck received his MS in Electrical Engineering and BS in Computer Engineering from Purdue University in 1999 and 1997, respectively. He is currently a Verification Engineer for Advanced Micro Devices in Boxboro, MA working on future AMD64 microprocessor designs. His research interests include grid computing, processor microarchitecture, and processor verification techniques.

Ladislau Bölöni is an Assistant Professor at the Electrical and Computer Engineering department of University of Central Florida. He received a PhD degree from the Computer Sciences Department of Purdue University in May 2000. He published more that 50 research papers and 2 books. His research interests include autonomous agents, grid computing and ad-hoc networks.

Tracy D. Braun received his Ph.D. in Electrical and Computer Engineering from Purdue University in 2001. He is currently the Network Engineer for 3rd Dental Battalion/U.S. Naval Dental Center, Okinawa, Japan. He received an M.S.E.E. from Purdue in 1997, and a B.S. in Electrical and Computer Engineering with Honors and High Distinction from the University of Iowa in 1995. He is a member of IEEE, IEEE Computer Society, Eta Kappa Nu, and AFCEA. He is a CISSP and MCP.

Wissam Chedid received the B.S. degree in electrical engineering from Lebanese University, Lebanon, in 2001 and the M.S. degree in electrical engineering from Cleveland State University, in 2003. His current research interests include embedded systems, power-efficient computer architecture, performance evaluation, simulation, and multimedia systems. He is a member of the IEEE Computer Society.

Mary Jane Irwin has been on the faculty at Penn State University since 1977, and currently holds the title of the A. Robert Noll Chair in Engineering in the Computer Science and Engineering. Her research and teaching interests include computer architecture, embedded and mobile computing systems design, power aware design, and electronic design automation. Her research is supported by grants from the MARCO Gigascale Systems Research Center, the National Science Foundation, the Semiconductor Research Corporation, and the Pennsylvania Pittsburgh Digital Greenhouse. She received an Honorary Doctorate from Chalmers University, Sweden, in 1997 and the Penn State Engineering Society's Premier Research Award in 2001. She was named an IEEE Fellow in 1995, an ACM Fellow in 1996, and was elected to the National Academy of Engineering in 2003.

Dr. Irwin is currently serving as the co-Editor-in-Chief of ACM's Journal of Emerging Technologies in Computing Systems, as a member of ACM's Publication Board, and on the Steering Committee of CRA's Committee on the Status of Women in Computer Science and Engineering. Dr. Irwin received her M.S. (1975) and Ph.D. (1977) degrees in computer science from the University of Illinois, Urbana–Champaign.

Jean-Luc Gaudiot received the Diplôme d'Ingénieur from the École Supérieure d'Ingénieurs en Electronique et Electrotechnique, Paris, France in 1976 and the M.S. and Ph.D. degrees in Computer Science from the University of California, Los Angeles in 1977 and 1982, respectively. Professor Gaudiot is currently a Professor in the Electrical Engineering and Computer Science Department at the University of California, Irvine. Prior to joining UCI in January 2002, he was a Professor of Electrical Engineering at the University of Southern California since 1982, where he served as Director of the Computer Engineering Division for three years. He has also done microprocessor systems design at Teledyne Controls, Santa Monica, California (1979–1980) and research in innovative architectures at the TRW Technology Research Center, El Segundo, California (1980–1982). His research interests include multithreaded architectures, fault-tolerant multiprocessors, and implementation of reconfigurable architectures. He has published over 150 journal and conference papers. His research has been sponsored by NSF, DoE, and DARPA, as well as a number of industrial organizations. He served as the Editor-in-Chief of the IEEE

Transactions on Computers. Currently he is an IEEE Fellow and elected chair of the IEEE Technical Committee on Computer Architecture.

Jung-Yup Kang received the BS degree and MS degree in computer science in 1992 and in 1994 from Yonsei University, Korea. He worked as a researcher at Samsung Semiconductor, Korea developing its 64-bit microprocessor from 1994 to 1996. He also worked for Synopsys as a design methodology consultant from 1996 to 1997. He then received his MS and PhD degrees in Computer Engineering from the University of Southern California in 1999 and 2004. He has published several papers focusing on computer architecture, computer arithmetic, parallel processing, and VLSI designs. He is currently working for Mindspeed technologies as an IP verification engineer.

Ben Lee received the B.E. degree in electrical engineering from the State University of New York (SUNY) at Stony Brook, New York, in 1984 and the Ph.D. degree in computer engineering from The Pennsylvania State University, University Park, in 1991. He is currently an Associate Professor in the School of Electrical Engineering and Computer Science at Oregon State University. He has been on the program committees and served as publicity chair for several international conferences. His research interests include network computing, computer system architecture, multithreading and thread-level speculation, and parallel and distributed systems. Dr. Lee is a member of the IEEE Computer Society.

Greg Link received his B.E. degree in Electrical Engineering (2002) from the Pennsylvania State University, and his B.S. degree in Physics (2002) from Juniata College. He is currently working towards a PhD in Computer Science and Engineering at the Pennsylvania State University. His areas of interest include Network-on-Chip design, thermal-aware computing, and application-specific processors.

Mohan Kumar is an Associate Professor in Computer Science and Engineering at the University of Texas at Arlington. His current research interests are in pervasive computing, wireless networks and mobility, P2P systems, and distributed computing. He has published over 100 articles in refereed journals and conference proceedings and supervised Masters and doctoral theses in the above areas. Kumar is on the editorial boards of The Computer Journal and the Pervasive and Mobile Computing Journal. He is a co-founder of the IEEE International Conference on pervasive computing and communications (PerCom)—served as the program chair for PerCom 2003, and is the general chair for PerCom 2005. He is a senior member of the IEEE.

Anthony A. Maciejewski received the BSEE, MS, and PhD degrees from Ohio State University in 1982, 1984, and 1987. From 1988 to 2001, he was a Professor of Electrical and Computer Engineering at Purdue University, West Lafayette. He is currently the Department Head of Electrical and Computer Engineering at Colorado State University. An up-to-date biography is at http://www.engr.colostate.edu/~aam.

Muthucumaru Maheswaran is an Assistant Professor in the School of Computer Science at McGill University, Canada. In 1990, he received a BSc degree in electrical and electronic engineering from the University of Peradeniya, Sri Lanka. He received an MSEE degree in 1994 and a PhD degree in 1998, both from the School of Electrical and Computer Engineering at Purdue University. His research interests include design, implementation, analysis, and use of Public Computing Utilities, security and trust aware resource management systems, and tools for teaching and learning computer networks. He has coauthored over 60 technical papers in these and related areas.

Amar Mukherjee is a Professor of Computer Science with the School of Computer Science at the University of Central Florida. Professor Mukherjee's interests include switching theory, VLSI design, multi-media data compression, and bioinformatics. He is the author of a textbook Introduction to nMOS and CMOS VLSI Systems Design (Prentice Hall, 1986) and is a co-author and editor of Recent Developments in Switching Theory (Academic Press, 1971) which is a collection of fundamental works on switching theory by leading researchers. Professor Mukherjee is a Fellow of the IEEE and has been inducted as a member of the Golden Core of the IEEE. He has served the IEEE as the editor of the IEEE Transactions on Computers for three terms and also served as Chair of the VLSI Technical Committee and a member of the Steering Committee for the IEEE Transactions on VLSI Systems. He continues to serve as the Chair of the Steering Committee of the Annual IEEE Symposium on VLSI.

Vijaykrishnan Narayanan received his B.E. degree in Computer Science and Engineering from SVCE, University of Madras in 1993 and his Ph.D. degree in Computer Science and Engineering from University of South Florida, Tampa in 1998. Since 1998, he has been with the Computer Science and Engineering Department at the Pennsylvania State University where he is currently an associate professor. His research interests are in the areas of energy-aware reliable systems, embedded Java, nano/VLSI systems and computer architecture. He has authored more than 100 papers in these areas. His current research projects are supported by National Science Foundation, DARPA/MARCO Gigascale Silicon Research Center, Office of Naval Research, Semiconductor Research Consortium and Pittsburgh Digital Greenhouse.

Albert I. Reuther is a member of the Technical Staff in the Embedded Digital Systems group at the MIT Lincoln Laboratory in Lexington, Massachusetts. Dr. Reuther holds a BSCEE, MSEE, and Ph.D. in Electrical and Computer Engineering from Purdue University (1994, 1996, and 2000, respectively) and an MBA from the Collège des Ingénieurs in Paris, France (2001). He is a member of IEEE, IEEE Computer Society, and Eta Kappa Nu honor society and has worked at General Motors, Hewlett-Packard, and DaimlerChrysler. His research interests include distributed and parallel computing, digital signal processing, and numerical methods.

Won Woo Ro received the BS degree in Electrical Engineering from Yonsei University, Seoul, Korea, in 1996. He received the M.S. and PhD degrees in Electrical Engineering from the University of Southern California in 1999 and 2004, respectively. He also worked as a research staff in Electrical Engineering and Computer Science Department in University of California, Irvine from 2002 to 2004. Currently, Dr. Ro is an Assistant Professor in the Department of Electrical and Computer Engineering of the California State University, Northridge. His current research interest includes high-performance microprocessor design and compiler optimization.

James P. Robertson is currently a Performance Architect at nVidia Corporation in Austin, Texas. From 1998 through 2004 he was a PowerPC Performance and Modeling Engineer at Motorola's Semiconductor Products Sector, now Freescale Semiconductor, Inc. He received an MSEE in 1998 and a BSCMPE in 1996, both from the School of Electrical and Computer Engineering at Purdue University. He is a member of IEEE, IEEE Computer Society, and Eta Kappa Nu.

Ravi Vijaya Satya received the bachelor's degree in Electronics and Communications Engineering from Jawaharlal Nehru Technological University, Hyderabad, India in 1999. He received his Master's in Computer Science from University of Central Florida, Orlando, USA in 2001. He is currently pursuing Ph.D. in Computer Science from the University of Central Florida. His research interests include Bioinformatics, Data Compression, and Pattern Matching. He is a member of the IEEE.

Behrooz A. Shirazi is a Professor and Chair of Computer Science and Engineering Department at University of Texas at Arlington. His areas of research interest and expertise include pervasive computing, software tools, and parallel and distributed systems. Dr. Shirazi is currently the Special Issues Editor-in-Chief for the Pervasive and Mobile Computing Journal. Previously, he has served on the editorial boards of the IEEE Transactions on Computers and Journal of Parallel and Distributed Computing. He has been a co-founder of the IEEE SPDP and PerCom conferences. He has received numerous teaching and research awards and has served as an IEEE Distinguished Visitor (1993–96) as well as an ACM Lecturer (1993–97).

H.J. Siegel is the George T. Abell Endowed Chair Distinguished Professor of Electrical and Computer Engineering, and a Professor of Computer Science, at Colorado State University (CSU). He is Director of the university-wide CSU Information Science and Technology Center (ISTeC). From 1976 to 2001, he was a Professor at Purdue University. He received two BS degrees from MIT, and the MA, MSE, and PhD degrees from Princeton University. He has co-authored over 300 published papers on parallel and distributed computing and communication, is an IEEE Fellow, and is an ACM Fellow. For more information: www.engr.colostate.edu/~hj.

Weifeng Sun is working towards a PhD at the University of Central Florida. His research interests include real-time computer graphics and data compression. Previously he was a member of technical staff in Bell Labs (China), Lucent Technologies. He was a member of the team of the wireless communication department which received the Bell Labs President's gold award, 2000. Sun holds a BS degree in computer software from University of Petroleum, China, and an MS degree in computer software from Institute of Software, Chinese Academy of Sciences.

Byung Y. Sung is a PhD candidate in the computer science and engineering department at the University of Texas at Arlington. His research interests include pervasive and ubiquitous computing, parallel and distributed system, and mobile computing. He received a BS in computer science and engineering from the Inha University, Korea, and an MS in computer science and engineering from the University of Texas at Arlington. Contact him at sung@cse.uta.edu; P.O. Box 19015, Department of Computer Science and Engineering, Arlington, TX 76010.

Tao Tao received the bachelor's degree in Electrical Engineering and Master's degree in Department of Computer Science and Engineering from Huazhong University of Science and Technology, Wuhan, P.R. China in 1994 and 1998, respectively. He obtained his Master's degree in Department of Computer Science, University of Central Florida, USA. He is currently a Ph.D. student at School of Computer Science, University of Central Florida, USA. His research interests include Image Compression, Compressed Pattern Matching, and Computer Graphics.

Theocharis Theocharides received his BSc. Degree in Computer Engineering (2002) from the Pennsylvania State University. He is currently a Ph.D. Candidate in the Computer Science and Engineering Department, at the Pennsylvania State University. He is part of the Embedded and Mobile Computing Design Center, and the

Microsystems Design Laboratory. His research interests include Systems-On-Chip Design and Interconnects, Application Mapping on Systems on Chip, Low-Power and Reliable VLSI Design and Application-Specific Processors. He is an IEEE Member and a Fulbright Scholar.

Mitchell D. Theys is currently an Assistant Professor in the Computer Science Department at the University of Illinois at Chicago. He received a Ph.D. and Masters in Electrical Engineering from Purdue University in 1999 and 1996, respectively. His current research interests include: distributed, heterogeneous and parallel computing, incorporating technology in the lecture hall, and computer architecture. Dr. Theys has several journals and refereed conference papers published. Dr. Theys is a member of the IEEE, IEEE Computer Society, Eta Kappa Nu, and Tau Beta Pi.

Bin Yao received the B.S. degree from Beijing University, Beijing, P.R. China, and the Ph.D. in Electrical and Computer Engineering from Purdue University, West Lafayette. He is a research scientist at the Bell Laboratories, Lucent Technologies. His research interests include fault tolerant computing, network monitoring, and embedded systems. He is a member of the IEEE and a member of the ACM.

Chansu Yu is currently an Associate Professor in the Department of Electrical and Computer Engineering at the Cleveland State University. Before joining the CSU, he was on the research staff at LG Electronics, Inc. for eight years until 1997 and then was on the faculty at Information and Communications University, Korea for three years. Prof. Yu received the BS and MS degrees in electrical engineering from Seoul National University, Korea, in 1982 and 1984, respectively. He received the Ph.D. in computer engineering from the Pennsylvania State University in 1994. He has been on the program committees of several international conferences. His research interests include mobile computing, computer architecture, parallel and clustering computing, and performance modeling and evaluation. Dr. Yu is a member of the IEEE and IEEE Computer Society.

Nan Zhang received the bachelor's degree in Applied Mathematics and Information Science from Beijing Institute of Economics, Beijing, P.R. China in 1990. He obtained his Master's degree in Department of Computer Science from National University of Singapore, Singapore in 1998. He is currently a Ph.D. student at School of Computer Science, University of Central Florida, USA. His research interests include Data Compression, Compressed Pattern Matching, and Information Retrieval. He is an IEEE student member.

Preface

The term computation gap has been defined as the difference between the computational power demanded by the application domain and the computational power of the underlying computer platform. Traditionally, closing the computation gap has been one of the major and fundamental tasks of computer architects. However, as technology advances and computers become more pervasive in society, the domain of computer architecture has been extended. The scope of research in computer architecture is no longer restricted to computer hardware and organizational issues. A wide spectrum of topics ranging from algorithm design to power management is becoming part of the computer architecture. Based on the aforementioned trend and to reflect recent research efforts, attempts were made to select a collection of articles that covers different aspects of contemporary computer architecture design. This volume of the Advances in Computers contains six chapters on different aspects of computer architecture.

In the early 1980s, the RISC (Reduced Instruction Set Computer) philosophy dominated the general design trend of computers with a theoretical peak performance of one clock cycle per instruction. The quest for higher performance and breaking the theoretical RISC performance barrier, combined with the experiences gained from the design of vector processors and multifunctional systems have motivated three architectural methodologies: super-scalar processor, Very Long Instruction Word machines, and super pipelined architecture. Chapter 1 is intended to address these architectural designs and their impact on modern microprocessor design.

Continued advances in technology have resulted in faster clock rate, reduced feature size, and lowered supply voltage. This trend has enabled the chip designers to incorporate more functional modules on the chip. The problem of establishing an interconnection among these modules is a natural by product of the increased chip density. However, several issues as such reliability, power consumption, and application mapping must be resolved before one can take advantage of these technological advances. Chapter 2 of this volume entitled “Networks on Chip (NoC): Interconnects of Next Generation Systems on Chip” addresses the aforementioned issues within the scope of the NoC architecture. In addition, it presents a case study of mapping the neural network architecture onto an NoC architecture.

Since the introduction of multiprocessing, research has shown special interest in using load balancing and task scheduling techniques as the means to improve performance. Chapter 3 entitled “Characterizing Resource Allocation Heuristics for Heterogeneous Computing Systems” extends the scope of this research to the domain of distributed systems where heterogeneity and autonomy are the two major architectural characteristics. This chapter also introduces a three-part classification scheme for heterogeneous computing platforms.

The design of high performance, compact, light weight, and reliable computer is often hindered by the heat dissemination problem. This observation has lead to the development of many power management strategies. Chapter 4 entitled “Power Analysis and Optimization Techniques for Energy Efficient Computer Systems” discusses several techniques that can reduce the energy consumption of a computer system at different levels.

Chapter 5 entitled “Flexible and Adaptive Services in Pervasive Computing” extends the domain of distributed processing to a pervasive computing environment. In this environment a vast number of smart devices such as sensors, embedded processors, personal computers, cell phones, and PDAs with limited resources and power source are communicating with each other through a network characterized by limited bandwidth and intermittent connectivity. In this platform, in response to an application, we need to develop means that effectively and efficiently locate the available services and compose new services. An adaptive and flexible service provisioning framework that satisfies such a need is the major theme of this chapter.

Finally, Chapter 6 investigates an application domain characterized by massive volumes of textual data. In this environment, by default, to conserve resources, information is compressed. Efficient search and retrieval of large volume of compressed information is an interesting issue and the major focus of this chapter. How to adopt an efficient storage and searching using the parallel and distributed computing is a challenging issue in the Internet age that will also be discussed in this chapter.

I hope you be able to find the contents of this volume useful in your research, teaching, and/or professional activities. In conclusion I would like to thank Marvin Zelkowitz, Andy Deelen, and Cathy Jiao for their support, encouragement, and useful suggestions during the course of this project.

Ali R. Hurson
Penn State University
University Park, PA, USA

Techniques to Improve Performance Beyond Pipelining: Superpipelining, Superscalar, and VLIW

JEAN-LUC GAUDIOT

The Henry Samueli School of Engineering
Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA 92697
USA
gaudiot@uci.edu

JUNG-YUP KANG

Mindspeed Technologies, Inc.
4311 Jamboree Road
Newport Beach, CA 92660
USA
jung-yup.kang@mindspeed.com

WON WOO RO

Department of Electrical and Computer Engineering
College of Engineering and Computer Science
California State University, Northridge
Northridge, CA 91330
USA
wro@csun.edu

ADVANCES IN COMPUTERS, VOL. 63
ISSN: 0065-2458/DOI 10.1016/S0065-2458(04)63001-4

1. Preface

Superpipelining, Superscalar and VLIW are techniques developed to improve the performance beyond what mere pipelining can offer.

Superpipelining improves the performance by decomposing the long latency stages (such as memory access stages) of a pipeline into several shorter stages, thereby possibly increasing the number of instructions running in parallel at each cycle.

On the other hand, Superscalar and VLIW approaches improve the performance by issuing multiple instructions per cycle. Superscalar dynamically issues multiple instructions and VLIW statically issues multiple instructions at each cycle. These techniques are pioneers (and now the basis) of modern computer architecture designs.

In this chapter, we describe and investigate examples of these techniques and examine how they have affected the designs of modern microprocessors.

2. Introduction—Overview

In general, the performance of a microprocessor is expressed by equation (1):

$$Execution\_Time = IC \times CPI \times clock\_cycle\_time \quad (1)$$

In this equation, *IC* corresponds to the number of instructions in the program at hand, while *CPI* represents the average number of clock cycles per instruction. *clock_cycle_time* represents the duration of a clock cycle.

In a basic pipeline architecture, only one instruction can be issued at each cycle (detailed information about pipelined architecture is given in [24–26]). Overcoming this limitation has been the subject of much work in the past: more specifically, it has been the goal of computer architects to reduce each variable of equation (1). For one thing, *CPI* can be improved by designing more advanced organizations and architectures and also by modifying the instruction set architecture (such as providing instructions which may require fewer cycles). *clock_cycle_time* can be improved by better process technology, advanced hardware implementation techniques, and also by more sophisticated architectural organizations. Finally, *IC* can be improved by implementing advanced compiler and instruction set architectures.

The three main techniques mentioned earlier (superpipelining, superscalar, and VLIW) are such improvements to the architecture. Indeed, superpipelining improves the performance by dividing the long latency stages (such as the memory access stages) of a pipeline into several shorter stages, thereby reducing the clock rate of the pipeline (in other words, superpipelining has the effect of reducing *clock_cycle_time* in equation (1)). Superscalar, on the other hand, improves the performance by dynamically issuing multiple instructions per cycle (this means that superscalar reduces *CPI*). VLIW (Very Long Instruction Word) improves the performance by utilizing the compiler to combine several instructions into one very long instruction and executing it (VLIW reduces *IC*). Table I highlights the differences among the three techniques.

For the last two decades, each of these techniques and a variety of combinations have had a significant effect on the performance of modern processors. Therefore, in this chapter, we first introduce the concepts and benefits of each technique and then examine examples of each kind.

TABLE I
COMPARISON OF SUPERPIPELINING, SUPERSCALAR, AND VLIW

	Superpipelining	**Superscalar**	**VLIW**
APPROACH & Effects	Dividing the long latency stages of a pipeline into several shorter stages. Effects on (*clock cycle time*) of equation (1).	Dynamically issuing multiple instructions per cycle. Effects on (*CPI*) of equation (1).	Utilizing the compiler to combine several instructions into one very long instruction and executing it. Effects on (*IC*) of equation (1).
Instruction issue rate	1	N	M
Instruction issue style	Dynamic	Dynamic	Static
Difficulty of Design	Relatively easy compared to the other two.	Complex design issues Run-time tracking.	Complex design issues Compiler support.
Keywords	Higher MHz, latch delay, clock skew.	Dynamic scheduling, Out-of-Order execution, Register renaming, Score boarding.	Compiler support, static scheduling, loop unrolling, trace scheduling.

3. Superpipelining

Superpipelining simply allows a processor to improve its performance by running the pipeline at a higher clock rate. The higher clock rate is achieved by identifying the pipeline stages that determine the clock cycle (in other words, the most time consuming stages) and decomposing them into smaller stages.

Superpipelining is the simplest technique of the three. It does not require additional hardware (such as functional units and fetch units) that a superscalar architecture does. It also does not require the complex controls (no multiple issue control or need to keep track of the instructions issued). Finally, it should be noted that superpipelining does not need the advanced compiler technologies required by the VLIW model.

3.1 Concepts and Benefits

The concept of superpipelining [5,15,16] is best explained by an illustration. Imagine the pipeline shown in Figure 1 (the simple pipeline of the DLX architecture [25, 24]). In this figure, each stage is drawn with a white box (with the corresponding name inside) while the gray skinny sticks represent the latches between the stages. The length of the double arrows represent the length of time required by each stage for completion of its task. From Figure 1, either the IF (Instruction Fetch) stage or the MEM (Data memory access) stage determine the clock rate because they are the longest time consuming stages of the pipeline (assume for the moment that the IF and MEM stages are nearly twice as long as the other stages).

We can increase the performance of the pipeline in Figure 1 by dividing the IF and the MEM (Data memory access) stages into two halves (and placing latches in between the new stages as in Figure 2). This allows the *clock_cycle_time* of equation (1) to be reduced (by half). This is the key point about superpipelining. (Organizing a pipeline with stages that have approximately the equal amount of execution time.) This reduced clock rate improves the performance. This is because the clock cycle for the pipeline is determined by the longest stage and as in Figure 3, if the pipeline

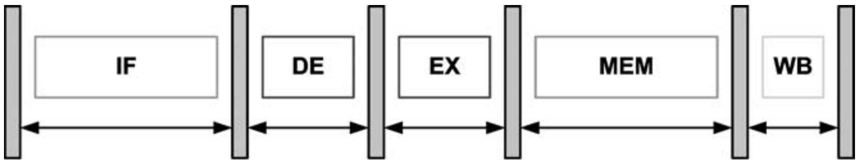


FIG. 1. Pipe latency dominated by memory accesses.

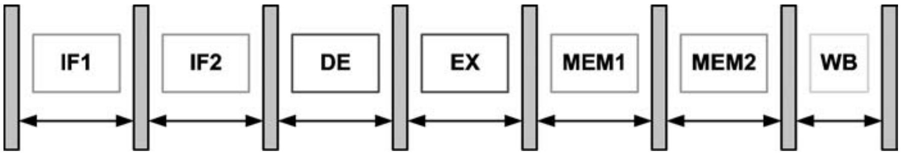


FIG. 2. Pipe latency not dominated by memory accesses.

has few long stages and the rest are nominal compared to the longer ones, there are wasted time when executing the shorter stages. Thus, if there are hundreds and thousands of instructions in a program, the aggregated wasted time will be significant (this wasted time is denoted with dotted lines in Figure 3(a)). On the other hand, if the pipeline was superpipelined with discrete design principles as it was in Figure 2, most of the stages will be balanced and the wasted time will be minimal as in Figure 3(b).

3.2 Design Challenges

At least superficially, superpipelining could appear easy to attain: simply divide the most time consuming stages into small units (and continue this process until all the stages are infinitely small). However, as we know, there is a limit to the efficiency (and potential of such a subdivision). Indeed, it can be observed from Figure 4 that a pipeline stage can be meaningfully pipelined only if the stage between two latches executes in more time (*A* in Figure 4) than one latch (*B* in Figure 4). Hwang [13] reports that in general, the time delay of a latch is about one or two order of magnitude shorter compared to the delay of the longest pipeline stage. Otherwise, we just spend more time on additional latches for longer pipelines. Second, after some point, it may simply be infeasible to continue dividing a specific pipeline stage. Third, as there are more pipeline stages and thus more latches to control, the clock skew problem [19] becomes significant which would in turn directly affect the time allotted to each stage to perform its execution.

Kunkel [19] shows the analysis of different latch designs and the effects of latch overhead in pipeline as well as data dependencies and clock skew overhead. He reports that if pipeline stages get extremely short, it becomes necessary to pad some delay to obtain performance improvement. He also says that eight to ten gate levels per pipeline stage lead to optimal overall performance.

Superpipelining also brings a number of major side effects. As the number of stages is increased, so is the number of forwarding paths (stages) and the delay (stall cycles) caused by branches and loads. The combination of the above two factors has strong adverse affects on the performance of the processor since it increases the *CPI*

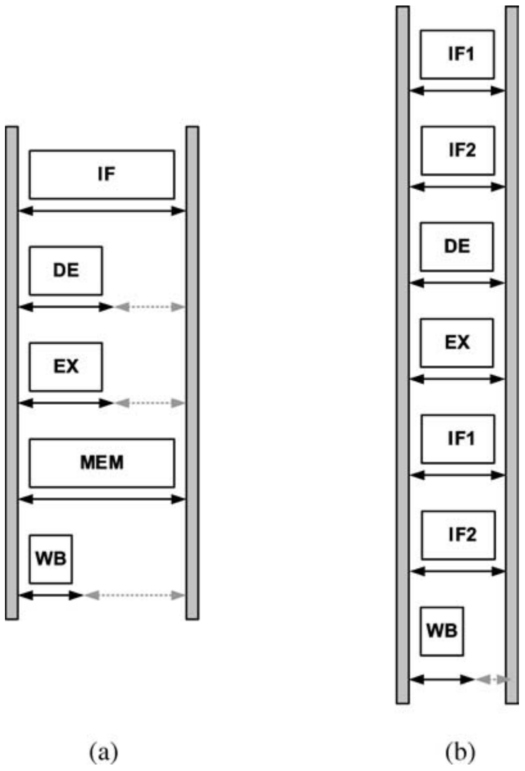


FIG. 3. Determination of pipeline clock cycle. (a) Pipeline with imbalanced stages; (b) Pipeline with balanced stages.

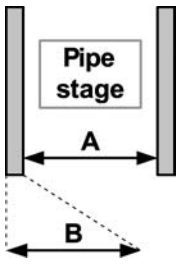


FIG. 4. Pipe latency restriction by the latency of a latch.

(therefore, techniques to improve *CPI*, such as hardware-based pipeline scheduling, branch prediction, multiple issue, and compiler-assisted parallelism exploiting techniques have been developed). These topics will be discussed in the context of superscalar and VLIW. However, it should be noted that when the above techniques are considered, superpipelining can improve *CPI* as well.

3.3 Example Architectures

Three processor architectures can show possible implementations of superpipelining. In those examples, we focus on how superpipelining has helped improve the performance of each architecture and show the design techniques to exploit the benefits of using superpipelining.

We first show an early version of a superpipelined architecture (MIPS R4000) and then move on to more modern and advanced models. The modern processors are not only superpipelined but also based on a combination of superpipelining and superscalar architecture.

3.3.1 MIPS R4000

The MIPS R4000 [4,10] is a superpipelined version of the MIPS R3000 which implemented only a 5-stage pipeline whereas the MIPS R4000 had 8 stages. The MIPS R4000 operates at double the speed (100 MHz) of the MIPS R3000 [6]. This was achieved by advances in the process technology and improved circuit design techniques [4].

One of the main goals of the architects of the MIPS R4000 was to improve the performance over that of MIPS R3000. The choice was between superpipelining and superscalar (and actually VLIW as well) [4]. Superpipelining was ultimately selected because:

(1) less logic is required compared to superscalar,
(2) the control logic is much simpler (no multiple issues and no need to keep track of the instructions issued and of the flow),
(3) (1) and (2) result in a faster cycle time, short design and test time, and
(4) no need for new compiler development.

The MIPS R4000 has a branch delay of three cycles and a load delay of two cycles. The MIPS R3000 had IF (Instruction Fetch), RF (Register Read), ALU (Add, logical, shift computation), DC (Data cache Access and tag check), and WB (Register File write) stages. The eight stages of R4000 are listed below:

- IF—Instruction Fetch, First Half,
- IS—Instruction Fetch, Second Half,
- RF—Register Fetch,
- EX—Execution Instruction Fetch,

- DF—Data Fetch, First Half,
- DS—Data Fetch, Second Half,
- TC—Tag Check,
- WB—Write Back.

3.3.2 The ARM11 Processor

ARM cores are famous for their simple and cost-effective design. However, ARM cores have also evolved and show superpipelining characteristics in their architectures and have architectural features to hide the possible long pipeline stalls. Amongst the ARM cores, the ARM11 processor is based on the ARM architecture v6. The ARM11 (specifically, the ARM1136JF-S processor [3]) is a high-performance and low-power processor which is equipped with eight stage pipelining. The core consists of two fetch stages, one decode stage, one issue stage, and four stages for the integer pipeline. The eight stages of the ARM11 core are depicted in Figure 5.

Below is described the function of each of the eight stages:

- Fe1—Instruction Fetch, First Half, Branch Prediction,
- Fe2—Instruction Fetch, Second Half, Branch Prediction,
- De—Instruction Decode,
- Iss—Register read and instruction issue,
- Sh—Shifter stage,
- ALU—Main integer operation calculation,
- Sat—Pipeline stage to enable saturation of integer results,

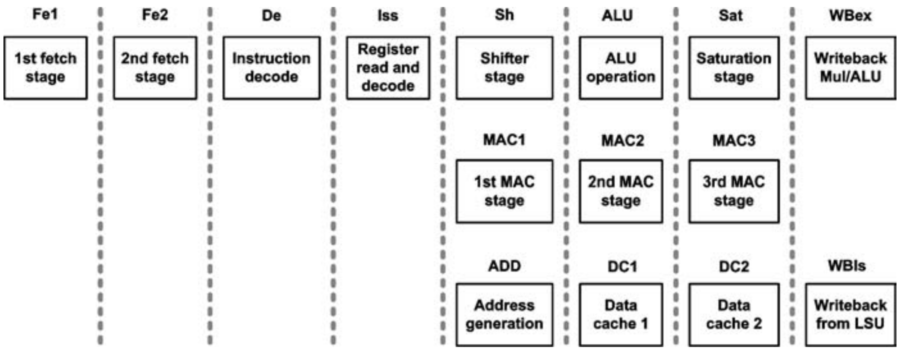


FIG. 5. Eight pipe stages of ARM11 core.

- WBex—Write Back of data from the multiply or main execution pipelines,
- MAC1—First stage of the multiply-accumulate pipeline,
- MAC2—Second stage of the multiply-accumulate pipeline,
- MAC3—Third stage of the multiply-accumulate pipeline,
- ADD—Address generation stage,
- DC1—First stage of Data Cache access,
- DC2—Second stage of Data Cache access,
- WBIs—Write back of data from the Load Store Unit.

The ARM11 core uses both dynamic (using branch target address cache and branch history) and static (using the direction of the branches) predictions to hide the deep penalty caused by having increased the length of the pipeline. The instruction prefetch unit of the ARM11 core buffers up to three instructions in its FIFO in order to [3]:

(1) detect branch instructions before they enter the integer units,
(2) dynamically predict those branches (to be taken), and
(3) provide branch folding of the predicted branches if possible.

By using this prefetch unit, the ARM11 core reduces the cycle time of the branch instructions and improves the performance. Previous ARM processors without the prefetch unit suffered from a one cycle delay when a branch was not taken and three or more cycles for the taken branches (since the target address was known only at the end of the execute stage).

3.3.3 The Intel NetBurst Microarchitecture

In this section, we discuss the Intel NetBurst microarchitecture (again, we focus on the superpipelining feature of the design). It is a deeply pipelined design (called hyper-pipelined technology) and it can, not only run at high clock rates (up to 10 GHz [14]), but it also allows different part of the processor run at different clock rates. For instance, the most frequently-executed instruction in common cases (such as cache hit) are decoded efficiently and executed with short latencies [14] therefore, improving the overall performance. The pipeline is also equipped with branch penalty hiding techniques such as speculation, buffering, superscalar, and out-of-order execution. The Pentium 4 processor and the Xeon processor are based on the Intel NetBurst microarchitecture.

However, in some modern complex processor designs (such as the NetBurst architecture), the number of cycles through which an instruction must go cannot be specifically determined because of the concept of “out-of-order execution.” However,

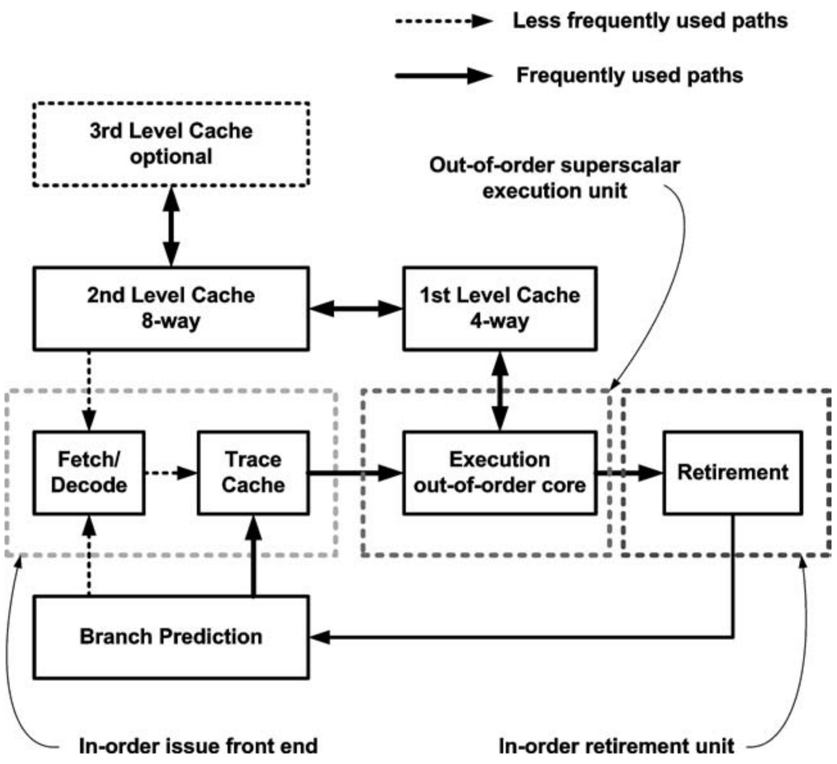


FIG. 6. Intel NetBurst pipeline architecture.

we can categorize the different parts of the pipeline through which an instruction must go. Thus, let us examine how these different parts of the NetBurst pipeline are correlated.

The NetBurst pipeline consists of three main parts and the pipeline architecture of NetBurst is depicted in Figure 6. Those three main parts are:

- the in-order issue front end,
- the out-of-order superscalar execution core,
- the in-order retirement unit.

In the NetBurst pipeline, the in-order issue front end feeds instructions to the out-of-order execution core. It fetches and decodes instructions. Instructions are decoded and translated into micro-ops (*μops*). The main responsibility of the front-end is to feed these micro-ops to the execution core in program order. It also utilizes the trace

TABLE II
COMPARISON OF MIPS R4000, ARM11 CORE, AND INTEL NETBURST

	MIPS R4000	**ARM11 Core**	**Intel NetBurst**
Number of Super-pipeline stages	8	8	3 distinguished groups of pipe stages (Front end, Execution unit, and Retirement unit. Each with different number of pipestages).
Pipeline Special Features	Advanced process technology and improved circuit design techniques (compared to 5-stage MIPS R3000).	Advanced branch predictions, prefetching, and buffering.	Hyper-pipelined technology (Pipe stages have different clock frequencies), speculation, buffering, superscalar, and out-of-order execution.

cache to store the program execution flow and it possesses a branch prediction unit. The execution core executes the micro-ops in an out-of-order fashion. The retirement unit retires the micro-ops in program order. In the next section, superscalar and out-of-order execution, which are another important architecture design feature of modern microprocessors (such as NetBurst architecture) will be discussed in detail.

The comparison of the features of the pipeline of the example superpipelining architectures are shown in Table II.

4. Superscalar Architectures

One of the main design issues for today's high-performance microprocessors is exploiting a high level of Instruction Level Parallelism (ILP). The ILP is a primary way to improve the parallelism of a sequential code. The basic two strategies for exploiting Instruction Level Parallelism are *multiple instruction issue* and *out-of-order execution*. Those two techniques were invented to execute as many instructions as possible in a given clock cycle. In addition, sophisticated branch prediction schemes and speculative execution techniques were proposed to overcome the limited amount of parallelism available within a basic block. Both techniques have successfully provided many opportunities to discover more parallelism across the control flow limit [29].

Traditionally, superscalar and Very Long Instruction Word (VLIW) architectures were developed as the main microprocessor models for exploiting ILP. Although both architectures target multiple instruction issue and out-of-order execution, they have adopted fundamentally different approaches. The significant differences lie in the way to schedule the instructions. Superscalar processors are mainly characterized with *dynamic scheduling* which uses hardware to schedule the instructions at run-time. On the contrary, VLIW architectures depend on *static scheduling* which schedules the instructions at compile time in a static way [11]. We will discuss the design features and some example architectures of superscalar and VLIW in this and the following section. First, the basic concept and design issues of superscalar architectures will be presented in detail.

4.1 Concept and Benefits

Superscalar architectures are strongly based on previous pipelined RISC processors. The most significant advantage of RISC processors is the inherent simplicity of the instruction set architecture. The early pipelined RISC processors were designed to execute one instruction for one pipeline stage. The major innovation of the superscalar architecture over the traditional pipelined processor is the *multiple instruction issue* advantage through *dynamic instruction scheduling*. In those approaches, each pipeline stage can handle multiple instructions simultaneously. Although the early superscalar architectures were designed to issue multiple instructions in an in-order fashion, this would consequently require out-of-order issue to maximize the effect of the multiple instruction issue feature. Together with the dynamic scheduling and enough backward compatibility, the multiple issue characteristic becomes the main advantage of the current superscalar processors.

The basic working mechanism of dynamic scheduling is quite simple. The instructions are simply fetched in the instruction sequence. They are then stored in a pool of instructions (also referred to as the *scheduling window*, *instruction window* or *instruction buffer*). Just as in the basic RISC architecture, the instructions are fetched sequentially following the program counter (PC). However, once the instructions are decoded, the dependencies among the instructions are analyzed. Then, the processor can deduce which instructions inside the pool can be executed without causing data hazards. Concurrent execution or out-of-order execution is made possible by hardware-based run-time dependency checking.

4.1.1 A Basic Model of Superscalar Pipeline

Figure 7 shows a typical model of a basic five-stage pipelined processor. In the single instruction issue RISC processor model, each stage handles only one instruction per cycle. In addition, the execution of the instructions should be made in-order, following the fetching sequence of the original binary code. On the contrary, each of the processing stages is able to handle a number of instructions in the superscalar pipeline model. For example, the IF (Instruction Fetch) stage can fetch multiple

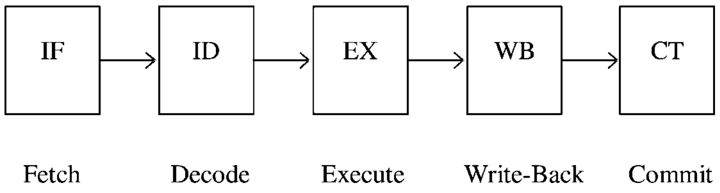
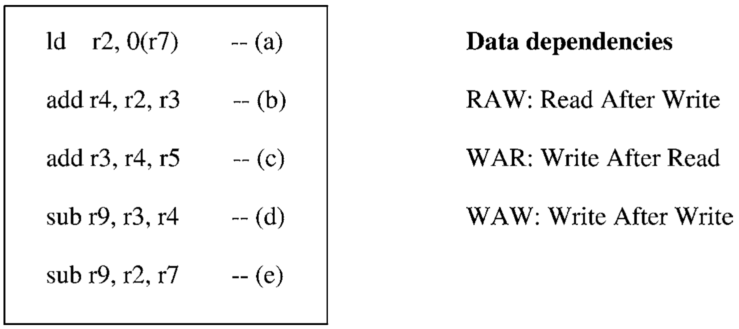


FIG. 7. Five-stage pipelined processor.

instructions from the instruction cache and the EX (Execution) stage can execute multiple instructions simultaneously. This multiple instruction issue characteristic inherently means out-of-order execution of the instruction sequence. It has been developed to reduce the execution time of a fixed size of code by exploiting fine-grain parallelism as low as at the instruction level.

However, executing multiple instructions simultaneously means the sequential characteristic of the original code cannot be guaranteed. This brings out highly critical problems since the current programming languages are strongly based on a sequential programming model, and the binary code is correctly executed only if its sequential characteristic is respected. An inappropriate execution order between instructions could cause data hazards due to the register naming dependencies. Figure 8 depicts possible data hazards due to an out-of-order execution.

The instruction stream on the left side of Figure 8 shows an example of a sequential instruction stream. In the previous pipelined RISC processor model, those instructions are fetched, executed, and committed in program order. This means that none of the instructions can be executed before any instruction ahead of it is committed. However, in the superscalar model, the processor aims at executing in a cycle



```
ld   r2, 0(r7)      -- (a)
add r4, r2, r3      -- (b)
add r3, r4, r5      -- (c)
sub r9, r3, r4      -- (d)
sub r9, r2, r7      -- (e)
```

Data dependencies

RAW: Read After Write

WAR: Write After Read

WAW: Write After Write

FIG. 8. Data dependencies in a sequential code.

as many instructions as possible through multiple instruction issue and out-of-order execution (within the allowed bandwidth).

Now, can the five instructions in Figure 8 be executed simultaneously? Obviously, executing those five instructions in parallel without considering the program order does not produce correct results. Indeed, there are some data dependencies.

For example, instructions (b) and (e) cannot be executed until instruction (a) completes, since both (b) and (e) need to read r2; there exists a RAW (Read After Write) data dependency. On the other hand, a WAR (Write After Read) dependency exists between the instruction (b) and (c) due to r3. Also, a WAW (Write After Write) dependency can be found between instructions (d) and (e). In the superscalar architecture model, the data dependencies which can cause data hazards are detected during the instruction decode stage. Based on those data dependencies, the instruction execution order is decided dynamically. It will be explained shortly in the next part. The dynamic scheduling of the Superscalar model is strongly based on the early Tomasulo [31] or Scoreboard algorithms [30].

4.1.2 Dynamic Scheduling of Superscalar

Superscalar architectures achieve dynamic instruction scheduling by using a scheduling window. In dynamic scheduling, the instructions are still fetched sequentially in program order. However, those instructions are decoded and stored in a scheduling window of a processor execution core. After decoding the instructions, the processor core obtains the dependency information between the instructions and can also identify the instructions which are ready for execution (ready instructions mean the instructions of which the source registers have been calculated). Indeed, there must be no data hazard in issuing the ready instructions.

In essence, the execution of an instruction *i* is decided by the previous instruction *k* which produces the value for the source registers of the instruction *i*. This means that the superscalar processors can execute the instructions as soon as the input values are available (also, the corresponding functional units should be available). The processor dynamically achieves parallel execution of the instructions by performing a run-time dependency check and waking up the following instructions.

However, at the same time, it also implies that every instruction should broadcast the register name and value across the issue window to wake up and select any ready instruction. This means that a superscalar architecture needs a complex wake-up and selection logic inside of the scheduling window. It should be also noted that a superscalar architecture is expected to deliver high degrees of Instruction-Level Parallelism, together with a large issue width which consequently requires more complex hardwired instruction scheduling logic. This will cause a significant overhead for future superscalar processors.

Although superscalar architectures require complex issue logic and scheduling mechanisms, they have been the standard general-purpose processor during the last decade. This is mostly due to the fact that this model possesses an inherent backward compatibility. This adaptability to legacy code and the ability to continue using conventional compilers make superscalar architectures quite attractive. That is the main reason why all major processor venders have focused on developing superscalar-styled ILP processors since the 1990s.

4.1.3 Register Renaming

Register renaming is a technique which increases the chance to uncover independent instructions by removing the false dependencies. Those false dependencies are caused by using the same register name. More specifically, the register renaming can eliminate the WAW (write after write) and WAR (write after read) dependencies among the instructions. Figure 9 shows one such example. The four instructions on the left side of Figure 9 show the original instruction stream produced by the compiler. Due to register r2, there exists a WAW dependency between instructions (a) and (c) and a WAR dependency between instructions (b) and (c). Those dependencies limit the parallel execution (or out-of-order execution) of the instructions. Clearly, the instruction (c) cannot be executed before instruction (b) reads r2. However, this data dependency can be removed when the register r2 in instruction (c) and (d) is given another register name. The four instructions on the right side show the instruction stream with register renaming.

Now, instruction (c) can be issued and executed, regardless of the execution of instruction (a), since it writes into register r2′. The register renaming scheme was originally developed as an compiling technique for an ILP processor such as VLIW.

```
                                        < after renaming >

add r2, r3, r4     -- (a)              add r2, r3, r4      -- (a)

add r4, r2, r3     -- (b)              add r4, r2, r3      -- (b)

sub r2, r6, r7     -- (c)              sub r2', r6, r7     -- (c)

mul r9, r2, r8     -- (d)              mul r9, r2', r8     -- (d)
```

FIG. 9. Register renaming example.

However, the superscalar architecture also uses a register renaming scheme for the same purpose. Its implementation is a purely hardware-based dynamic method. There are two ways we can implement the register renaming scheme in the superscalar model. The first method explicitly provides a larger number of physical registers than the logical registers. Then, at run-time, the processor can map any available physical register to the logical register. This technique requires a register mapping table. The second method uses a reorder buffer (ROB). In this approach, each entry of reorder buffer works as a physical register.

4.1.4 Reorder Buffer

A reorder buffer (ROB) provides two main functions in modern superscalar processor. The first task is to guarantee the in-order completion of a sequential instruction stream. Especially, the in-order completion is crucial for correct speculative execution and the implementation of precise interrupts [34]. As seen in Figure 7, a basic superscalar processor has two stages for instruction completion: the Write Back (WB) stage and the Commit (CT) stage. The WB stage allows the later instructions to use the result of the instruction. However, it does not allow the instruction to write the value into the register file or memory. The instruction still has to wait inside the ROB until it retires from the CT stage. The retire process in the CT stage strictly follows the program order (*in-order completion*). This in-order completion guarantees that the program can return to the precise state in any point of the program execution. The precise state means the sequentiality of the program execution is guaranteed. This is extremely important for interrupt or miss-speculation cases.

The ROB is implemented as a circular FIFO and retires the instructions following the program order; each entry of the ROB is assigned at the decoding stage and must hold the original program order. During the commit stage, the instructions at the head of the ROB are scanned and an instruction is committed if all the previous instructions are committed (multiple instructions within the commit-width can also be committed during the same cycle).

The second function of the ROB is to achieve register renaming through the slots of the ROB. When the processor uses an ROB, an available entry in the ROB is assigned to an instruction at the decoding stage. Each entry of the ROB contains three fields which indicate the instruction type, the destination register, and the result. The destination register field indicates the logical name of the register and the result field holds the value for the register. It also has a single bit ready indicator. When the ready bit indicates the value is ready, the later instructions which need the values can simply read the value; otherwise the later instruction should be left to wait inside a scheduling window until the value is ready.

4.1.5 Branch Prediction

A superscalar architecture is capable of delivering instructions beyond the current basic block by predicting the future control path. Therefore, excellent branch prediction is essential if we are to have a large pool of instructions to choose from. More particularly, the current trend in superscalar design is to achieve large issue-width with a large scheduling window. To satisfy those demands, the number of in-flight instructions (in other words, the instruction inside the processor in a clock cycle) should be large enough to find the ready instructions to issue or execute. By using efficient branch prediction and speculation techniques, fetching many instructions from the correct control path is possible.

The simplest way to predict branch is static branch prediction; it means the branch instructions are either always taken or always not-taken. However, this static method does not accurately predict the branch outcome. Therefore, current superscalar processors mainly depend on dynamic branch prediction which utilizes the previous history to decide the behavior of future branches. In current superscalar designs, many dynamic branch prediction techniques have been proposed and developed.

A basic form of branch prediction consists in using the local history of a branch instruction. The Branch History Table (BHT) is accessed by indexing a certain number of lower bits of the PC of the branch instruction, where the table provides the history of the branch behavior. In addition, other prediction schemes also consider the global branch history together with the local history. Those combined schemes significantly enhance the branch prediction success rate. Such schemes include two-level adaptive branch predictions [36] and branch correlation [23]. The interested reader is referred to these publications for details on the working mechanism and design features of those branch schemes.

Besides predicting the branch direction (either *taken* or *not-taken*), there had been one more prediction on the target address. It is possibly achieved by using a Branch Target Buffer to provide the target address prediction. This operation can determine the expected target address at the instruction fetch stage; therefore, the BTB is accessed during the IF stage and the next PC can be obtained for the very next cycle [34].

4.2 Design Challenges

In a conventional superscalar design, wake up and select operation cannot be pipelined and should be implemented as a one-cycle operation [22]. Therefore, as far as the size of an issue window is concerned, more instructions within an issue window mean more communication overhead (wire delay) for the wake up and select operations in a clock cycle time. In fact, wires tend to scale poorly compared to semiconductor devices, and the amount of state that can be reached in a single clock period eventually ceases to grow [1]. Consequently, there is a scaling problem with regard to the issue window size of the superscalar design. The goals of a larger instruction window and of a faster clock become directly antagonistic in the issue window design of future superscalar processors [1]. This phenomenon will be even more noticeable as higher degrees of parallelism are needed. It is well recognized that the centralized scheduling logic would be the most critical part of a processor and would limit the ultimate clock rate [22].

Another important issue in processor design is how to solve the problem caused by the dramatically growing speed-gap between the processor and main memory. An obvious solution would be the use of a large scheduling window: searching across a wider range of instructions can offer more opportunities to uncover independent instructions which can hide long access latencies. However, since a large instruction window will cause a lower clock rate, other approaches for processor design must be investigated. As an example, the partitioning of a single scheduling window can reduce the complexity of the centralized design as well as the size of each component [7,18]. Indeed, a higher clock rate can be achieved since the decentralized model reduces the length of the wire for communication. This microarchitecture technique is called *clustering*.

The last design challenge of the dynamic scheduling is the considerable amount of power consumption. According to Gowan et al. [9], 18% of total power consumption of the Alpha 21264 processor is used by the instruction issue unit. A large amount of power is dissipated by the wake up and the selection logic. This is mainly due to the global communication of the centralized scheduling unit. As an alternative design, the clustered architectures can reduce the power consumption of the centralized instruction issue unit. Separating a single scheduling window into multiple structures can reduce the power consumption of the aggressive scheduling unit of a superscalar architecture.

4.3 Example Architectures

We consider two microprocessor models as being good representatives of superscalar architectures: the Alpha 21264 and the MIPS R10000.

4.3.1 Alpha 21264

The Alpha 21264 (EV6) processor (Figure 10) was introduced in 1998 with a 0.35 μm CMOS process technology operating at 600 MHz frequency. It is the first Alpha processor designed with an out-of-order issue strategy.

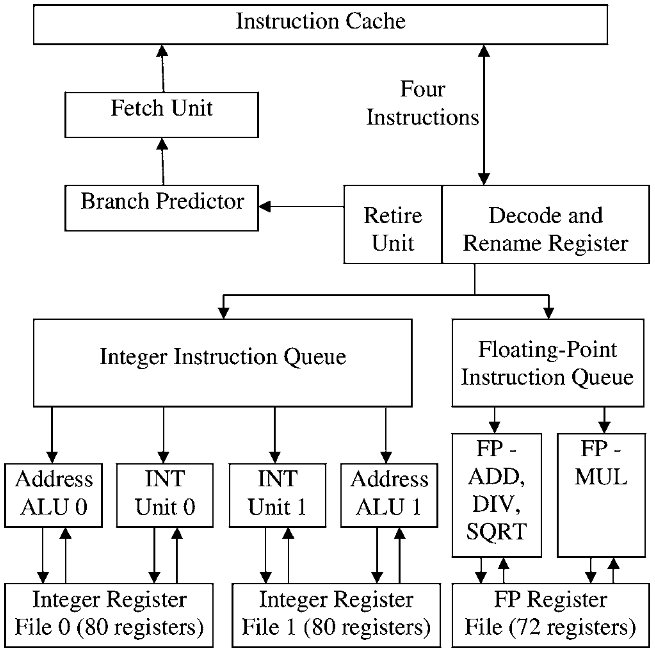

Instruction Cache
Fetch Unit
Four Instructions
Branch Predictor
Retire Unit
Decode and Rename Register
Integer Instruction Queue
Floating-Point Instruction Queue
Address ALU 0
INT Unit 0
INT Unit 1
Address ALU 1
FP - ADD, DIV, SQRT
FP - MUL
Integer Register File 0 (80 registers)
Integer Register File 1 (80 registers)
FP Register File (72 registers)

FIG. 10. Alpha 21264 block diagram.

The Alpha 21264 has a 4 way issue bandwidth. However, the execution stage supports 6-way issue in a cycle with four integer functional units and two floating-point functional units. It is also designed to support 80 in-flight instructions in a processor. Alpha 21264 has only two floating-point units and a single physical register file. Both floating-point units are fully pipelined.

The Alpha 21264 processor is implemented with a 64-KB, two-way set-associative L1 data cache. The earlier model of the processor (Alpha 21164) had only a 8-KB data cache which resulted in many more cache misses than the more modern 21264 processor. It now has a huge L1 data cache. As a result, the access latency to the data cache requires two clock cycles. However, two concurrent accesses to the level 1 data cache are supported.

Figure 11 shows a detailed description of the Alpha 21264 pipeline. In its basic inception, an instruction is required to go through all seven stages until it can write back the results. Since the 64 KB data cache needs two clock cycles to deliver the data, load/store pipeline operations require two more clock cycles than normal operations.

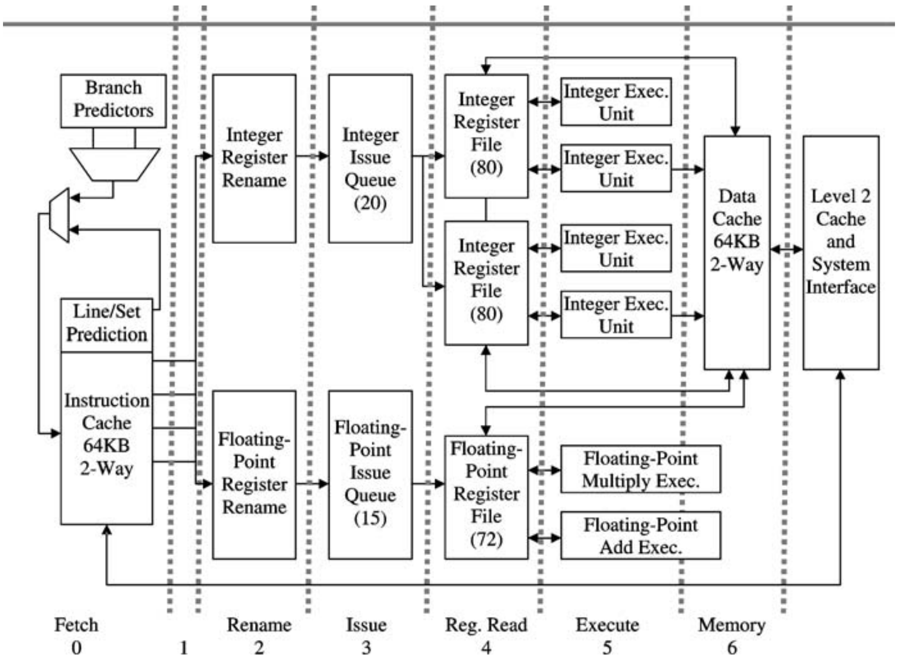


FIG. 11. Alpha 21264 pipeline.

One distinct feature of the Alpha 21264 is in that the fetch stage uses line and way prediction for the next instruction to fetch. It is possibly achieved by using the next line predictor and set predictor. The purpose of those predictors are similar to that of the Branch Target Buffer: it provides the address for the next instruction to fetch. As for the control flow prediction, the processor still has a branch prediction mechanism: a hybrid branch predictor is implemented. One more interesting feature of the processor is the load hit/miss prediction which helps to schedule load instructions.

The EV7 architecture (Alpha 21364), the fourth generation of the Alpha processor, inherited many characteristics from the Alpha 21264 processor. Indeed, the Alpha 21364 is designed to be the System-On-Chip version of the Alpha 21264. It uses the EV6 as the core processing element and adds peripheral features such as integrated second level cache, memory controller, and network interface.

4.3.2 MIPS R10000

The MIPS R10000 [35] is based on the previous MIPS processors. The previous MIPS processors were mainly single issue pipelined RISC processors. The MIPS

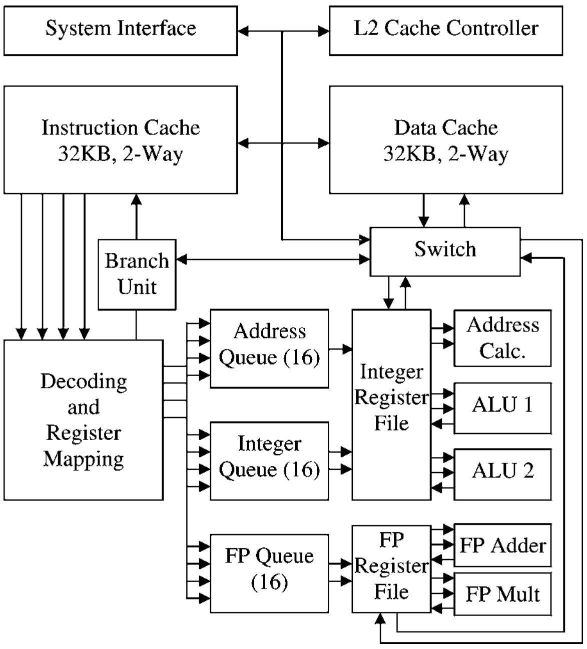


FIG. 12. MIPS R10000 block diagram.

R10000 is the first out-of-order issue superscalar processor which is implemented for the MIPS IV ISA. The MIPS R10000 processor fetches four instructions from the instruction cache in every cycle and decodes those instructions. Any branch is immediately predicted upon detection.

The first implementation of the MIPS IV ISA architecture is the 4-issue superscalar processor (R8000). The R8000 is able to execute two ALU and two memory operations per cycle. However, the R8000 processor does not have a dynamic scheduling capability. The most distinguishable advantage of the R10000 (Figure 12) is the out-of-order execution feature which is enabled by the dynamic scheduling.

Four instructions can be fetched and decoded at each clock cycle. Those instructions are stored in one of the three instruction queues after the decoding stage. Each queue has 16 entries for instructions. Those issue queues can also read operands from the register file. Once the register dependencies are resolved and functional units are available, the instructions can be executed in one of the five functional units. Normal integer operations take one clock cycle while load store instructions requires two clock cycles in the execution stage. The floating-point operations need three clock cycles for their execution.

TABLE III
COMPARISON OF THE TWO COMMERCIAL SUPERSCALAR PROCESSORS

Processor	Alpha 21264	MIPS R10000
Issue Width	4 way	4 way
Integer Unit	4	2
Address Unit	0	1
Floating Point Units	2	2
L1 I-Cache	64 KB	32 KB
L1 D-Cache	64 KB	32 KB
Integer Instruction Queue	20 entries	16 entries
Floating Point Instruction Queue	15 entries	16 entries
Integer Register	80	64
Floating Point Register	72	64
System Bus Width	64 bits	64 bits
Technology	0.35 µm	0.35 µm

Both level 1 caches are 32 KB in size and two-way interleaved. The core processor chip also controls a large external level 2 cache. The R10000 is implemented with a 64-bit split-transaction system bus used to communicate with the outside of the chip. Up to four R10000 chips can be integrated using this bus architecture.

As a summary of the two example architectures, Table III shows a comparison between the Alpha 21264 processor and the MIPS R10000 processor.

5. VLIW (Very Long Instruction Word)

VLIW has been developed to exploit Instruction-Level Parallelism by using a long instruction word which contains multiple fixed number of operations. Those operations can be fetched, decoded, issued, and executed at the same time without causing any data or control hazards. Therefore, all operations within a single VLIW instruction must be absolutely independent.

5.1 Concept and Benefits

In a VLIW architecture, parallel execution of multiple instructions is made possible by issuing a long instruction word. A single long instruction word is designed to achieve simultaneous execution of a fixed number of multiple operations. Those operations must be independent of each other to avoid possible data hazards. Indeed, several independent instructions are integrated inside a very long instruction word.

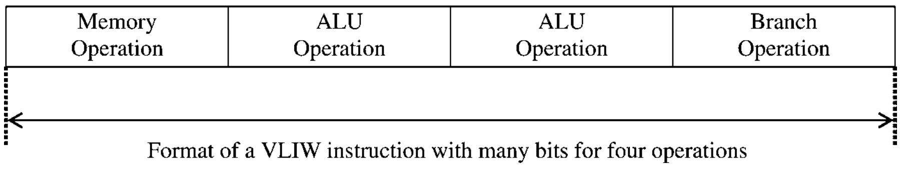

Memory Operation
ALU Operation
ALU Operation
Branch Operation
Format of a VLIW instruction with many bits for four operations

FIG. 13. An example of VLIW instructions.

The VLIW instruction is wide enough to allow the concurrent operation of multiple functional units. Its size normally ranges from 64 to 128 bits, and even up to 1024 bits. Figure 13 shows a typical format of VLIW instructions. Many bits on the long instruction enable a single instruction word to sufficiently control the several functional units directly and independently in every cycle.

Since it is the long instruction word which delivers the potential ILP to the hardware, a VLIW processor can be designed with a simpler hardware compared to an equivalent superscalar processor: it need not include the special units for the run-time dependency check and instruction scheduling. The block diagram of a simple VLIW processor is shown in Figure 14.

A VLIW architecture is, by essence, meant to activate multiple functional units at the same time. Therefore, the VLIW compiler should uncover independent operations to be executed in parallel. This means that the compiler must perform a detailed

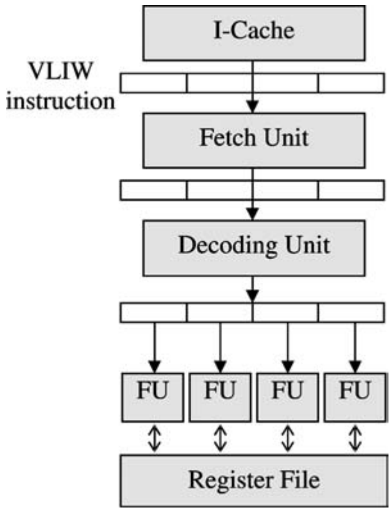


FIG. 14. VLIW hardware.

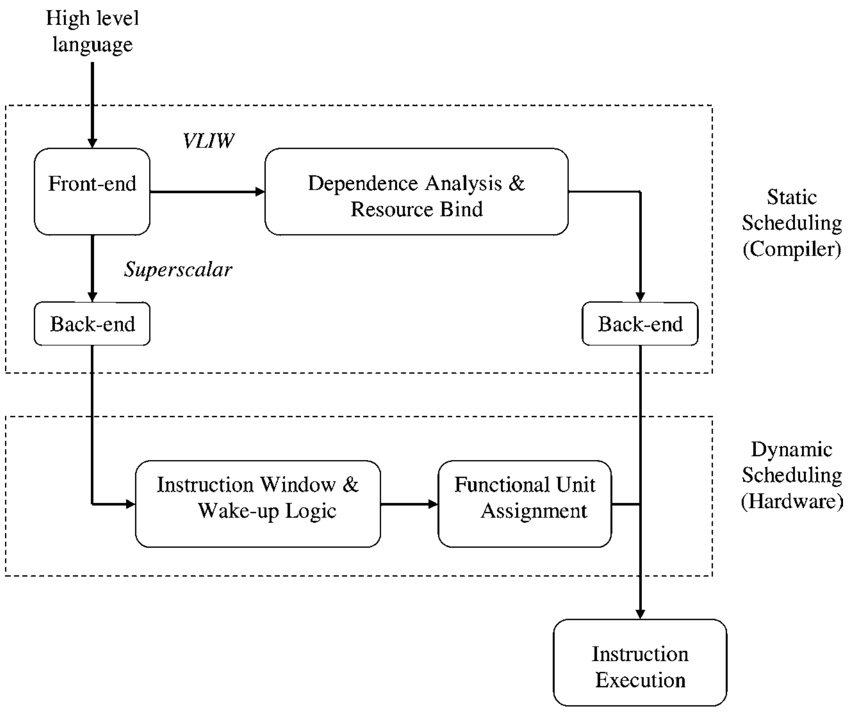

High level language
VLIW
Superscalar
Front-end
Dependence Analysis & Resource Bind
Back-end
Back-end
Static Scheduling (Compiler)
Instruction Window & Wake-up Logic
Functional Unit Assignment
Dynamic Scheduling (Hardware)
Instruction Execution

FIG. 15. Instruction scheduling in the superscalar and VLIW architectures.

analysis on the data-flow and control-flow at compile time (which is when the potential ILP is fixed). When the compiler cannot find enough instructions to fill in the width of the long instruction, it simply inserts a NOP.

When the interaction between the compiler and the hardware for instruction scheduling is considered, the fundamental differences between the VLIW and the superscalar design approaches can be explained as shown in Figure 15: in superscalar architectures, most of the scheduling is handled by hardware at run-time. However, in a VLIW architecture, the scheduling is decided by the compiler in a static way. Indeed, an intelligent compiler is the key component which ultimately decides the performance of a VLIW architecture.

Since the ILP within a basic block is quite limited, the VLIW architecture needs to examine more instructions to find more ILP. It is possibly achieved by looking at the instruction stream beyond the control-flow limits. For that purpose, several techniques such as *loop unrolling* and *trace scheduling* have been introduced in the VLIW design techniques. Those techniques will be explained shortly.

Compared to the dynamic scheduling of superscalar architectures, static scheduling does not require a complex scheduling logic. In addition, VLIW can uncover more parallelism by searching over a wider range of static code. Also, it is quite beneficial to know the source code structure to find parallelism in the VLIW architecture. However, several limitations such as long compilation time, not enough compatibility, and code explosion make VLIW architectures difficult to use in practice [20].

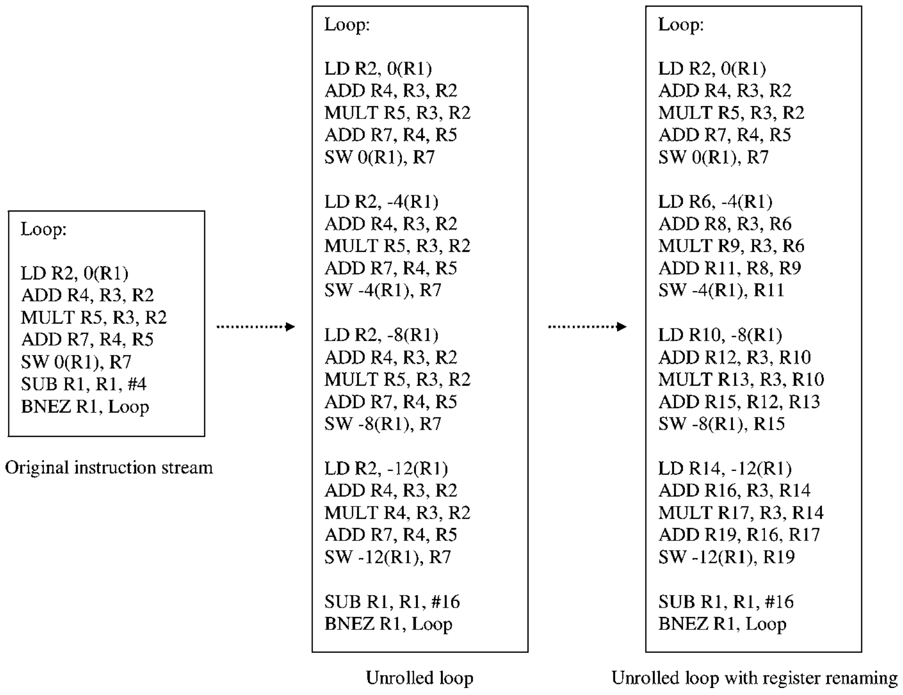
In conclusion, we can say that VLIW architectures do not have the hardware complexity of current superscalar architectures. However, they impose a much heavier burden on the compiler than superscalar architectures. This means that there exists a clear trade-off between the two approaches in the current high-performance microprocessor design. In the following part, more detailed techniques for the VLIW architecture is explained.

5.1.1 Loop Unrolling

Loop Unrolling [11] is one basic technique to find more Instruction Level Parallelism out of the sequential instruction stream. It is based on the idea of expanding the multiple instances of a simple loop-body. Then, a wider range of instructions is at our disposal since we can have more instructions from the multiple iterations. It is most beneficial to unroll loops when the loop body does not include any complex control flow.

As seen in Figure 16, the original loop body has a sequence of seven instructions. Therefore, without loop unrolling, the compiler can only pick from a pool of seven instructions to uncover parallelism (assuming there is no branch prediction). Loop unrolling will maximize the size of instruction window across the control flow limits by expanding the loop body. The loop body can be expanded by unrolling the several loop iterations. For example, in Figure 16, the next four iterations of the loop can be integrated as the single loop of the second diagram. Now, the basic block has four times many instructions than before. Therefore, more instructions can be examined for possible parallel execution.

Although we can have more instructions after loop unrolling, there are name dependencies between iterations. To reduce the false dependencies between instructions (since they may use the same register names), register renaming is imperative before the final instruction stream can be obtained. However, some instructions might have loop-carried dependencies, which occur that later iterations of the loop body depend on the computation results of the previous iteration. Therefore, those dependencies must be carefully respected during register renaming.



```
Loop:

LD R2, 0(R1)
ADD R4, R3, R2
MULT R5, R3, R2
ADD R7, R4, R5
SW 0(R1), R7
SUB R1, R1, #4
BNEZ R1, Loop
```

Original instruction stream

```
Loop:

LD R2, 0(R1)
ADD R4, R3, R2
MULT R5, R3, R2
ADD R7, R4, R5
SW 0(R1), R7

LD R2, -4(R1)
ADD R4, R3, R2
MULT R5, R3, R2
ADD R7, R4, R5
SW -4(R1), R7

LD R2, -8(R1)
ADD R4, R3, R2
MULT R5, R3, R2
ADD R7, R4, R5
SW -8(R1), R7

LD R2, -12(R1)
ADD R4, R3, R2
MULT R4, R3, R2
ADD R7, R4, R5
SW -12(R1), R7

SUB R1, R1, #16
BNEZ R1, Loop
```

Unrolled loop

```
Loop:

LD R2, 0(R1)
ADD R4, R3, R2
MULT R5, R3, R2
ADD R7, R4, R5
SW 0(R1), R7

LD R6, -4(R1)
ADD R8, R3, R6
MULT R9, R3, R6
ADD R11, R8, R9
SW -4(R1), R11

LD R10, -8(R1)
ADD R12, R3, R10
MULT R13, R3, R10
ADD R15, R12, R13
SW -8(R1), R15

LD R14, -12(R1)
ADD R16, R3, R14
MULT R17, R3, R14
ADD R19, R16, R17
SW -12(R1), R19

SUB R1, R1, #16
BNEZ R1, Loop
```

Unrolled loop with register renaming

FIG. 16. An example of loop unrolling.

5.1.2 Software Pipelining

Software pipelining is a technique which enables the later iterations of a loop to start before the previous iterations have finished the loop. Therefore, the later iterations of a loop can be executed, pipelined with the previous iterations. Of course, the loop unrolling technique also achieves the same scheduling as software pipelining, which is merely a subset of loop unrolling. However, in loop unrolling, a simple form of instruction pool is obtained from the multiple instances of the same loop body. The compiler can then schedule each of the instructions in turn. On the contrary, in software pipelining, the parallelism is obtained between the different parts of the same loop body by executing later iterations a couple of clock cycles later.

The main advantage of software pipelining over a simple loop unrolling technique is the following; loop unrolling requires overhead for each instance of an unrolled loop, whereas software pipelining only imposes start-up and wind-down overhead. For example, assume in a simple example that there are 100 iterations and that the loop body contains 4 instructions for the normal computation and 2 instructions for its control flow. If we unroll the loop over 4 iterations, this results in $16 + 2$ instructions for an extended loop (the loop which includes 4 iterations in this example). Just assume that four iterations of an instruction can be executed in parallel, and the parallel execution of four instructions takes one clock cycle. Therefore, in the loop unrolling case, one unrolled loop execution requires 4 cycles for the computation and 2 cycles for the control instructions. Since the unrolled loop needs to be iterated 25 times, the total required cycle time is 150 cycles. However, we first can eliminate the control flow instructions in the software pipelining. Then, the first iteration starts alone and the second iteration starts when the second instruction of the first iteration starts. Both operations can be executed in parallel. Since execution can support up to four instructions in parallel, the fifth iteration can start just after the first iteration finishes (recall that one loop execution includes the execution of four instructions). In this particular case, the 100th iteration starts at 100 cycles and it requires 4 more clock cycles to finish. Therefore, 104 cycles are required for the overall execution.

Software pipelining sometimes provides an easier approach to parallelize a loop with loop-carried dependencies. In the previous example, there are no loop-carried dependencies. Therefore, each iteration of the original loop can be executed in parallel. However, if there existed loop-carried dependencies, those would have to be carefully taken into account. Figure 17 shows one example of software pipelining which includes a loop carried dependency; the second instruction *Add* $R4$, $R5$, $R2$ requires the updated value of $R5$ in the previous loop iteration. Therefore, simple loop unrolling should consider this dependency to schedule the instruction. As seen in the Figure 17, the software pipelined code can yield some added ILP. As seen in the example, software pipelining can be an easier approach to schedule parallel execution of a loop body with loop-carried dependencies; however, if the distance between the instructions for loop-carried dependencies is too high, software pipelining does not provide much of a benefit: the later iteration cannot be executed until the previous iteration finishes and provides the results.

5.1.3 Trace Scheduling

Loop unrolling and software pipelining is quite beneficial to extend the range of the instruction window for a simple loop body. However, when the loop contains very complex control flow, it cannot be easily unrolled. The *Trace Scheduling* [8] technique has been developed to find more ILP across the basic block boundary even with a complex control flow. *Trace* is considered a large extended basic block considering the control path which is frequently taken [2,21]. This technique considers the branch instructions as jump instructions inside a trace, and it schedules the instructions inside the trace as a large basic block.

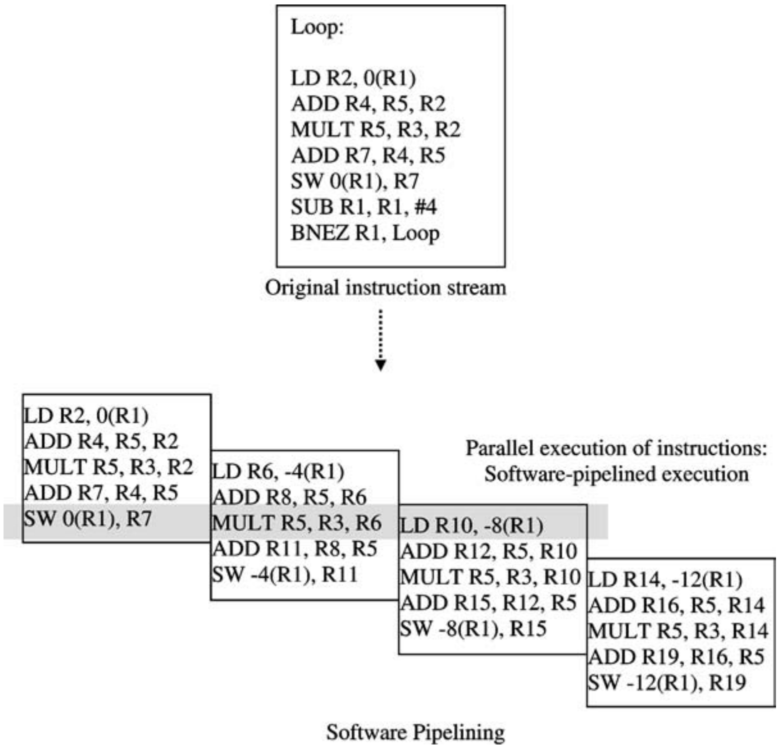


FIG. 17. An example of software pipelining.

However, to guarantee the correct computation of the other path execution, Trace Scheduling requires some compensating code to be inserted for the case when the program flows out of the trace. Trace Scheduling depends on the frequent occurrence of the Trace path. Since the VLIW compiler decides traces in a static way, the traces may not be sufficiently well predicted. Therefore, if the predicted path execution does not happen frequently, the compensation code overhead can be higher than the advantage given by trace scheduling.

The above three techniques can be used to increase the amount of parallelism by extending the instruction pool. All three approaches strongly depend on an appropriate and accurate control flow prediction; using them is beneficial only when the behavior of branches is predictable by the compiler. To cope more aggressively with dynamic events inside the control flow, ISA supported techniques such as conditional instructions and predication have been introduced in addition to the traditional VLIW concept. We will cover those techniques in the example architecture section.

5.2 Design Challenges

The largest challenge for VLIW architecture is the code explosion problem. Code explosion is caused by the need for the compiler to insert NOPs in the object code when the available parallelism of the code is not sufficient to fill each VLIW instruction. These NOPs may waste considerable amount of bits in the VLIW code.

Also, a large number of registers are required in the VLIW architecture due to transformations such as loop unrolling, register renaming, and trace scheduling. In addition, a large data transport capability between functional units and register file is required. Since the VLIW instructions are, by definition, very long, they will require a high bandwidth between the instruction cache and the fetch unit.

Although static scheduling has several advantages over dynamic scheduling, pure VLIW has not enjoyed a successful commercial implementation. The main weakness of the architectures has to do with backward compatibility; conventional compiler or binary must be considerably modified in the VLIW architectures. In fact, superscalar architectures have a huge advantage due to the compatibility which is one reason why they have been chosen by almost all processor vendors [32].

Indeed, the VLIW instruction should be adjusted to the hardware configuration. Therefore, the compiler should be notified the hardware characteristic in advance. It consequently means that there is not even compatibility between different VLIW processors.

5.3 Example Architectures

5.3.1 The IA-64 ISA and Itanium Processor

The Intel Itanium [28] is the first commercial processor which implements the IA-64 ISA [12]. The IA-64 ISA is a 64-bit instruction set architecture implementing EPIC (Explicitly Parallel Instruction Computing) [27], a design which is a joint project of Intel and HP. Although the Itanium processor is not a pure VLIW processor, many features of Itanium resemble that of a VLIW architecture.

First of all, a 128-bit wide instruction bundle is used for the long instruction word. Indeed, a single bundle holds three instructions. Each base instruction requires 41-bit wide, and 123 bits on a bundle are used for those three instruction. The remaining 5 bits can be used as the template [28]. Those template bits are used to deliver the execution information of the bundle. The template bits are very useful to avoid empty operations (NOPs), and allow higher flexibility in the exploitation of parallelism. Figure 18 shows the format of a three-instruction bundle.

The IA-64 provides 128 65-bit general registers with 65 bits for each register and 128 82-bit floating-point registers. It also has 64 1-bit registers designed to indicate

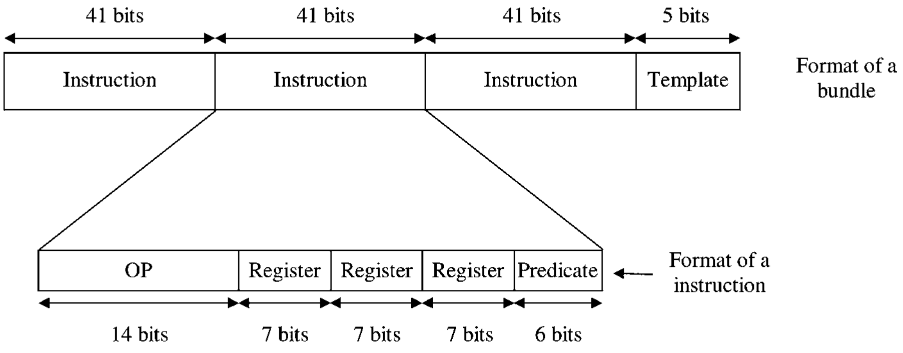


FIG. 18. Format of a instruction bundle.

the result of conditional expression instructions. In addition, eight 64-bit branch registers are implemented for function call linkage and return. This ISA also provides space for up to 128 64-bit special-purpose application registers.

Two distinct features of the IA-64 are predication and load speculation. Originally, the predication requires the instructions for predicated execution. For example, the instruction itself detects a conditional variable to decide whether the instruction can be executed or not. In other words, the instruction is executed only when its predication is true. However, in the IA-64 architecture, all the branch paths are eagerly executed and the instructions inside those paths are committed only when the predication is true. Under the predication, the execution of each instruction needs to be accompanied by a dedicated predicate register among 64 predicated registers. Then, the instructions can only be committed when the specified predicate register is confirmed as *true*.

Figure 19 shows a simple example of the predicated execution. The *CMPEQ* instruction sets the value of the predication registers based on the comparing operation. Then, the later predicated instructions are committed according to the contents of the dedicated predication registers. The predication in IA-64 means both paths of a branch instruction must execute in parallel. After finding the branch result, only the correct path execution is committed and can affect the program execution. Indeed, it is very similar to the multi-path execution or eager execution [33].

The other technique called speculative load execution of the IA-64 enables to move load instructions across the control flow boundary. Speculative loading, indeed, has been proposed to reduce the access latencies of load instructions. For speculative loading, a load instruction is converted into two other instructions: the speculative load instruction which is moved before a branch instruction and the checking instruction which stays in the original location to check the proper execution of the program flow. An example of the speculative load execution is shown in Figure 20.

Source Code

```
if (a = b)
a = 0;
else
b = 0;
```

Normal Code without Predication

```
beq r1, r2, L1
J    L2
L1:
move r1, 0
J    L3
L2:
move r2, 0
L3:
```

Predicated Code

```
CMPEQ r1, r2, p1/p2    // check (a == b) sets p1, p2
[p1] MOV r1, 0         // if (p1 == true) r1 = 0
[p2] MOV r2, 0         // if (p2 == true) r2 = 0
```

FIG. 19. Predication example.

Normal Code

```
add r7, r7, r8
mul r4, r5, r6
add r2, r2,r3

beq r3, r4, L2

L1:
lw r9, 0(r7)
add r3, r9, r4

L2:
add r5, r6, r7
```

$\longrightarrow$

Speculative Load Execution

```
lw.s r9, 0(r7)
add r7, r7, r8
mul r4, r5, r6
add r2, r2,r3

beq r3, r4, L2

L1:
chk.s r9
add r3, r9, r4

L2:
add r5, r6, r7
```

FIG. 20. Speculative load execution.

The *lw.s* instruction is a speculative load instruction which is moved across a control flow boundary, and the *chk.s* instruction is the checking instruction in the original location. The checking instruction confirms the validity of the speculative execution.

The first IA-64 processor, Itanium, is implemented with a six-wide and 10-stage pipeline. It is designed to operates at a frequency of 800 MHz. It has four integer units, four multimedia units, two load/store units, three branch units, two extended-precision floating-point units, and two additional single-precision floating-point units. Two bundles are fetched in every clock cycle. Therefore, the machine can fetch six instructions at each clock cycle.

6. Conclusion—Impact on Modern Microprocessors

The three architecture styles just discussed have significantly impacted the design of modern processors, each to their own degree. While superscalar architectures have dominated the commercial field, superpipelining techniques have had a strong influence on the bottom design of processors, a number of VLIW compiler techniques have made their imprint on the design of compilers for conventional architectures.

Superpipelining could appear to be the simplest technique to implement. However, it is inherently tied to low-level implementation issues and is limited by the amount of hardware partitioning which can be attained.

VLIW has the advantage of design simplicity as it pushes much of the complexity to the compiler stage. However, code explosion, requirements for high bandwidth between the instruction caches and the fetch units, and lack of backward compatibility has reduced the potential of this model.

Superscalar has known the most success of all three techniques and has indeed satisfied the users' ever increasing hunger for more computing power. However, it is intrinsically restricted by two basic considerations. First, the ability to uncover sufficient ILP (Instruction-Level Parallelism) is seriously questionable for wide superscalar processors. Second, with larger scale of integration, clock skew and signal propagation delays become a major consideration (a pulse may take dozens of clock cycles to propagate across a chip). This may, in many cases, reduce the effective clock frequency which can be achieved.

Both these problems have been somewhat successfully attacked by a number of hardware and software techniques as we have described. Future models [17] which exploit Thread-Level Parallelism (TLP), include Simultaneous MultiThreading (SMT) and Chip MultiProcessor (CMP). They hold much promise for the design of future architectures.

REFERENCES

[1] Agarwal V., Murukkathampoondi H.S., Keckler S.W., Burger D.C., “Clock rate versus IPC: the end of the road for conventional microarchitectures”, in: *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[2] Allen R., Kennedy K., *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann, San Francisco, CA, 2002.

[3] ARM Limited, *ARM1136JF-S and ARM1136J-S Technical Reference Manual*, vol. 1, ARM, 2003.

[4] Bashteen A., Lui I., Mullan J., “A superpipeline approach to the MIPS architecture”, in: *IEEE COMPCON Spring ’91*, 1991, pp. 8–12.

[5] Bhandarkar D., “RISC architecture trends”, in: *CompEuro ’91, 5th Annual European Conference on Advanced Computer Technology, Reliable Systems and Applications*, May 1989, pp. 345–352.

[6] Cragon H.G., *Memory Systems and Pipelined Processors*, Jones and Bartlett, Boston, 1996.

[7] Farkas K.I., Chow P., Jouppi N.P., Vranesic Z., “The multicluster architecture: reducing cycle time through partitioning”, in: *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.

[8] Fisher J.A., “Trace scheduling: a technique for global microcode compaction”, *IEEE Transactions on Computers* **30** (7) (1981) 478–490.

[9] Gowan M., Biro L., Jackson D., “Power considerations in the design of the Alpha 21264 microprocessor”, in: *Proceedings of the 35th Design Automation Conference*, June 1998.

[10] Heinrich J., *MIPS R4000 Microprocessor User’s Manual*, second ed., MIPS Technologies, Inc., Mountain View, CA, 1994.

[11] Hennessy J.L., Patterson D.A., *Computer Architecture: A Quantitative Approach*, third ed., Morgan Kaufmann, San Francisco, CA, 2003.

[12] Huck J., Morris D., Ross J., Knies A., Mulder H., Zahir R., “Introducing the IA-64 architecture”, *IEEE Micro* **20** (5) (September 2000) 12–22.

[13] Hwang K., *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, New York, 1993.

[14] Intel Corporation, *IA-32 Intel Architecture Software Developers Manual*, vol. 1, Intel Corporation, 2004.

[15] Jouppi N.P., “The non-uniform distribution of instruction-level and machine parallelism and its effect on performance”, *IEEE Transactions on Computers* **38** (12) (December 1989) 1645–1658.

[16] Jouppi N.P., Wall D.W., “Available instruction-level parallelism for superscalar and superpipelined machines”, in: *Proceedings of the Third Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 272–282.

[17] Kavi K., Lee B., Hurson A., “Multithreaded systems”, in: *Advances in Computers*, vol. 46, 1998, pp. 287–327.

[18] Krishnan V., Torrellas J., “A clustered approach to multithreaded processors”, in: *Proceedings of the International Parallel Processing Symposium (IPPS)*, March 1998.

[19] Kunkel S.R., Smith J.E., “Optimal pipelining in supercomputers”, in: *Symposium on Computer Architecture*, 1986, pp. 404–414.

[20] Moon S.-M., Ebcioglu K., “An efficient resource-constrained global scheduling technique for superscalar and VLIW processors”, in: *The 25th Annual International Workshop on Microprogramming*, December 1992.

[21] Muchnick S.S., *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, CA, 1997.

[22] Palacharla S., Jouppi N.P., Smith J.E., “Complexity-effective superscalar processors”, in: *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[23] Pan S.-T., So K., Rahmeh J.T., “Improving the accuracy of dynamic branch prediction using branch correlation”, in: *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOSV)*, October 1992.

[24] Patterson D.A., Hennessy J.L., *Computer Organization and Design: The Hardware/Software Interface*, second ed., Morgan Kaufmann, San Mateo, CA, 1994.

[25] Patterson D.A., Hennessy J.L., *Computer Architecture: A Quantitative Approach*, second ed., Morgan Kaufmann, San Mateo, CA, 1996.

[26] Ramamoorthy C.V., Li H.F., “Pipeline architecture”, *Computing Surveys* **9** (1) (March 1977) 61–102.

[27] Schlansker M.S., Rau B.R., “EPIC: explicitly parallel instruction computing”, *IEEE Computer* **33** (2) (February 2000) 37–45.

[28] Sharangpani H., Arora K., “Itanium processor microarchitecture”, *IEEE Micro* **20** (5) (2000) 24–43.

[29] Srinivasan S.T., Lebeck A.R., “Load latency tolerance in dynamically scheduled processors”, in: *Proceedings of the 31st Annual International Symposium on Microarchitecture*, November 1998.

[30] Thornton J.E., “Parallel operation in the control data 6600”, in: *Proceedings of the Fall Joint Computers’ Conference*, vol. 26, October 1961, pp. 33–40.

[31] Tomasulo R.M., “An efficient algorithm for exploiting multiple arithmetic units”, *IBM Journal* (January 1967) 25–33.

[32] Tyson G., Farrens M., Pleszkun A., “Misc: a multiple instruction stream computer”, December 1992.

[33] Uht A.K., Sindagi V., Hall K., “Disjoint eager execution: an optimal form of speculative execution”, in: *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995.

[34] Šilc J., Robič B., Ungerer T., *Processor Architecture: From Dataflow to Superscalar and Beyond*, Springer-Verlag, Berlin, 1999.

[35] Yeager K.C., “The MIPS R10000 superscalar microprocessor”, *IEEE Micro* **16** (2) (April 1996) 28–40.

[36] Yeh T.Y., Patt Y.N., “Alternative implementations of two-level adaptive branch prediction”, in: *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

Networks on Chip (NoC): Interconnects of Next Generation Systems on Chip

THEOCHARIS THEOCHARIDES, GREGORY M. LINK, NARAYANAN VIJAYKRISHNAN, AND MARY JANE IRWIN

Microsystems Design Lab
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA, 16802
USA

Abstract
Traditionally, the design of on-chip interconnects has been an afterthought in the design process of integrated circuits. As the complexity of interconnect and the capacitance, inductance and resistance associated with the wires have increased with technology scaling, the delays associated with wires can no longer be neglected. Consequently, planning the design of these interconnection networks early in the design stage has become critical in ensuring the desired operation of the integrated circuits. Network on Chip is an on-chip communication mechanism based on packet based data transmission to support early planning of interconnect design. This chapter reviews the various aspects of Network on a Chip and concludes with a case study of a neural network design using such a communication fabric.

ISSN: 0065-2458/DOI 10.1016/S0065-2458(04)63002-6

1. Introduction

Technology scaling has enabled chip designers to increase the number of transistors on a single chip, while reducing the overall area of the chip and the supply voltage. Designs using transistors with smaller feature size also enables faster switching response, an effect which has boosted the system clock frequency into the gigahertz era. By the year 2008, the International Technology Roadmap for Semiconductors (ITRS) [4] predicts that there will be over a billion gates on a single chip. However, there are various impediments that must be surpassed in achieving the goals set by the ITRS roadmap.

Technology scaling does not magically solve all problems; in fact, designers are now presented with new challenges such as leakage power, transient faults due to noise, and process variation [1–3]. Crosstalk noise, electromagnetic interference noise and radiation induced charged particles become more problematic as the power supply voltage and the channel length are reduced, resulting in transient faults known as upsets. Another issue of major concern is the disproportional scaling of the gate delays vs. the wire delays, as noted in Figure 1, where for values lower than 0.1 μm, the wire delay is expected to grow faster and surpass the gate delay. Wires seem "longer" now, as while everything else shrinks, chip size overall stays almost the same with simply more components placed on it. A wire traveling across the chip is now therefore considered long [1–3,5–10]. While technology scaling reduces the size of a transistor gate and correspondingly the size of an entire cell, long interconnect wires do not scale in similar fashion. With the 10 GHz barrier almost broken, logic gates and consequently IP cores, will end up producing results much faster than the results would travel to the recipient IP core. In other words, a signal will end up needing more time to travel between two components than the components would need to actually compute it. The amount of chip area reachable by any signal within a single clock period reduces drastically, resulting in increased delays when dealing with large centralized structures such as on-chip caches. In addition, the use

Abbreviations

NoC	Network on Chip	**PE**	Processing Element
ITRS	International Technology Roadmap for Semiconductors	**IC**	Integrated Circuit
		MAC	Multiply-Accumulate
IP	Intellectual Property	**VC**	Virtual Channel
DSP	Digital Signal Processing / Digital Signal Processor	**LDPC**	Low-Density Parity Check
		ACK	Acknowledgment
FPGA	Field-Programmable Gate Array	**NACK**	Negative Acknowledgment
ASIC	Application Specific Integrated Circuit	**ANN**	Artificial Neural Networks
		PCI	Peripheral Component Interconnect
QoS	Quality of Service		
SoC	System on Chip	**VME**	VersaModule Eurocard

of the core based design philosophy for designing complex systems-on-chip results in all sorts of data traveling across the chip. For example, DSP cores generating radio signals coexist with control and configuration data, generated by different cores, at different frequencies, with multiple destination cores. Communication consequently needs to be fully distributed with little or no global control, as cores need to initiate data transfers autonomously based on their own needs [1–7].

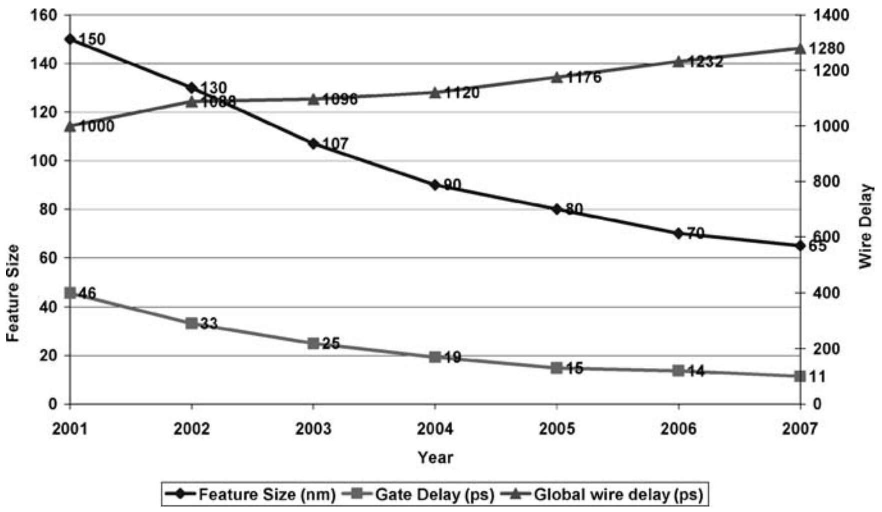


FIG. 1. Impact of technology scaling on gate and wire delays [4].

Consequently, the design of an interconnect structure that transfers data from any source to any destination on the chip, with as low a latency as possible, as high a reliability as possible, and with predictable performance characteristics is critical to the success of complex future system-on-chip designs. Based on these needs, researchers, inspired by the way traditional networks work, proposed a novel communications structure for on-chip interconnections [1] called Networks-On-Chip (NoC). NoC architectures contain on-chip routers attached to each IP core to support on inter-IP communication. The architecture attempts to separate two functions of a design; communication and computation. The overall architecture is similar to traditional network architectures, and is also partially inspired by the FPGA routing architecture. Similar to traditional networks, NoC architectures depend on fundamental concepts such as congestion avoidance and control, routing algorithms, data buffering and reliability. While traditional networks have made significant progress in addressing these issues, using algorithms and techniques developed for traditional networks to address these issues for on-chip networks requires a re-evaluation given the vast differences in the design constraints [10–12].

The basic NoC structure is similar to traditional networks [5–7]. Communicating nodes interface to a router, and each router communicates with neighboring routers, networking thus the entire chip. The communicating nodes are the IP cores in our case, also known as Processing Elements (PEs). The original idea [1–3] calls for a switch dedicated to each PE placed on the chip, and a network structure joining all switches to each other as shown in Figure 2.

Switches communicate with each other via short and regular wires. The uniformity of the interconnect design enables the architect to predict the wire delays much earlier in the design cycle and optimize the design to meet the desired constraints more efficiently. In contrast, the length of point to point interconnects used in traditional ASIC design cannot be accurately predicted until after physical design is completed. Further, the NoC structure permits multiple data transfers to occur concurrently between the different on-chip routers, unlike bus designs, where the communication medium may only be accessed by one sender at a time. Data is transmitted in packets, where a packet is made up of the header and the data fields. The header contains addressing and control information similar to traditional networks, and the data is the actual data that would be transmitted in traditional bus architectures [2].

To give a better idea on how NoC operates, let us consider a simple example network structure shown in Figure 3. We partition a chip into a 4×4 array of 16 tiles. Each tile receives user-specific components/PEs (IP cores) such as embedded processors, DSP, memory cores, etc. Each component is placed into a tile. Each tile now communicates with others via the underlying network structure; there is no top-level connection other than the network wires. Network logic occupies a very small area between the tiles. The network interface acts as a simple reliable datagram interface

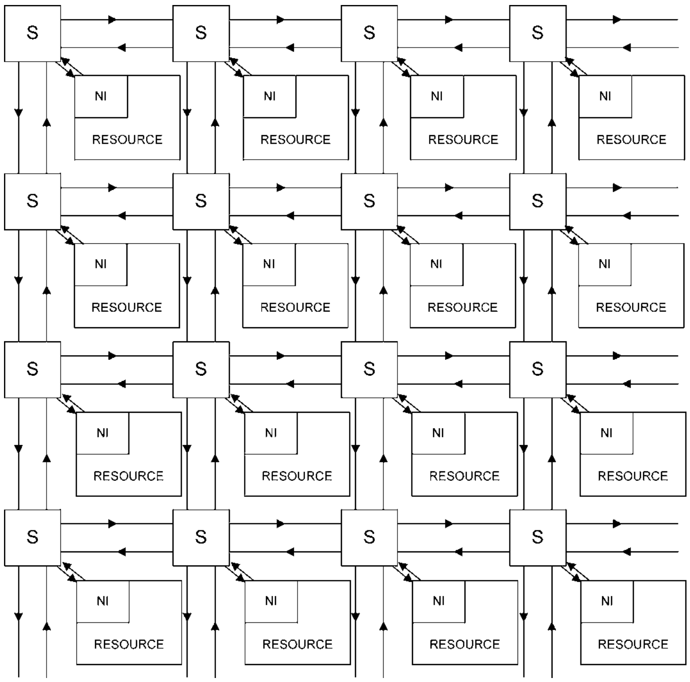


FIG. 2. NoC basic architecture showing the switches interfacing to each resource. Resources include pre-designed IP cores, or user-designed components [1–3].

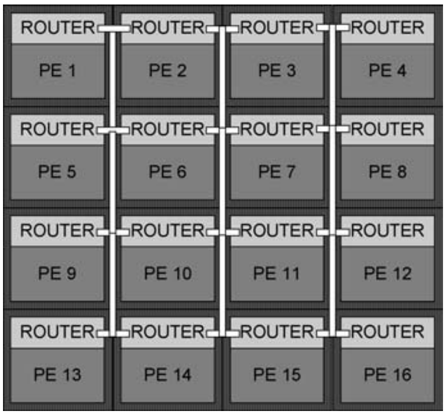


FIG. 3. Tiled NoC architecture (4 × 4), where routers communicate with each other via a higher metal layer and resources (Processing Elements—PEs) are attached to each router.

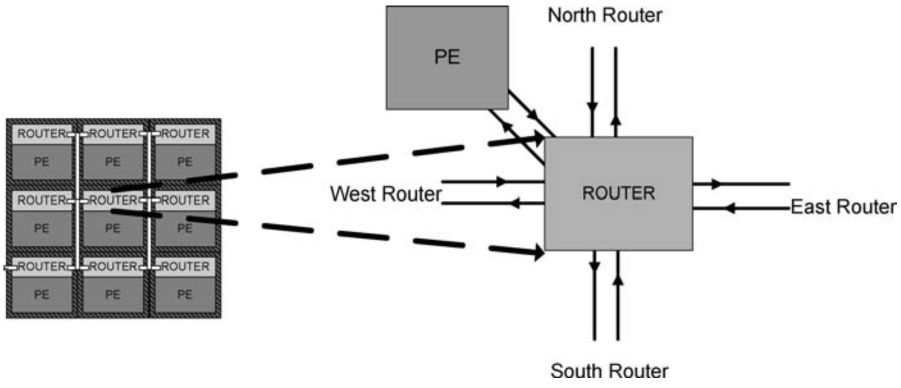


FIG. 4. NoC router diagram and its connections.

to each tile. Each tile has an input port to insert packets into the network and an output port to receive packets from the network. Data transfer is done via routers, attached to each tile. The router interfaces to the rest of the network via 5 ports—the PE port, and the 4 networking ports, north, south, east and west as shown in Figure 4 [2].

The router logic is typically kept very simple so that the overall network area remains as small as possible. It consists of a crossbar, a routing decision unit, and buffers, which occupy most of the router area. The router transfers data in *flits*; a *flit* is defined to be the amount of data transmitted in a single clock cycle. A packet, depending on its size, can be transmitted in several flits. Flits are assembled and disassembled at the PEs, thus router logic is kept minimal. When a flit arrives in the router, the router uses the flit header information to route the flit to the destination output port depending on the destination PE and its relative location on the chip. The routers provide buffer space, which reduces network congestion and improves performance. The size of the buffers depends on the overall network architecture and application needs. Routers also act as drivers, receivers and repeaters across the chip, reducing the area overhead of the design [2]. The network architecture also allows for fault tolerance protocols and wiring, as well as a simplified, regular way of applying power optimization. In addition, routers can also be configured to prioritize packets, ensuring quality of service (QoS) levels for applications where performance must be well characterized [16–20].

The NoC architecture offers significant advantages when compared to traditional bus architectures. It provides significant improvement over bus synchronization, given that routers can act as “pipeline stages” across long wires. The NoC architecture is more easily expandable and can be easily reconfigured to support different communication patterns or to handle faults. In contrast to expanding the interconnect system by plugging in PEs to standard backbone busses such as PCI or VME, the system now simply expands by inserting a new switch to the network, and attaching a PE to the switch. The NoC design can be implemented hierarchically, similar to the traditional network layer hierarchy. Hierarchical abstraction allows for a much more accurate model for purposes of simulation and testing, and enables the designers to control many more parameters in each layer to fine-tune the network for optimal application performance [21–23].

In NoC designs, single synchronous clock regions will span only a small fraction of the chip area. At the same time, miscellaneous self-synchronous PEs can communicate with each other using network-oriented protocols. The structured network wiring leads to a much more deterministic model and accurate electrical characterization. Shorter wiring additionally reduces latency, increasing the clock frequency and potentially increasing the network bandwidth. Furthermore, a better abstraction level can be adopted by using layered communication architecture. Given the presence of noise and other transient fault sources, we can introduce error detection and correction mechanisms to take care of ubiquitous failures due to the inherent unreliable physical medium. By separating the communication infrastructure from the computation nodes, we achieve distributed control of the network traffic and resemble the interconnection architecture of high performance parallel computing systems, on a single chip [24].

In this chapter, we will discuss the NoC architecture and its major highlights in Section 2. Section 3 will present the communication structure employed in NoC architectures, and Section 4 will introduce application mapping on NoC architectures. Power consumption and reliability of NoC-based systems are highlighted in Sections 5 and 6 respectively, and Section 7 presents some design tools used in evaluation of NoC architectures and relevant research. Section 8 presents an example application mapped on NoC architecture, to illustrate the concepts presented in this chapter. Finally Section 9 concludes by summarizing the advantages of NoC architecture and its promising future towards improved SoC design, and emphasizes potential research areas.

2. NoC Overview

2.1 Hardware Overview

2.1.1 NoC Routers

The primary component for NoC architectures is the on-chip switch/router. The router resembles the traditional router/switch concepts; however the environment

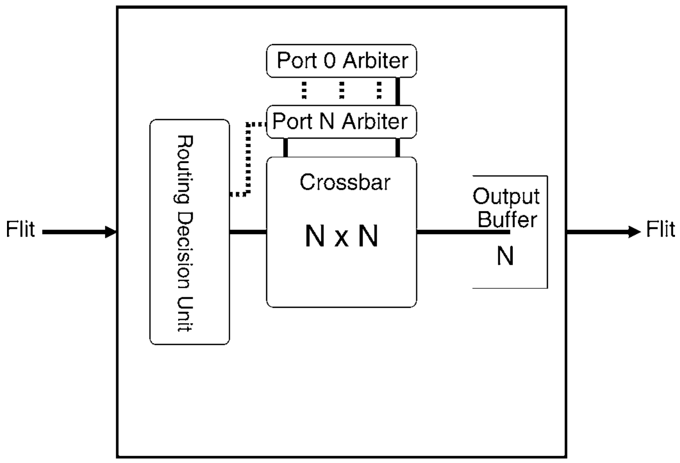


FIG. 5. Router detailed block diagram.

and operation differ drastically. While traditional networks enjoy plenty of routing flexibilities such as varying routing algorithms, NoC routers are limited in terms of buffering space, complexity and latency, and are designed with minimal options to facilitate application requirements. Typically a router consists of the port dedicated to the PE, and four ports to talk to the rest of the networks [5]. An example block diagram of a NoC router is shown in Figure 5. NoC routers communicate between each other using point-to-point wires, whose length however becomes a design parameter now; therefore it can be modeled more accurately. The router contains sufficient logic to perform the routing operation as well as related data transfer logic such as error correction/detection, buffering space, packet prioritization, etc. Routers also act as repeaters and bridges between various sections on the chip, further reducing the global wiring length [2,17–20].

On-chip routers/switches are responsible for multiple operations depending on the application. They direct data from and to each PE, provide buffering capability, resolve priorities, provide fault tolerance capabilities, and integrate computation and communication data along the network. The basic components of the router are the input/output ports, the crossbar switch, the buffers and the routing unit. The input/output ports receive and send data to the external world. The buffers keep data waiting to be routed, minimizing data loss and reducing congestion. The routing unit is responsible for directing the data between input and output ports, and finally the crossbar is the data redirection unit [1–7]. The router can be pipelined or non-pipelined. A pipelined router can be clocked faster and possibly include stages for error detection and correction, priority routing, etc. A non-pipelined router forwards received data within one clock cycle, and requires minimal resources. The router design is of course constrained by the area of the chip; as the network hardware has to be minimal, the router has to consume as little area as possible. Therefore, depending on application requirements, it is desired to provide only minimal and absolutely necessary buffering as well as any other logic required for routing operations [17–20].

2.1.2 Input/Output Interface

The network can communicate with the external world using dedicated I/O nodes, namely ingress and egress nodes. These nodes can be either PEs or independent nodes. The ingress node(s) receive data to be transmitted to PEs from the external pins, while the egress node(s) collect data received from PEs to send outside the network. Note that each ingress or egress node connects to a network router, where data is sent or received from in packetized format in agreement with the network protocol. Data sent to the network is not packetized, so ingress nodes are responsible for packetizing the data in accordance with the network packet protocol and forward the data to the network when the network is ready to receive the data. Ingress nodes additionally prevent network congestion from happening, by either buffering any input data that is not ready to enter the network (which requires sufficient buffering capabilities) or rejecting any data that is sent to the network in the event that the network is congested. In similar fashion, egress nodes provide buffering capability for data leaving the network, but cannot be accepted by the external world due to any reason, thus maintaining the input-output flow without loss of data [15].

Given that the NoC interconnect is not part of the computation structure, in VLSI implementation, the wires between the routers can connect be routed using one or two dedicated metal layers [2]. Doing so, allows for routing wires to lie “on top” of the chip, enabling the use of differential signals and shielding wires for reducing noise induced transient faults. Using higher metal layers to connect router wires also reduces the need for wire drivers and repeaters resulting in an overall positive effect on the chip communication.

2.1.3 Processing Elements (PEs)

The purpose of building a NoC structure is to connect multiple IP cores (also called Processing Elements (PEs) in this chapter) that can be either homogeneous or heterogeneous. A PE can be any component that performs computation based on the application mapped on the chip. The PE receives data from other PE(s) via the network infrastructure, performs its assigned computation task and returns the data to the appropriate PE(s) or the network output. The PEs interface to the network via specific network interface logic built in each PE. Based on the network communication protocol that the designer chooses, each PE contains hardware to process arriving network data, by stripping off the packet header, identifying the data to be used for computation, and processing that data [17–20]. When a PE wishes to send data to another PE, it uses hardware similarly designed based on the network communication protocol to packetize the output data and send it to the network. Based on the network congestion, a PE can only send data when the network is ready to accept it. It must be noted that some NoC architectures drop packets when there is buffer overflow, but this depends entirely on the design [42,43]. At the same time, traditional protocols which employ acknowledgement methods (ACK/NACK) are not practical with on-chip networks, due to latency and hardware issues; therefore congestion control at the PE level becomes important. The PE network input/output interface hardware is sometimes called PE wrapper, as it forms a virtual wrapper where the PE (IP core) can be placed. The PE wrapper hardware typically consists of an input and an output interface. The input interface usually contains a packet assembler (where packets are assembled from incoming flits) and a data redirector, where the header information is decoded and the data included in the packet is identified and redirected to the computation unit inside the PE. At the output interface, the wrapper contains a table with the destination addresses of the PEs that it communicates with, and other relevant control information in order to packetize the output data and send it to the network. The network interface hardware has to be as small as possible, and appear invisible to the PE computation core [15,24].

PEs are addressed using address models similar to the traditional network addressing schemes. Each PE has a unique address ID, assigned a priori by the application, and made known to all associated PEs during the network configuration. Consequently, PEs store address information for all other PEs that they communicate with using dedicated memory. Ingress and egress nodes also are addressed similarly. An important concept of NoC architecture is the configuration stage. Addressing and other network relevant information is sent to all PEs with details such as addresses of PEs that will be communicated with, addresses of ingress and egress nodes, computation specific data, and normal PE initialization data. The process is essential so that PEs will be able to transmit data to and from the network, be able to follow the network [15].

We will explore further the network hardware and illustrate how the design parameters affect the performance of the network, during our extensive analysis of the communication structure(s) in Section 3. A brief overview however follows next.

2.2 Communication Overview

2.2.1 *Data Transfer: Packets vs. Flits*

The overall communication happens via the switching activity of the network infrastructure. The major object traveling across the network is the data packet. The packet consists of two fields, header and data. The header is used to identify the data that arrives at a destination. The header contains information fields such as control information, priority, sequence number (if the data packet is part of a larger data block for example), source and destination address, error correction/detection fields, etc. Similar to traditional networks, header bits add identity and aid in smooth traversal of the network. Header bits are used by the router in the routing procedure, used by the recipient PEs for data identification and classification, and used by the ingress/egress nodes to identify input/output data. Header bits however are extra bits, adding overhead traffic to the network. In traditional bus architectures, only necessary data and possibly error control bits travel across the bus. Also, increase in data transmission/switching results in increased dynamic power consumption. Therefore, headers have to remain as small as possible, requiring the least amount of bits as per application demands. However, allocating some extra (and initially unused) bits in the header field offers the designer a particular advantage; the designer can reconfigure the application after the network is designed, something that is used in traditional networks.

The smallest transmissable unit that travels between two routers is called a flit [2–7]. A packet can consist of multiple flits, or a packet can be itself a flit. In cases where a packet is made up of multiple flits, there are a number of ways that the packet can be routed, including packet switching and wormhole switching. In all cases, the header information is usually transmitted with the first flits, allowing the receiving router to begin computing the next destination early. The subsequent flits follow the path discovered by the first flit, ensuring that they all arrive in the same order and with no intermediate flits from another packet. Flits can be assembled into packets at either the router, or the PE. Assembling at the router is known as packet switching, and is not usually done in NoC designs because waiting for all flits to arrive before continuing transmission consumes extra hardware (buffering) and latency. Assembling only has to be done in cases where two (or more) networks communicate with each other and packet formats differ between the networks. This technique is similar to the *IP tunneling* in traditional networks [2], and allows data exchange between two or more networks using separate protocols. On-chip networks have not yet evolved to the point where more than one network resides on a chip; designers however will eventually be given that option as the technology scales down.

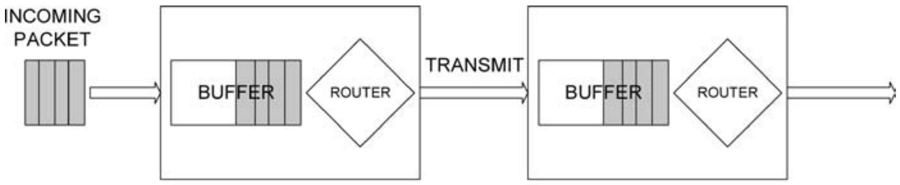
Flits which travel across the network arrive at a router via one of the input ports. They are queued and routed based on the header field that is contained in the flit.

The router can either store incoming flits until the output port is able to send to the network where the router forwards the flit to its destination port, or, reject a flit from entering the router queue if there is no space and the network is congested. If the router contains sufficient buffering capacity, data loss is avoided; else mechanisms such as acknowledgments and negative acknowledgments can be employed to guarantee service. Whether packet loss can be tolerated or not depends on the application mapped onto the NoC. When the PE is about to transmit a packet, it breaks it down into flits attaching header data, and transmits each flit until the entire packet is gone. Initial NoC implementations used a packet as a flit; in this case, no packet assembly is required. However, breaking down packets into flits allows for smaller buffer sizes and lower wiring requirements. On the other side, breaking a packet into multiple flits increases the latency of transmission, and can result in increased congestion. This is an important tradeoff that designers have to consider.

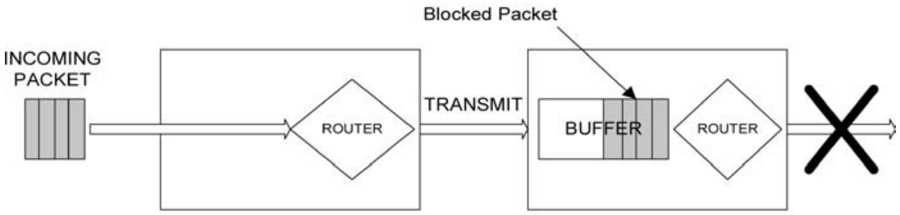
2.2.2 Routing

Data transmission can occur at every routing node, where each node is responsible for receiving a flit/packet and sending that to a destination node. Most networks include some system to ensure that any message sent will eventually arrive at its destination, as the complete loss of information will negatively affect most applications. Data transfer from node to node can happen using one of the following switching methods: *store-and-forward*, *virtual cut-through* and *wormhole routing*. In *store-and-forward* switching, a packet is received and stored in its entirety before it is forwarded to the next network destination. In other terms, the last flit of each packet must be received before the first flit of said packet is able to begin being transmitted to the next router. The router has to provide sufficient buffering capacity for each packet, and the method implies latency of at least the time required to receive the entire packet, times the number of routers the packet travels. This can be too costly when dealing with multi-hop paths. This method of routing is useful in networks where each packet is checked for integrity at every router, as it allows a single error coding to protect and repair an entire packet [15–24]. An example is shown in Figure 6(a).

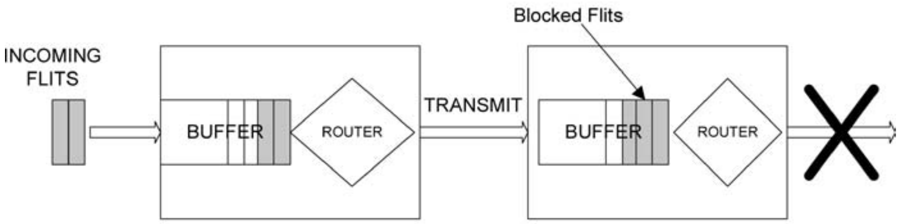
Virtual cut-through switching begins forwarding a packet as soon as the destination router/PE can guarantee that the entire packet will be accepted by the destination router. If the destination (either a router or a PE) cannot guarantee full reception of the packet in its entirety, then the packet is stored in the current router as a whole until the destination node is ready to accept it. The *virtual cut-through* algorithm is similar to the *store-and-forward* method in terms of buffering requirements, however allows a faster communication as a packet can immediately begin being transmitted, provided the destination can receive it. Contrary to *store-and-forward* routing, the



(a) Store and forward routing



(b) Virtual Cut-Through



(c) Wormhole routing

FIG. 6. Switching techniques.

network latency depends on the congestion levels (and hence buffer utilization) in the network [15–24]. An example is shown in Figure 6(b).

Lastly, in *wormhole switching* packets are split into flits and allowed to travel forward, even if sufficient buffer space for the entire packet is not available. Flits are routed as soon as the destination router/PE can guarantee acceptance of even a single flit, even when not enough buffering capacity exists for an entire packet. The moment the first flit of a packet is sent in an output port, the output port is reserved for flits of that packet, and when the leading flit is blocked, the trailing flits can be spread over multiple routers blocking the intermediate links. As a result, while wormhole routing offers significant advantages in terms of both required buffer space and minimum latency, it is more susceptible to congestion and the designer has to be extremely careful to maintain performance as network utilization increases. An example of wormhole routing is shown in Figure 6(c). The latency of wormhole routing is proportional to the sum of packet size (in flits) and number of hops to the destination, rather than the product of packet size (in flits) and number of hops, as in store-and-forward packet transmission [15–24].

A major problem that NoC architectures have to deal with is deadlock when routing data. Deadlock occurs when multiple packets/flits wait for interdependent resources to be released for progressing without being able to escape, as shown in Figure 7. There are two types of deadlocks that can happen in NoCs, *buffer* and *channel* deadlock. *Buffer deadlock* occurs when all buffers are full in a *store-and-forward* network. This leads to a circular wait condition, where each node is waiting for space availability to receive the next message, but as message x cannot progress, it will never release its buffer space to message $x + 1$. As such, a circular wait state occurs, in which no packet advances, and no buffer space is opened for the packet ‘after’ it to advance. *Channel deadlock* is similar, but will result if all channels around a circular path in a wormhole based network are busy (each node has a single buffer used for both input and output) [16–24].

Deadlock can be avoided in a number of ways, and extensive research has been performed in this area in regards to traditional networking. One possible method of avoiding deadlock is to prevent circular dependencies by carefully selecting a routing

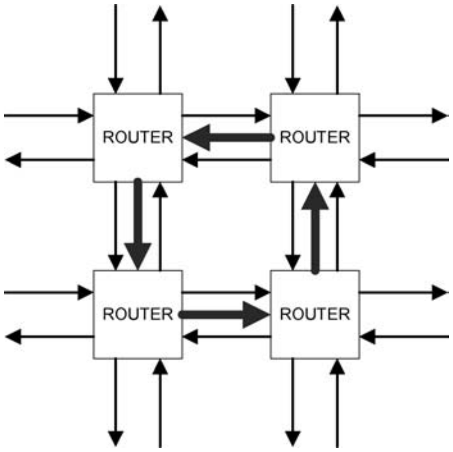


FIG. 7. Deadlock in NoC.

algorithm in such a way that it avoids creating data cycles among routers. Alternatively, designers can choose to implement extra channels per routing node, called *escape* channels, which also do not allow circular dependencies. The presence of escape channels ensures that in a situation where a number of packets are competing for each other's resources, at least one of said packets will be able to make forward progress (via the escape channel), ensuring that resources are freed, and the remainder of the packets can also progress. Escape channels can be either reserved channels implemented in the router, or *virtual channels*, which we will be addressed later in Section 5. The design of deadlock free routing paths is a heavily researched area, as they are key in preventing deadlock from happening all together [17–20].

Due to the nature of the network structure and constrained resources, a poor choice of routing algorithm can be catastrophic to the overall network communication. Each algorithm can be classified in a number of ways, such as either deterministic or adaptive, source or distributed, and unicast or multicast. Deterministic routing requires a priori knowledge of the network topology and connections, whereas adaptive routing algorithms evaluate the available paths across the network and choose one that will provide the best path for a message (according to various metrics). The decision is made on a packet by packet basis and routes may change due to congestion and path failures induced by failure of switches or communication channels. In source oriented routing, the source node that generates the packet/flit also generates the routing path that the flit/packet will follow. This technique is widely used in prioritized packets, where QoS is targeted. Distributed routing on the other hand allows each router to determine the best path for a packet to continue following. Finally, multicast routing is used in the case where the same data has multiple PE recipients, and is transmitted to them in parallel, whereas unicast routing involves a single destination. The application demands again determine the routing scheme to be used. Quality of Service (QoS) is a concept that has also been addressed in NoC architecture. Routing algorithms that offer QoS are preferred in cases where certain data is more time sensitive or important than the rest. As a result, router hardware can potentially be enhanced with QoS directed logic, where packets are identified as high priority and forwarded in front of the queue. Similarly, dedicated routing paths can be allocated to high priority data, helping to ensure prompt and fault-free delivery [17–20].

The main objective is to provide deadlock free and low-latency routing of data from a source PE to a destination PE. Input data is multiplexed to the output ports of each router, using a control circuit which implements the routing algorithm. Routing algorithms are either deterministic, i.e., determined at set-up time, or adaptive, i.e., dynamically adapting to the current network state. Deterministic routing implies knowledge of the network topology a priori, such that each router is configured with the relative location of each PE and the PE's address. Adaptive algorithms use network related information such as congestion or known faulty locations, to choose the best possible path for a packet to reach its destination. Adaptive routing can be accomplished through a number of means, the most common of which involves routing tables at each router that are updated according to the algorithm's particular characteristics. The algorithm selected determines table entries, and the table is used to select which output port data will be sent to. Each table entry contains the relative location of the destination node, the next node/link that the packet has to go to be closer to the destination, and a metric which indicates how good the path is. Metrics can include the number of hops towards the destination, the path delay, path utilization, etc. For example, information concerning the network utilization can be used for adaptive routing, or routing to avoid a broken link. Similarly, the hop count can be used as a deterministic metric in routing towards the destination node [15–20].

Deadlock can be avoided using dimension-ordered routing algorithms in mesh topologies [26]. Dimension-order algorithms route packets entirely in one dimension first, and then in the dimension that the packet has to travel. This technique is essentially similar to the escape channel without the additional hardware. A popular dimension-ordered routing technique is X–Y routing (or Y–X) where a flit/packet is routed as necessary first in the X-direction, and then in the Y direction to the destination (or vice versa). Each router has to know the relative location (i.e., X–Y coordinates) of each destination address, allowing the router to make a decision where to route each incoming flit/packet. In the case of broken links, the algorithm must be able to maintain the X–Y flow to eliminate chances of deadlock occurrence. This algorithm, while easily understood and implemented, is actually overly restrictive, in that while it is a sufficient algorithm for deadlock free operation, it is not necessary. Other algorithms exist that satisfy the sufficient and necessary conditions for deadlock free operation [26,27].

One such deadlock-avoidance algorithm is West-first (East-first) routing. This algorithm allows a packet to follow nearly any path to its destination, on the condition that if the packet wishes to travel in the 'west' direction, it must do so before any other directions are traveled. The algorithm therefore disallows a subset of all available routes, in order to restrict packets to deadlock-free paths. The algorithm provides deadlock free operation and can be fully adaptive.

Dimension-order routing can also be implemented in hierarchical routing; routers are organized in a hierarchical fashion, where a router receives all data destined for a cluster of PEs and routers. That router forwards incoming data for the group that it handles to the group routers, which handle local traffic only. This technique is better suited for networks where groups of PEs talk frequently and usually independently from other groups of PEs. Group communication is isolated from the rest of the network, and is similar to the sub-netting concept of traditional networks.

An adaptive routing algorithm which can potentially result in a deadlock case is the *hot-potato* routing algorithm which is used in traditional networks. In this case, the routers have little or no buffering space to store packets before they are moved on to their next destination. Hot potato routing results in smaller routers, and less complicated routing hardware. In normal routing situations, when multiple packets contend for a single outgoing channel, packets are either buffered internally, and packets that are not buffered are dropped. In hot potato routing, all packets are constantly routed until they reach their final destination, usually on any available output port. Individual communication links can not support more than one packet at a time; consequently packets bounce like a "hot potatoes" sometimes moving further away from their destination since they have to keep moving around the network. This technique allows multiple packets to reach their destinations without being dropped and increases the reliability of the system. This is in contrast to "store and forward" routing where the network allows temporary storage at intermediate locations. Hot potato routing however is mostly used in applications where the criticality of the data traveling across the network is high and the area of the routers is minimal; otherwise, it consumes large overhead bandwidth and much more power than normal routing methodologies. While this algorithm is susceptible to deadlock, the algorithm can also result in a livelock scenario, where a packet will be routed forever, never actually reaching its destination. As such, systems such as a packet lifetime counter, or other livelock avoidance systems, are used [26,27].

Figure 8 shows examples of a dimension-ordered (*X*–*Y* routing), hierarchical routing, and hot potato routing. Hierarchical routing can be dimension ordered (*X*–*Y*, east first, west first, etc.). Hot potato routing is completely random, hence might re-

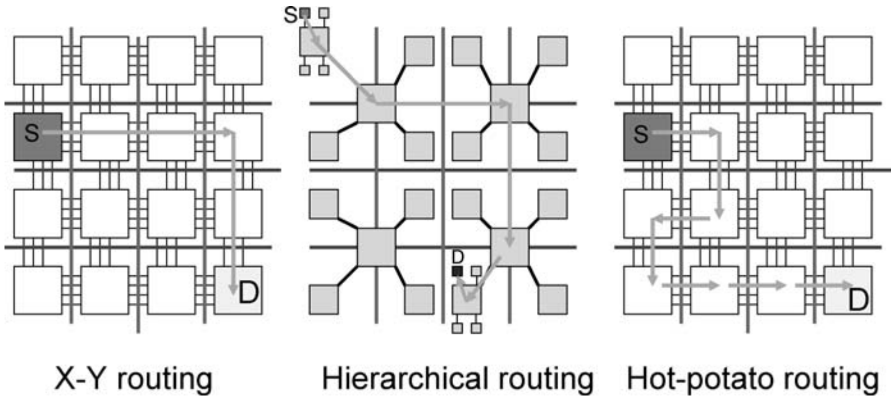


FIG. 8. Routing techniques.

sult in deadlock and large overhead bandwidth as packets travel around the network until they reach their destination.

Generally, routing algorithms involved in traditional networks can also be adopted to work with on-chip networks. Deciding which algorithm will be implemented depends on the application traffic and the chosen network topology. Routing algorithms which work very well with mesh topologies for instance, can result in deadlock in toroidal topologies. Consequently, the designer must carefully evaluate the tradeoffs such as the buffer requirements, the queue sizes and the amount of routing ports prior to deciding which routing algorithm to implement. Routing is in essence a graph theory problem; choosing the right network topology for the application will likely impact the decision on routing algorithm.

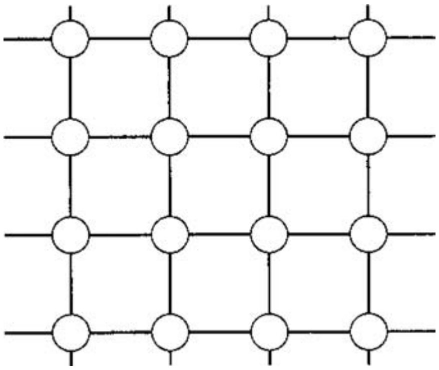
3. Communication Structures

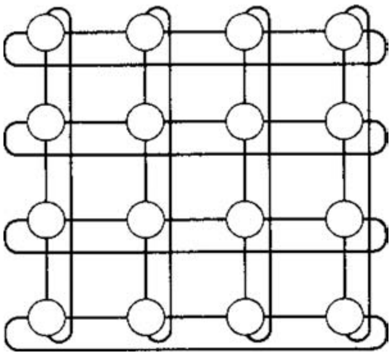
The underlying structure of the NoC architecture is the communication architecture. As the primary purpose of the network is to provide the best communication between the PEs, the architecture that defines that is obviously the most important concept. In this section, we will cover network topology choices, buffer and crossbar design issues, virtual channel implementations, and how each parameter contributes to the overall network performance.

3.1 Network Topologies

Traditional network topologies have been adopted by NoC researchers, as their properties in terms of network performance are already known. However, when dealing with VLSI, we are limited in terms of the 3rd dimension; therefore the topology is constrained by the amount of metal layers available to us. The most common topologies used are the two-dimensional mesh structure, the two-dimensional torus structure, and the three-dimensional mesh structure. All three are shown in Figure 9. Note that irregular topologies are also used in a number of designs as well [42,43] and could emerge as equally attractive as regular topologies.

The circles represent the network switches, and the empty space in between shows the tiles where the PEs are placed. The two-dimensional mesh structure is the most favored because it is the simplest of all in terms of wiring and overall design. Three-dimensional mesh and two-dimensional torus structures are attractive however to applications, as a large number of problems map better and in a more natural form in a more highly connected topology. A particular property of the n-dimensional mesh and tori is that they do not need to have the same number of nodes in each


(a) 2-D Mesh Network


(b) 2-D Torus Network

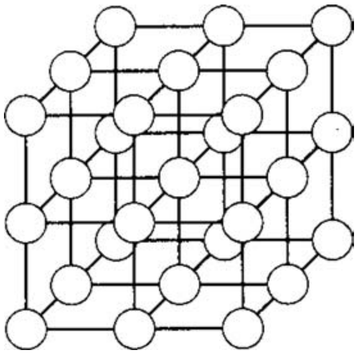

(c) 3-D Mesh Network

FIG. 9. Network topologies.

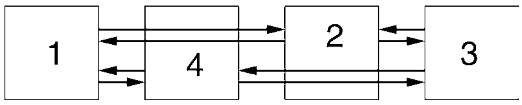


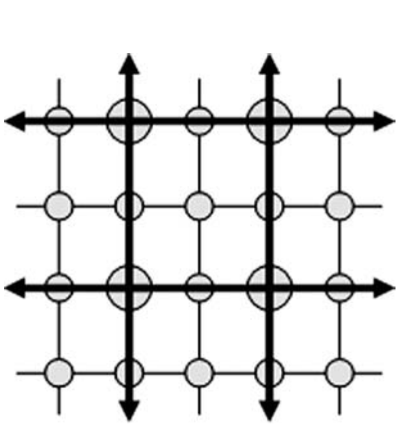
FIG. 10. Sample layout of 2D torus.

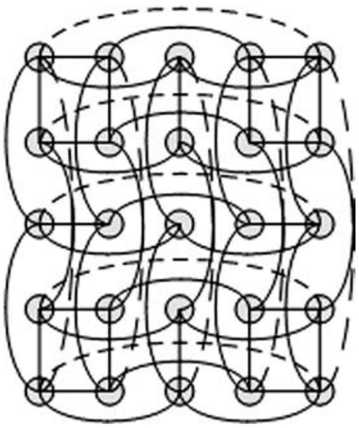
dimension; thus we can reduce the node-to-node hop count, a significant benefit in terms of latency and power consumption.

Toroidal structures allow packets leaving one 'edge' of the network to enter the opposite 'edge' of the network, significantly reducing congestion hotspots. Though logically it seems that some connections must span the entire chip, in the toroidal architecture the nodes can be placed in such a fashion as to nearly equalize the interconnect distance between logically adjacent nodes. This node placement technique is shown in Figure 10 [23,24].

Traditional mesh and toroidal structures with a large number of nodes result in multi-hop paths between two distant nodes. Hence, in an attempt to reduce the amount of hops that a flit has to travel across the network, researchers proposed the use of express channels which are implemented in similar fashion as toroidal and mesh structures, however provide connections between non-neighboring nodes, skipping over a predetermined amount of nodes. The express cube topology [78] uses bypass links between n nodes, to reduce the hop count between distant PEs. The bypass links are called express channels, allowing data traveling long paths in terms of hops, to reach its destination node in fewer hops. An example is shown in Figure 11(a), where each node represented as a larger circle (called express node), skips over 1 node at a time to communicate with the next express node in the structure. Note that there are two physical connections coming off each express node; the channel that communicates with the immediate neighboring node, and the express channel. This implementation, although effective in reducing the hop count, can result in more complicated wiring and has to be used with caution by designers. It can be applied in all types of topologies, let that be mesh, torus or tree structures. For example, Figure 11(b) shows a toroidal structure with bypass links—the figure illustrates how the wiring becomes more complicated and congested with the increase in dimensions. In the case where the express channel bypasses a single node at a time, the maximum number of hops in the data path is reduced by half [26,27,38].

Another widely used structure is the so called honeycomb structure (hierarchical mesh), where a number of PEs are clustered per router, with each router communicating with each other router in a mesh or toroidal topology, and each PE clustered around the router communicating with all other PEs directly. A two-dimensional mesh honeycomb is shown in Figure 11(c), where 6 PEs share a single router, which in turn communicates with the rest of the network. The main advantage of the hon-


(a) Express Cube Structure


(b) Toroidal Structure with express Links

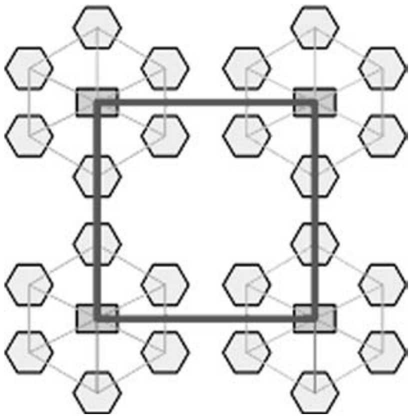

(c) Honeycomb/Hierarchical mesh structure

FIG. 11. Mesh and torus based network topologies.

eycomb structure is that it keeps a localized communication pattern amongst PEs which talk together and frequently, thus reducing the network congestion and power consumption. The main drawback of honeycomb structures is the amount of data congestion that happens at the shared switch. Knowledge of the application task flow graph however can address this problem efficiently. In addition, the amount of clustered PEs can also be determined from the mapping methods we already explored [26,27,38].

Other, less frequently used network topologies include trees and fat trees. In a tree network there is only one path between any two nodes. The taller the tree, the

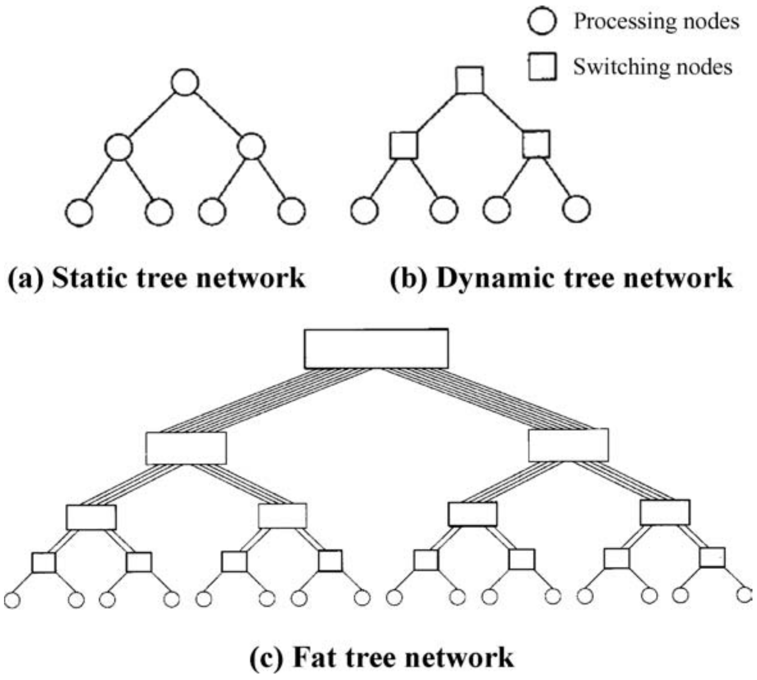


FIG. 12. Tree network topologies.

greater is the communication bottleneck at high levels of the tree. Using the NoC approach, we can design two alternative tree based topologies, such as dynamic tree networks and static tree networks. Examples of both cases are shown in Figures 12(a) and (b). Alternatively, we can have fat tree networks (Figure 12(c)) where routers communicate hierarchically with each other, with increased bandwidth for the top level routers. Trees suffer however from more complicated wiring that does not necessarily translate to the 2-dimensional structure of a chip easily [15].

In general, traditional network topologies can be adopted to be used for NoCs. On-chip dimensions however constrain the amount of dimensionality enjoyed by typical networks; therefore low-dimensioned (mesh) structures are often preferable in NoC architectures. Depending on the number of the components and the communication pattern employed in the application, the designer can select the network topology to facilitate the application. Mesh structures have been the most commonly deployed in NoC applications, however toroidal and tree structures can be used in special scenarios, like for example a neural network processor (which we will describe at the

TABLE I
COMPARISON OF SOME COMMON NETWORK TOPOLOGIES [15]

	2-d Mesh	2-d Torus	3-d Torus	Hierarchical Mesh	Fat Tree
Wiring complexity	Low	Medium	High	Low	High
Max wire length	Low	Medium	High	Medium	High
Routing complexity	Low	Medium	High	Low	High

end of this chapter) where the feed-forward layered architecture takes advantage of the torus network's ability to loop data around the edge nodes to keep forward data transfer and limit congestion. Overall, topology is an issue mostly related to the communication pattern of the application; however it does play an important role in network congestion and resource utilization. Table I summarizes the characteristics of some network topologies, in terms of routing complexity, wiring complexity and wire length [15,23].

For mesh structures, the wiring complexity is low as wires run straight from router to router, without any crossover wires. As we move however to toroidal structures, wiring complexity increases, as wires have to travel "around" or "over" routers, which in VLSI implementations, consumes additional wiring channels. This adds extra capacitance and increases the maximum wire length as well as the wiring complexity. The advantage gained however from these topologies is expressed in terms of network congestion—on average, data on a toroidal network travels a lower number of hops in order to reach its destination, as the diameter of the network is smaller due to the "wrap-around" edge. Consequently the designer has to evaluate the performance gain when using more advanced topologies, and whether it is worth doing so. Typically, for small networks mesh structures are often appropriate as their relative low complexity and small maximum network distance makes up for the extra congestion they might suffer.

3.2 Data Link

The sole purpose of the data link is to transmit data between PEs and routers reliably, as fast as possible, and using as low resources and power as possible. The data link consists of the physical wires that transmit data between chip components, let that be routers, PEs, ingress or egress nodes. In addition, data link components include hardware that transmit data, such as drivers, repeaters, error detection and correction components, and other resources that work in transmitting data. End to end communication is achieved via the switching links (wires); these wires connect to the output buffers of each component. When a component is about to send data, based on the network communication protocol, the connected buffer places the data on the bit lines which transmit the data to the input buffer of the other component. The input buffer, again based on the handshake protocol, is ready to accept the data within a given timing window. When the data is received, the data link is ready to accept the next data set. Provided that the timing is correctly calculated, the receiver will read the data prior to the sender placing other data on the line. Each end point uses two ports; an input and an output. Consequently, data transmission is unidirectional, eliminating the need for a shared medium and the issues that come with it [1–3,5–12].

The primary issue when communicating over such links is the handshake protocol. The handshake protocol can be implemented either synchronously, asynchronously, or using the clock and additional control/latching signals. Synchronous communication is commonly used, where data stays on each link over a period of one cycle; the output buffer places the data on the wires where the input buffer reads them at the next clock cycle. The process is synchronized with the clock, eliminating the need for additional handshake signals, making it popular among most applications. Using the clock adds also predictability to the application, and makes cycle-accurate simulators very accurate when modeling data link transmission. Given the smaller length of data link wires, the wire delay can be kept less than the clock cycle hence allowing synchronized communication to operate without any timing issues. Synchronous communication however is not necessarily the best method; depending on the application, two particular components might be able to communicate with each other at a much faster rate than the clock cycle. Restricting their communication to the clock frequency implies a reduction in their utilization. In order to improve performance and eliminate dependence completely on the clock, we can either use an asynchronous handshake between components, or additional bit lines, which implement data transfer control signals [10–15,67].

Asynchronous communication is done via handshake signals; when a component wishes to send data to the network, the network router associated with the component is notified via a predetermine signal. If the router is ready to receive the data, it sends a signal to the component that it is ready to receive the data. The component then places the data on the bit lines, and the router reads the data within some timing window, determined by the network protocol. Upon reception, the router sends an acknowledgment signal to the component thus ensuring successful transmission. This method allows both the router and the component to work completely independently of each other until data is ready to be sent. However, this method has to be universal between routers and components, with the control signals possibly requiring a noticeable amount of area, as well as increasing the actual transmission delay.

Asynchronous handshake signals do not need a clock to operate; however, most pre-designed components (IP cores) are synchronous with a clock. Using an asynchronous communication model, would require some sort of interfacing between the component and the network interface hardware. In addition, asynchronous operation requires some finite amount of time until data is actually transmitted. As a result, a hybrid method can also be implemented. A hybrid communication protocol can use two additional signals per port, plus the clock. The signals are called *ready* and *valid*. *Ready* indicates that a component is ready to receive data; *valid* indicates that the sender transmits valid data. Using the sender's *valid* signal, and the receiver's *ready* signal, and the positive (negative) edge of the clock, data is transmitted between the sender and the receiver when both signals (*ready* and *valid*) are high, and the clock signal rises (falls). The sender places the data on the bit lines and raises its valid signal; when the receiver is ready to accept data, it raises its ready signal, at which point it reads the data.

An important issue is the amount of wires placed between routers and components. If we assume a regular, tiled structure, with the router laid out next to the PE, wiring area is restricted to the width of the router [23]. Designing data link interconnects using dense wires implies dealing with noise sources due to crosstalk, process variation, and environmental variation. As the technology scales down, supply voltage is reduced, lowering the noise margins. In addition, low voltage swing interconnects are more sensitive to noise. Crosstalk noise on a wire is induced by the switching activities on a neighboring wire, due to the capacitive coupling between the adjacent wires. Crosstalk noise affects the timing of signals, increasing propagation delay and clock cycle time. Consequently this leads into performance degradation, as well as functional failures. Other sources of noise are caused due to changes in characteristics of devices and wires caused by the IC manufacturing process and wear-out. Similarly, noise is caused by changes in the supply voltage, the chip temperature and local coupling caused by the specifics of the implementation. Designing fault tolerant data links is not only limited to the end-to-end error detection and correction hardware, but also done at the wire level, where a certain minimum wire pitch has to be kept; wire spacing becomes an issue. Some link designs propose the use of pairs of wires transmitting a differential signal in order to reduce the impact of noise sources, but increasing the number of wires necessary for data transmission. In dealing with interconnect noise, designers can use established methods such as wire shielding and spacing, or can utilize a cross talk prevention coding scheme. Overall the designer needs to design for a crosstalk aware interconnect, using a reliability scheme that will send and receive data at the lowest possible power consumption level [73,74].

As a result, the designer has to be aware of the chip floor plan, and plan accordingly for the amount of wires placed between routers. The number of wires is dependent on both the flit size as well as overhead due to error protection or crosstalk encoding that can be added to a link. For example, a crosstalk resistant design is desirable, as it allows for a higher maximum operating frequency, but it requires either shielding wires, increased wire spacing, or other crosstalk addressing methods. The

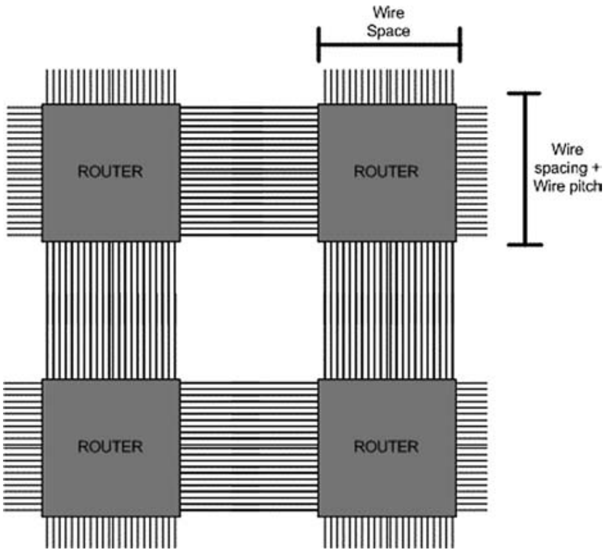


FIG. 13. Relationship between router width and number of wires.

size of the router and the number of wiring channels available are interrelated, in that a large router will have more wiring channels available along each side, allowing for an increased flit size, and hence more bits transmitted per cycle. At the same time, however, an increased flit size requires wider buffers and a larger transceiver assembly, increasing the router area further. As such, an early approximation of flit size and router area is important to further analyze the NoC interconnect. The designer can choose the remaining parameters thereafter. Figure 13 shows how the routers and wiring relate to each other [23].

3.3 Crossbar

Crossbars have an N number of inputs and an M number of outputs (where M can be equal to N), and connect inputs to outputs based on the routing decision unit. A crossbar example is shown in Figure 14. The inputs and the outputs are connected via a tristate buffer—the control signal is generated from the routing control hardware. Typically, NoC crossbars are $N \times N$ (with 4 ports for input/output to the network and a port for the PE, meaning a 5×5 crossbar as shown in the example in Figure 14), but it all depends on the number of ports per router—routers which communicate hierarchically might have more output ports than input ports, or vice versa.

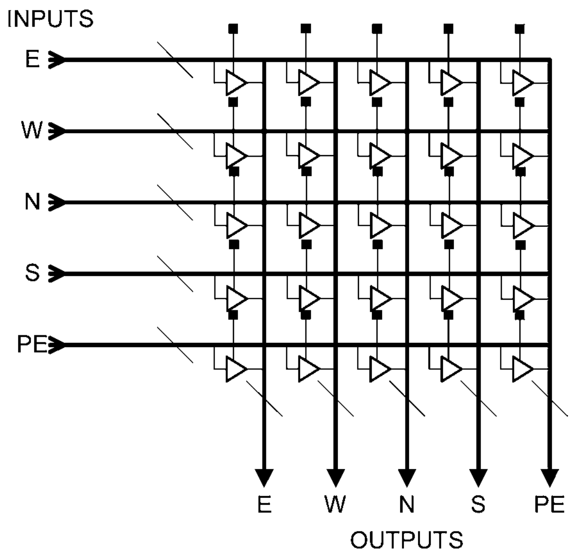


FIG. 14. A typical NoC crossbar. The tri-state buffers are controlled by the routing hardware, and connect a given input port to the desired output port.

At each connection point between a row and a column, the tristate buffer acts as the switching cell; the switching cell either provides a connection to the row and a column or not, depending on the switching signal. Switching cells can be designed using three main approaches; pass transistors (wired-OR), multiplexer based and decoder-based using NAND-inverter gates or tree of OR gates [79,80]. An example pass-transistor based implementation of the cell is shown in Figure 15.

Depending on the size of the crossbar, the implementation can differ. For small crossbars where the number of input bits is less than 16, a pass transistor crossbar is

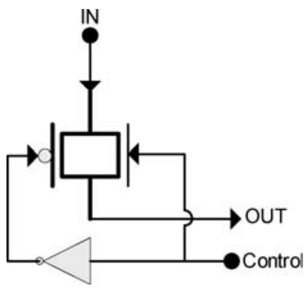


FIG. 15. An example crossbar switching cell.

more desirable due to complexity and area reasons. As the number of inputs grows larger however, multiplexer and decoder based crossbars tend to perform better, because the gain in speed exceeds the area overhead. Increase in the inputs results in an increase in the length of the crossbar wires, hence an increase in the capacitance and consequently the delay. Multiplexer and decoder based crossbars avoid this problem therefore they are faster as the number of inputs increases, but at the cost of extra hardware. The primary design parameter therefore for crossbar selection is the number of inputs and outputs [79].

As such, the delay through a crossbar increases significantly with the number of ports connected, placing a limit on the connectivity of the network. As each bit transmitted through the crossbar is independent, however, the number of bits transmitted per cycle has little effect on frequency. A technique that helps increasing the amount of inputs and outputs is called crossbar segmentation, where tristate buffers are inserted through each crossbar line, limiting the capacitance seen by a given driver; however this requires additional hardware, and while it increases the number of ports supported, it results in additional crossbar area [63]. Additionally, crossbar switching is a power hungry process, as bit lines swing signals at a very high frequency. As a result, designing for low power is a must when dealing with crossbars. We will investigate various power reduction techniques later in this chapter. Like all components in a NoC, crossbars are constrained in terms of space and layout, so an additional design parameter that has to be explored a priori to designing a crossbar, is the floor mapping of the crossbar as well.

3.4 Virtual Channels

The concept of virtual channels (VCs) has been introduced in traditional networks, with the purpose of improving the QoS of the network, and to provide escape routes for deadlock scenarios. Virtual channels are a means by which numerous types of communication can be prioritized for access to a resource. In general, augmenting an input buffer to support virtual channels allows for a more complicated access to the crossbar. Augmenting an output buffer, on the other hand, allows for a more complicated access pattern to the output link. In both cases, the more intelligent access patterns can limit head-of-line blocking, give priority of certain types of traffic over other types of traffic, and eliminate the possibility of deadlock by preventing resource-reservation chains. Virtual channels are "formed" by storing incoming data into n buffers, one for each virtual channel (rather than the single buffer found in a base system). Each buffer's output is then multiplexed to the output resource; the multiplexer select signal is what controls operation. As such, the selection of which data to send next is critical in ensuring QoS and efficient operation. The concept is illustrated in Figure 16, which shows the use of VCs at the output buffering of a

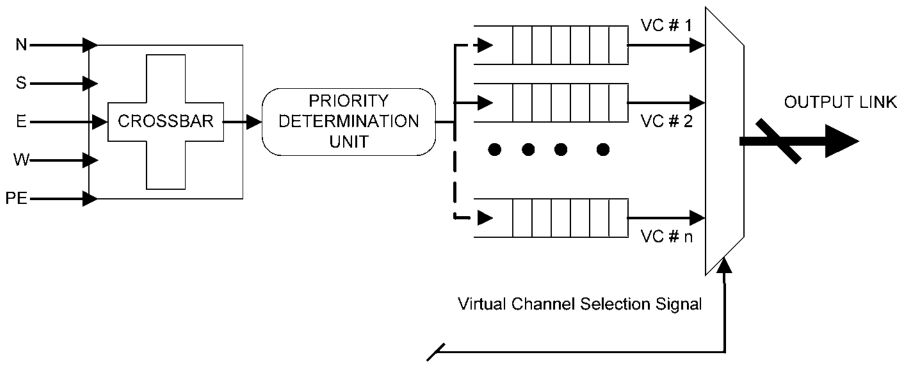


FIG. 16. Virtual channel concept for NoC.

system. Buffers (and hence virtual channels) can be placed either at the input or the output port [2,7–10,16–24].

The placement of incoming data into a particular VC's buffer is done using a number of systems, the most common of which is a weighted round robin system. In such a system, channels are prioritized with larger weight values containing higher priority data than channels with lower weight values. This implies that higher priority data will receive preferential access to the contested resource, reducing latencies for the high priority data at the expense of the low priority data.

This unfair access to resources requires the designer to evaluate the priority of each packet, ensuring that critical packets are always routed, with packets transmitting control data often being the most critical. Virtual channels can also be used to eliminate deadlocks by eliminating resource contention chains. This can be implemented in a number of ways, the simplest of which in a torus network is to place a 'dateline' in the design, which, when a packet crosses it, the packet moves from one group of VCs to another. In such a situation, a chain of packets circling the torus will exist simultaneously in both groups of VCs, but in no case can the chain ever attempt to gain access to a buffer already held by one of the packets in the chain. Further information on the use of VCs to eliminate deadlock is available in traditional networking resources, and as such, we omit it from our discussion.

While having a large number of virtual channels reduces blocking, increases the number of levels of QoS that can be provided, and reduces latency for high priority data, there are drawbacks to their use. Obviously, the inclusion of a multiplexor into the flow of data increases the delay of the device and increases the area required, but only by a small amount [17–20].

The major drawback of virtual channels for NoC is the amount of buffer space that they consume. As each channel requires its own buffer; we must split the limited resources of the NoC among each virtual channel. As such, if a NoC that has 16 packets of output buffer space available is divided into four virtual channels (of equal size), each VC can only contain 4 packets of data. If a 5th packet were to be granted access to the crossbar and attempt to enter the already full VC, a number of situations could occur: the incoming packet could be lost (a bad prospect due to the overhead required in retransmission), one of the existing packets could be dropped (also a bad prospect), or the incoming packet could be placed into an improper channel resulting in possible deadlock and violated terms of QoS. As all these options are undesirable, it is common to restrict access to any destination in which one of the VCs is full. This can lead to increased contention upstream, and result in poor performance [67].

Consequently, the designer faces another tradeoff challenge; the number of virtual channels vs. the buffering capacity per channel, both constrained by the total buffering space that we can allocate for a given resource. Recall that buffers consume large amounts of both dynamic and static energy, and designing with virtual channels—although a great theoretical concept—becomes in practice more difficult to implement in NoC architectures than in traditional networks [24].

4. Application Mapping Issues

Once the specifications for the application have been created, the application can be expressed as a task graph. Each node in the task graph represents a task that must be mapped to a physical PE and each edge in the graph represents the communication flow between the tasks. First, the architect needs to select the set of physical PEs that need to be integrated to implement the different tasks. This is followed by the mapping stage which determines how the tasks are assigned to the physical PEs [33–39]. In this section, we illustrate the application mapping using a LDPC decoder, mapped on NoC [25]. In this application, the task graph is a bi-partite graph that includes nodes implementing two different tasks (the bit and check node functions) as shown in Figure 17 (left). The target design contains two different physical PEs, called bit PE and check PE to implement these two tasks. The mapping phase involves assigning the nodes in the task graph to the actual physical PEs on the chip. Since there can be more nodes in the bi-partite graph than the number of physical PEs, multiple nodes in a bi-partite graph are assigned to a single physical PE. Such a many to one mapping is referred to as *hardware virtualization*, as multiple logical tasks share the same physical node.

Virtualization (clustering) is the process of mapping more than one application component, onto a single physical PE. We will refer to the PE as the physical com-

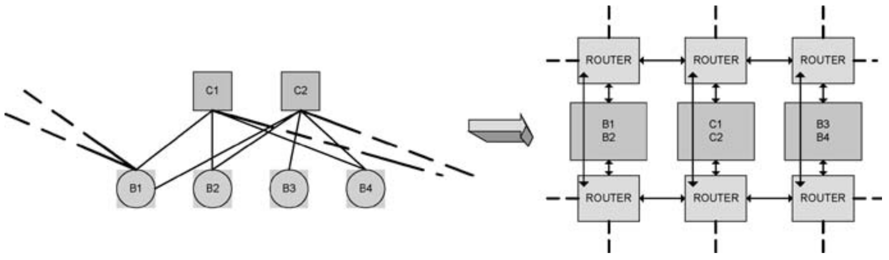


FIG. 17. Hardware virtualization.

ponent, and the mapped units as the virtual components. The concept is similar to the virtual memory concept; however the reader should see it as the concept of mapping a large number of virtual components onto a small number of physical components, and the procedure in doing so. In order to understand this concept, let us consider a part of an LDPC bipartite graph, shown in Figure 17 (left) where a set of virtual units (check nodes and bit nodes—B and C) communicate with each other in the manner shown. If we group the virtual units that perform the same task (check or bit) for instance on the same physical unit, then we can place 6 virtual units using 3 physical units, as shown in Figure 17 (right). This is defined as a virtualization of degree of 2, where there are two virtual units per physical unit. In order however to implement this, we need to modify the physical PE hardware in such a way as to give the PE the ability to identify what input has been received, and to which of the virtual units is addressed to. We can do that using the virtual addressing concept. In this concept, virtual components are addressed using an index number, ranging from zero to the number of virtual components that have to be placed. This address is called the virtual address. When each PE is mapped on the chip, each physical component is addressed using an index number ranging from zero to the number of physical components on the chip. The two index numbers now, the virtual and physical, constitute the address which traverses the network. Routers only look at the physical address; PEs however look at the virtual address to identify which virtual unit the data is addressed to. This scheme is efficient in allowing hardware resource sharing, where large hardware is shared among a number of virtual components. This allows for a non-linear area expansion when increasing the number of virtual components placed on a chip, as for instance, when mapping 2 components on a PE, the PE area increases by a factor much less than 2.

The degree of virtualization is an important tradeoff concept that the designer needs to address. Adding more virtual components in a physical unit, requires additional address decoding hardware with implied increase in both latency (and in many cases pipeline stages) and power consumption. In addition, virtual components sometimes require their own memory space to store sensitive data values necessary to each individual component. Depending on the size of these memories, it might be prohibitive in terms of memory area to map more than one virtual component per physical component. Increase in virtualization is a powerful tool to reduce the communication volume, power consumption and of course increase the amount of parallelism; however it often results in increased power consumption at the PE level and increase in the overall chip area, drawbacks which can outweigh the benefits of virtualization.

Mapping of application components is a critical step in determining the performance of the application [33–39]. In particular, mapping affects issues such as network bandwidth and effective application throughput, network congestion, resource utilization, QoS and power consumption. Intelligent mapping recognizes overlapping computations and arranges the components in such a way to increase throughput, increase parallelism and reduce the computation latency [33,34]. For example, the logical tasks that need to communicate often should be mapped to either the same physical PE or to another PE that is in close proximity in the chip. Various mapping algorithms based on genetic algorithms, statistical approaches and graph algorithms such as Dijkstra's and Prim's algorithms have been proposed [33–39].

5. Power Consumption

Power consumed in the NoC (as in any CMOS design) can be classified as dynamic and static (leakage) power. Dynamic power is consumed when there is switching activity; that is, when the signal changes from zero to one. In contrast, leakage power is consumed independent of any activity. Given the amount of routing components and the data transferred between them, it is obvious that NoC based designs can dissipate significant power. In particular, some researchers argue that approximately 40% of the power consumption on a SoC is consumed by the interconnection network [63]. Consequently, reducing the power consumption of the interconnection network has drawn much attention [58–64].

As data is transferred between routers and components, it is buffered and multiplexed in the router buffers and the PE buffers. Consequently, the transistors in those components suffer a very large switching rate, with most of them changing state nearly continuously. Reducing the amount of switching activity is therefore the primary target for low power designs, but at the same time, designing leakage-aware circuits is vital. Power consumption in a NoC can be modeled by the energy consumed per flit transmitted, shown in equation (1) below [63,65]:

$$\begin{aligned} E_{\text{flit}} &= (E_{\text{wrt}} + E_{\text{rd}} + E_{\text{arb}} + E_{\text{xb}} + E_{\text{lnk}}) \cdot H \\ &= (E_{\text{buf}} + E_{\text{arb}} + E_{\text{xb}} + E_{\text{lnk}}) \cdot H. \end{aligned} \tag{1}$$

The flit energy consists of the energy consumed in reading/writing from/to the buffers (E_{buf}), the energy spent in the arbiter performing the routing operation and multiplexing data using the crossbar ($E_{\text{arb}} + E_{\text{xb}}$) and in the energy dissipated in the data links (E_{lnk}). This is multiplied by the number of hops that a flit takes to reach its destination. It is obvious therefore that reducing the amount of hops effectively reduces the power consumption, and consequently the switching activity of the network. Various solutions can be implemented at the architectural and microarchitectural level to achieve such a drop in network distance traveled [35,36].

At the architectural level, a technique which reduces the switching activity of the network is the minimization of the distance between two components which communicate frequently with each other. This can be accomplished by using the reduction of the communication distances as the optimization criteria during the application mapping stage. Hardware virtualization helps in reducing the switching activity by placing similar components which communicate frequently on the same physical PE, thus keeping data with in the PE structure and not placing any data on the network. Once again though, placement of more than one virtual component on a physical component, results in increasingly larger physical components, resulting from the additional memory and hardware necessary to map the virtual components. Consequently, the designer needs maintain a fine balance between the amount of components mapped to each PE and the overhead power consumption and area [65].

Another way of reducing power consumption is by using encoding techniques to reduce the switching activity of the network. The idea is to recognize the nature of the data transferred and encode the transmissions to reduce number of lines that toggle their values. This can be done in various ways. An example, Gray encoding, is shown in Table II. In a Gray-encoded bit string, there is at most one toggle at a time between sequential bit strings [71].

TABLE II
GRAY ENCODING VS. BINARY ENCODING

Binary code		Gray code	
Sequence	#toggles	Sequence	#toggles
000	3	000	1
001	1	001	1
010	2	011	1
011	1	010	1
100	3	110	1
101	1	111	1
110	2	101	1
111	1	100	1

#bits	Binary toggles ($B_n = 2(2^n - 1)$)	Gray toggles ($G_n = 2^n$)	B_n/G_n
1	2	2	1
2	6	4	1.5
3	14	8	1.75
4	30	16	1.88
5	62	32	1.94
6	126	64	1.99
∞	–	–	2.00

The tradeoff for Gray encoding is the hardware overhead at both sides of the data link; data has to be encoded and then decoded at the arriving end, resulting in an increase in latency and area. In cases however with high switching activity, the power reduction becomes more significant than the overhead area and latency of the encoders and decoders. Gray encoding however, although efficient compared to binary, loses effectiveness when a totally random bit string sequence occurs. To handle such cases, other methods of encoding such as Bus Invert (BI) encoding have been devised. The idea is that at each cycle, a router decides whether sending the true or the compliment signal, leads to fewer toggles. The scheme needs an additional polarity signal on the bus to tell the bus receiver whether to invert the signal or not. The encoding results in overhead in area, power and delay of additional logic to encode/decode the data. It potentially reduces the maximum number of toggling bits from n to $n/2$ and under uniform random signal conditions (non-correlated data sequence), the toggle reduction has an upper bound of 25%. The scheme can also be applied in 'segments' within the transmitted data, where groups of bits can have their own polarity signal. When data is data is transmitted in sequential bit strings, a proposed encoding, called T0, uses address incrementer circuitry to the receiver, and adds an INC signal line to address bus. Instead of transmitting an entire word across the network, we only the INC line without sending the second data word if the second is consecutive to the first one. This is very useful when data transmitted are binary addresses, and reduces address bus transitions by 36% over binary encoding. It also outperforms Gray encoding when the probability that consecutive data words are transmitted exceeds 50% [71].

Similar to the various encoding schemes discussed here, the application itself can be used as a source of power reduction mechanisms. In cases for instance where the data patterns transmitted through the network are repeated frequently, we can encode frequently transmitted data using fewer bits, and transmitting only the necessary bits while maintaining the old values on the line. This encoding technique can potentially reduce the power consumption by a large fraction, depending on the application. The main tradeoff that designers need to explore is whether the area of the encoding/decoding hardware and the power consumption resulting from that hardware is beneficial in applying encoding techniques. Designing power-aware components at the circuit level, results in large potential power savings as well.

Crossbar switching activity is one of the primary sources of power consumption. While matrix crossbars are common designs, when a flit enters the input port of the crossbar and leaves the output port, all input lines and output lines toggle. Consequently, plenty of switching activity is unnecessary, as flits will have to travel only up to the output line instead of switching the whole line, as shown in Figure 18 (right). The only necessary switching happens along the dashed lines; the next is redundant. A way to eliminate this is by segmenting the cross bar using tristate buffers,

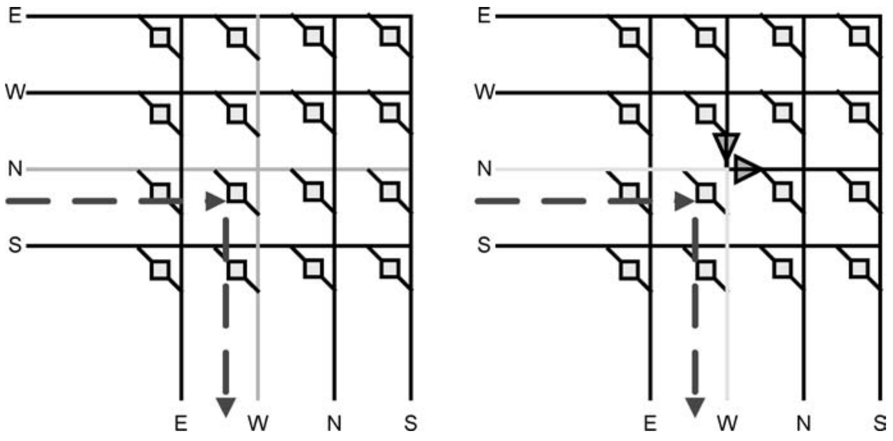


FIG. 18. Matrix vs. segmented crossbar.

and switching only the lines necessary to route a flit instead of the entire crossbar, as shown in Figure 18 (left) [63]. Similarly, cut-through crossbars can be used to reduce the switching capacitance, as proposed in [63].

Energy consumed on the buffers is also important. In wormhole routing, when a flit enters an input port, it is written to the input buffer queue, and attempts to travel through the crossbar. When the routing decision is complete and access to the crossbar is granted, the flit travels from the input buffer, through the crossbar and into the output buffer (or directly into the output link). In some situations, we can bypass the input buffer in order to reduce the power consumption. When the flit enters the router, provided the buffer is empty and the crossbar is available, the packet can be stored directly in the output buffer, limiting the energy consumed in unnecessary input buffer writes and reads. Else, it is written in the input buffer and treated normally. The idea is to eliminate the read operation, performing only the write operation when necessary. There are some improvements of this idea in [63] where buffer design was modified to eliminate the read operation by overlapping the bypass path and buffer write bit lines, using the SRAM write bit lines as the bypass path, extending beyond the buffer. A write-through buffer essentially eliminates the buffer read energy when bypassing can be done, increasing slightly the hardware cost and delay.

Several power reduction techniques have also been discussed throughout the chapter. For instance, using bypass links as shown in Section 5 reduces the amount of hops required for a packet to travel on the network. Similarly, optimal mapping of the application components results in reduction of the volume traveled across the network.

6. Reliability

The continued scaling in process technologies makes it imperative to consider reliability as a first class design constraint. Even the International Technology Roadmap for Semiconductors (ITRS) acknowledges reliability as a cross-cutting problem concerning both designers and test engineers [4]. The reliability of digital circuits is affected by a number of different sources of noise [68–71], which include the internal noises such as power supply noise, cross talk noise, and inter symbol interference, and the external noises such as electromagnetic interference, thermal noise, slot noise, and noise induced by alpha particle effects.

Traditionally, fault tolerance in networks has been achieved through complex protocols (e.g., TCP/IP), however complicated algorithms are undesirable due to the limited chip area (and hence processing power) available for the interconnection network. In particular, requiring each PE to store all packets transmitted until some sort of acknowledgement (or unacknowledgement) is received can require significant storage space due to the high bandwidth of the NoC infrastructure. For example, using either a sliding window protocol, as in a traditional network, could increase the storage hardware to the point where the storage of sent packets would consume more area than the PEs. Thus, other means of ensuring packet transmission, such as stochastic routing algorithms, have been explored by researchers. Similarly, fault tolerant routers and network interfaces have been proposed, either using error detection and correction techniques, or by taking advantage of the application operation and utilizing specific application properties in achieving fault tolerance [65,66,75,77].

At the circuit level, several error correction and detection techniques can be used to tolerate the different noise sources. Traditional error correction hardware can be placed between routers; however this hardware can be costly in terms of power consumption and area overhead. In particular, while the significant capacitances of traditional off-chip networks ensure that the cost of transmitting additional bits of data is almost always greater than the cost of encoding said data, the comparatively small capacitances found on-chip require a re-evaluation of the relative cost of transmission to the cost of encoding and decoding. Bertozzi et al. compare the energy behavior of different error correction and detection schemes [72]. Their results show that error detection methods combined with retransmission are more energy efficient than error correction methods under the same reliability constraints. Different coding methods have different capabilities to detect or correct errors induced by different noise sources. While the coding scheme adopted can be designed for the worst case noise scenario, such an approach will be inefficient in terms of both energy and performance. It must be observed that the induced noise fluctuates due to various environmental factors (such as altitude for soft error rates) and operational conditions (e.g., supply voltage variations due to changes in switching activity). Hence, it is necessary to design a system with self-embedded intelligence to provide the minimum amount of protection to meet the desired reliability levels.

One such adaptive scheme presented in [73] exploits the fact that a smaller voltage (hence, a smaller noise margin) would be sufficient for transmission in a less noisy phase of execution. Therefore, they increase (decrease) the voltage when the noise increases above (decreases below) a threshold point. The decrease in supply voltage provides energy savings while providing the required reliability needs. Another approach presented in [74] dynamically changes the coding strength based on the observed noise levels. The idea behind this strategy is to monitor the dynamic variations in noise behavior and use the least powerful (and hence the most energy efficient) error protection scheme required to maintain the error rates below a pre-set threshold.

At the architectural level, a very efficient technique is to use fault-tolerant routing algorithms. Such algorithms route data with the purpose of guaranteeing delivery of the data within some timeframe. Researchers have proposed various ways of doing so, using flooding techniques. Flooding implies every router in the data path routing data in all of its ports, therefore replicating packets. Consequently, it is expected that at least one packet will reach its destination PE. Given the design space however, flooding is not the best way of reliable communication, as flooding results in a huge amount of data flow on the network, resulting in a dramatic increase in the power consumption and network congestion per message sent. Consequently, researchers have proposed various modified flooding techniques to reduce the amount of packets routed in the network. Such algorithms include probabilistic and deterministic flooding [65,77].

Probabilistic flooding routes packets using probabilities. A router will pass the message on to all of its neighbors with a parameter p, where $0 \leqslant p \leqslant 1$. A high value of p corresponds to a high probability that the message will reach its destination, but it also produces a large number of messages in the network. A low value for p corresponds to a smaller number of messages in the network, but a smaller chance that the message will arrive at its destination. Probabilistic flooding does not guarantee arrival of the message, even with a fault free network. An improved methodology implies a deterministic algorithm, where the value of the routing probability depends on the Manhattan distance of the routing node from the destination. The probability higher for neighboring nodes closer to destination; messages will then tend to take paths that lead towards the destination. The directed flooding algorithm, as it is called [77] requires additional hardware for multiple random choices and each port 'decides' whether to forward a packet or not. A novel algorithm for fault tolerant routing uses the concept of the redundant random walk, inspired from the random walker problem from mathematics. The idea is to generate an initial finite quantity of messages—called 'walkers', and n different random walks through the network. The walkers attempt to enter nodes that are closer to their destination, similar to directed flooding. Packets always choose exactly one output port. The redundant random walk algorithm uses a finite number of messages and dramatically reduces the amount of overhead in comparison to directed and probabilistic flooding. In general though, fault tolerant routing algorithms operate on the issue of redundancy; the routing operation itself can employ fault tolerance techniques, such as traditional TCP/IP protocols.

7. Tools

Given the recent development of NoC, there are currently only a few tools developed for NoC systems. It is expected however that as researchers get more familiar with the architecture, more tools will gradually surface. Currently, researchers use standard network simulators such as NS-2 [40,41] to evaluate NoC topologies and network parameters, and standard design flow tools to create the hardware required for NoC.

Researchers at Carnegie Mellon University have created a cycle-accurate NoC simulator [81] which takes as inputs instantiated network topologies, application components and component parameters, network parameters and simulates the network returning network performance metrics such as bandwidth, throughput and latency. Stanford researchers have developed a mapping tool which evaluates network topologies, and a compiler which generates a synthesizable SystemC model of a NoC based on a library of NoC components [42,43,62]. Other researchers have proposed plenty of mapping and placement algorithms, to achieve higher performance, smaller area, lower power consumption and various other optimizations [44–57]. The Royal Institute of Technology (KTH) in Sweden has also developed an architectural NoC simulator [56], and a power estimation tool [60], both used in their research evaluation.

Currently, tools are mostly created in academia for research purposes, and are used to evaluate NoCs and at the same time provide useful feedback in terms of the simulator itself. As the field grows however, it is expected that industry will start developing NoC simulators and performance evaluation tools. It is a great research topic for academia and industry, and provides fundamental ground for the development of NoC-based systems.

8. Application Domain—Case Study

As the NoC architecture is relatively new, several issues are currently under investigation by researchers both in academia and industry. Consequently, the evolution of NoC based applications is still in the development stage; however a number of applications such as an MPEG-2 layer video processor [30], network processors [33,34], a combined video–audio processor [35] and a neural network processor [76] have already been successfully implemented on NoC architectures. NoC architectures so far seem ideal for highly parallel DSP processing, as typical DSP applications consist of multiple parallel operations, which are highly dependent on each other and need to send and receive data to each other. For these reasons, FPGAs have evolved as architectures for DSP—NoC architectures however seem more appropriate, as they are not constrained to the limited hardware capabilities of an FPGA. DSP algorithms are not however the only application types that NoC is ideal for. With technologies shrinking and the number of components on-chip increasing, many currently software-based applications will benefit from NoC architectures. Examples include complicated computer vision algorithms, biomedical applications, and vehicle and airplane electronic components [15].

The NoC concept is primarily new and most applications that have been implemented were mainly aimed at evaluating NoC architectures and network parameters, rather than improving application performance. Currently researchers are attempting to provide the groundwork for application growth; therefore their focus is on creating specific network architectures suitable for wide range of applications. These architectures should be highly parametrizable, scalable and reconfigurable. Consequently, research projects evolved, all attempting to implement network architectures for on-chip use. The *Nostrum* project [11] is a communication protocol stack for NoC, which assists the designer in selecting the optimal protocol parts to create a layered communication approach. *Nostrum* merges traditional hardware application mapping with novel NoC architectures, providing the packet switched communication platform which can be reused with multiple SoC applications. Similarly, the *PROTEO* project [29] provides to the designer an IP library of pre-designed communication blocks such as routers, links and network interface hardware that can be used to create NoC architectures. The library components can be configured using automated tools, allowing designers to only specify the parameters they need. The *SoCIN* project [13] offers a scalable network based on parametrizable routers that can be used in synthesizing NoC-based chips. The *Netchip* project is aimed at creating a NoC design flow, automating the time-intensive design process that is required in optimizing the network. Their target is to create a design flow which maps application on specific topologies, evaluating and selecting the optimal topology, and generating the selected topology automatically [42,43].

In general, research currently focuses on developing a NoC based platform all the way from the physical design to the operating system that will manipulate the chip operation. The latter in particular has been very intriguing to researchers as an operating system addressing NoC chips has to provide multiple issues similar to traditional network operating systems, but with the flexibility and approach used for single-chip operating systems [31,32]. In particular, NoC based operating systems will have to handle issues such as resource sharing and allocation, priority, QoS and bandwidth management, and in the cases where the chip is designed for a single application, particular application demands trimmed to the NoC supporting architecture. It is believed however that distributed operating systems which are addressing these issues, will provide a good basis for thought in developing NoC operating systems, as although the overall concept is new to the chip design area, the approach is based on multiple existing ideas [31].

In this section, we will illustrate how an application can be mapped and how it benefits from NoC architectures. We use a Neural Network processor [76] to explain how applications with high interconnect requirements benefit from NoC architecture, and how the network parameters are chosen in such a way to satisfy particular application demands.

8.1 Neural Network Processor Mapped on NoC Architecture

Artificial neural networks (ANN) are used in a wide range of applications, including pattern and object recognition, statistics and data manipulation, security applications, data mining, machine learning applications and digital signal processing. Due to the popularity of ANN applications, there has been significant research involving artificial neural network implementations, both in software and hardware. Software implementations have been researched and optimized heavily, yet still lack in performance. This makes software implementations not applicable to real time computation and emphasizes the requirement of a hardware implementation for real time systems [28]. This need is further stressed by an increasing demand for machine learning and artificial intelligence that exhibit both speed and portability in their implementation. In particular, security and emergency applications require mobility in ANN applications [28]. As a result, hardware neural network implementations have been proposed for quite some time with a large amount of activity in the early 1990s. However, hardware architectures faced major implementation issues in terms of resource demand and interconnection. In the most common neural network topologies, every node in layer n must forward its result to every node in layer $n+1$, resulting in extremely complicated inter-layer interconnect for any reasonably large number of neurons per layer (see Figure 19).

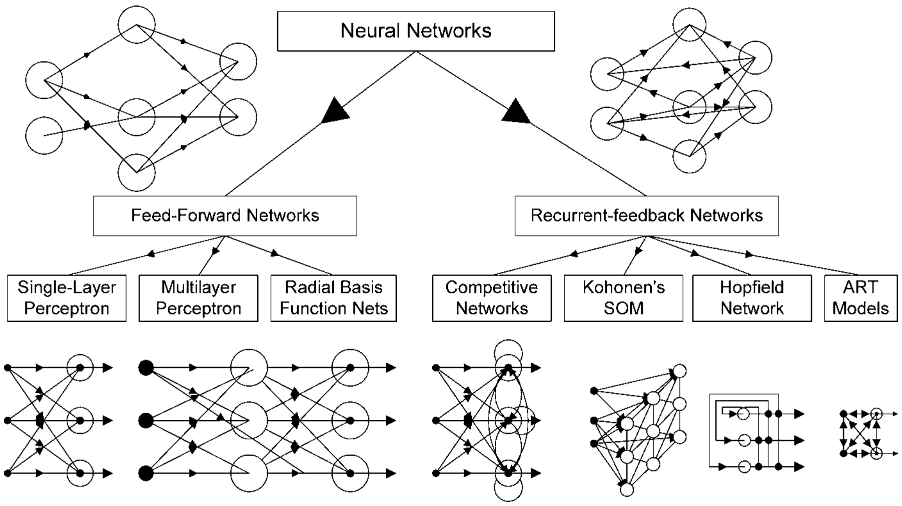


FIG. 19. Various neural network topologies [28].

Neural networks operate in two steps—the first involves training to perform an operation, the second involves adapting to perform that operation based on the training. Neurons are used in multilayer configurations, where they are trained to perform a desired operation in a similar manner to the way the human brain operates. A neuron takes a set of inputs and multiplies each input by a weight value. Weights are determined during the training process. The result of each multiplication is accumulated in the neuron until all inputs have been received. A threshold value (also determined in training) is then subtracted from the accumulated result, in order to determine whether the minimum expected sum was met. This result is then passed through an activation function to determine the final output of the neuron. The activation function used depends on the application. Typical activation functions include the sigmoid function, the hyperbolic tangent function, Gaussian and linear functions. The output is then propagated to a number of destination neurons in another layer, which perform the same general operation with the newly received set of inputs and their own weight/threshold values. The accuracy of the computation is determined by the bit widths of the inputs and the weights, as well as the accuracy of the activation function. Neurons can be connected in different configurations. The most commonly used configuration is the multilayer feed forward (FF) perceptron, although certain applications benefit from particular configurations. Radial Basis Functions networks (RBF) are also another popular approach. RBF networks have very few layers, with fewer computations, with their activation functions being the Euclidean distances between an input vector and an already determined training vector.

It is seen that neural networks demand a high bandwidth between layers, and as large data words as possible to represent accurate weights and activation functions. The NoC architecture should alleviate interconnect issues and support multiple target applications via dynamic reconfigurability, network topology independence, and network expandability. Given the layered topologies, shown as example in Figure 19, the network architecture must provide sufficient support for reconfigurability of the network to adjust to different network topologies. The most notable applications demands therefore are reconfigurability, high bandwidth between high-input layers, and large data words to represent the weights. Nevertheless, the network overhead data has to be minimal due to the fact that a data value propagated to a next layer is propagated in multiple neurons adding necessity for some sort of multicast operation. As a result, any overhead data is transmitted multiple times over the network, resulting in both extra power consumption and causing additional congestion [28].

8.2 Neural Network Architecture

The NoC architecture can be reconfigured to the application's demand to implement popular ANN topologies, such as single and multi-layer perceptron, RBF networks, competitive networks, Kohonen's self-organizing maps, and Hopfield networks. However, as almost 80% of the existing neural network applications use the multilayer perceptron configuration [28], the on-chip routing algorithm and network topology is selected in a way to optimize the feed forward multilayer perceptron. All other neural network topologies are also supported, at minimal performance cost resulting from increased network congestion. The architecture consists of four types of units/PEs—a neuron unit, an aggregator, an ingress node and an egress node. The architecture calls for two packet payloads, training and computation. Training packets carry configuration values including network topology configuration codes, weight values and activation function values, which are loaded into the units' memories. Training is implemented off-chip, using software to obtain the training weights. Computation packets carry fixed-point values between neurons to enable ANN application computations.

8.2.1 Network Mapping

As we already discussed, neuron hardware consists of MAC units, and large activation function memories, which makes the placement of a large number of physical neuron PEs on a single chip prohibitive. Given however the layered configuration of neural networks, and the fact that neurons of the same layer operate in parallel and often are configured using the same activation function [28], the designer can cluster four neurons around a single activation function unit and a NoC router. At the same time, the MAC units in each physical neuron can be shared; by providing sufficient weight memory to hold data for as many virtual neurons the designer wishes to have mapped on a single physical neuron, the designer can map neurons from separate layers on each physical neuron enhancing parallelism. Same layer virtual neurons receive data concurrently; therefore if the designer can arrange same layer virtual neurons in such manner as to allow each one to operate in parallel with every other same layer virtual neuron, the designer can accomplish a highly parallel computation. A potential mapping of more than one same layer virtual neurons on the same physical neuron with sharing of the MAC unit, would force the two (or more) same layer virtual neurons to operate sequentially, with potential reduction in the network throughput.

Physical placement of neurons also affects the performance. If the designer knows in advance the network topology, the designer should map the physical neurons in such way to reduce both the length of data propagation between layers in terms of network hops, and the network congestion. Using the multilayer perceptron approach, where each layer forwards data to all of the forward layers, it makes sense to place neurons of the nth layer immediately before neurons of the $(n+1)$th layer, and so on. In the architecture described next, the designer achieves this by placing first layer neurons on the first column, second layer neurons on the second column, and so on. Needless to say that input layer neurons should be placed next to the ingress nodes, and output layer neurons should be placed next to the egress nodes to reduce I/O data traffic.

8.2.2 Ingress and Egress Nodes

The ingress and egress nodes are responsible for communicating with the external data source and sink. These two nodes are also responsible for accepting or rejecting data from the external data source for congestion control purposes. To prevent data loss due to congestion, the combined memory space in these nodes allows storage of up to 512 32-bit inputs. The ingress nodes inject data into the network, while the egress nodes collect results from the network and pass it to the external world. These units are relatively much smaller than the other units, and have I/O pads to communicate with the external data source.

8.2.3 Neuron Unit (NU)

The neuron operation consists of a sequence of multiplications and accumulations of a set of inputs per layer, with preloaded weights. The accumulated sum is then passed to an activation function. Neuron hardware, therefore, requires a high precision multiplier as well as sufficient memory to hold one weight value for each of its input connections. It also needs a large functional unit to compute the activation function value. This would result in a prohibitively large neuron; hence it is decided to cluster partial neurons into groups of 4, each called a "neuron unit". The neuron units include the weight lookup tables as well as individual multiply-accumulate units. These four neuron units are clustered around an "aggregator", which implements the activation function, and is discussed in more detail later. Combined, four neuron units and an aggregator act as four entire neurons. This architecture reduces the overall area per neuron required on chip, and has little negative effect on performance, as the activation function is used only once per neuron computation. The clustering of four neurons together does not limit the neural network topology, as same layer neurons share the same activation function and same layer virtual neurons can be mapped on the clustered physical neurons. It becomes evident hence that the activation function unit can be shared if the designer constrains the 4 neuron units that share the activation function to receive all of their data sets within the same layer. To do so, the designer can constrain the number of neurons on each layer to a multiple of 4. In practice however, setting the weights of a neuron to zero effectively removes it from the layer, allowing the number of neurons in the computation to be constrained by hardware resource availability alone, rather than clustering by four.

Each neuron unit, shown in Figure 20 (left), consists of hardware to support computation corresponding to multiple separate layers, allowing it to be part of any layer computation. Each unit provides support for hardware virtualization, allowing the logical mapping of multiple logical neurons into one physical neuron unit as described earlier in this chapter. This requires additional input buffers for the neuron, but still only one multiply-accumulate unit is required. Virtualization is achieved by using a virtual address in the header of each packet. The neuron node identifies the virtual address of the neuron/layer that sent the data, and uses that address to gen-

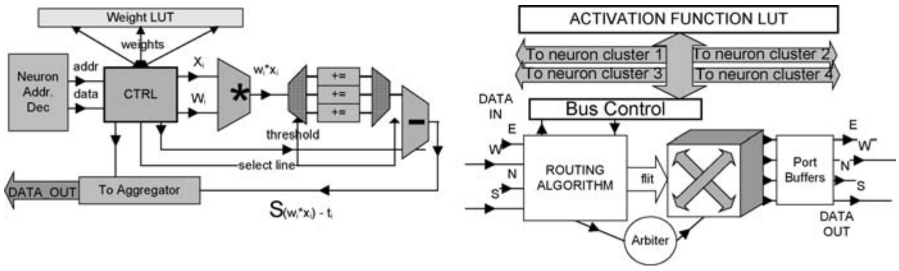


FIG. 20. Neuron (left) and aggregator (right) units.

erate the proper virtual and physical destination addresses. By virtualizing neurons, the designer can increase the number of neurons available to an application without significant increase in the hardware resources. The unit consists of a decoder which decodes the neuron address and determines which neuron in the computation the data corresponds to. A control logic block receives the decoded header and data and sends a memory request for the appropriate weight. Weights are addressed using the layer and virtual source identification number as addressing bits, information contained in the header. The control block, in cooperation with the arbiter, maintains the computation flow using initial configuration values stored during configuration mode. Control logic is simple, consisting of counters and flag registers to keep track of to which layer and which neuron a given input value belongs. This information is used to construct the outgoing data returned to the aggregator. The execution pipeline is composed of 10 stages and is shown in Figure 20 (left).

Once all inputs for a given layer have been received and the computation for the layer is completed, the arbiter directs the proper accumulated sum to the aggregator. The aggregator will perform the activation function on the incoming data, and combine the outputs of the four logical neurons into a single packet, as discussed next.

8.2.4 Aggregator (NoC Router)

The aggregator performs two operations; as an activation function unit, it computes the activation function result for all four neuron nodes attached to it, and as a routing unit, routes packets in the network. This particular case shows that a router does not have to be isolated, but part of a larger component as long as the communication flow is kept separate from the computation flow. Figure 20 (right) shows the block diagram of the unit. The unit consists of the NoC router hardware already explained in this chapter, with the modification that each routing node now is connected to 4 neuron nodes (processing elements) instead of the traditional 1 processing element per routing node which we illustrated in this chapter. This implementation takes advantage of the fact the neural network activation functions remain the same for each layer. Therefore, this allows multiple same-layer neurons to share an activation function unit. To allow for optimal reconfigurability, the activation function is implemented via lookup table, programmed using configuration packets. The activation function values are loaded into the memory on configuration, and remain unchanged during the computation unless the application makes any dynamic adjustments. The architecture supports 4 different activation functions; the hyperbolic tangent function, the piecewise linear function, the sigmoid function and the Gaussian function. These are the most popular functions used in neural network technology today [28]. Functions such as the sigmoid and the Gaussian are characterized by parameters which change for each application.

When a neuron unit finishes its computation and sends the results back to the aggregator, the aggregator performs the activation function via the appropriate lookup table. It then combines the output of the four logical neurons into a single packet, which it forwards to the appropriate aggregator (in the next logical layer) or to an egress node. This clustering of payloads into a single packet reduces overall traffic on the network, and results in higher efficiency.

8.2.5 System Overview

The system performs two types of operations; configuration and computation. Configuration involves initialization of control registers and the network structure, allocation of physical and virtual addresses in order to achieve desired network topology, and initialization of the weights and activation function values for the on-chip memories. The network is fully reconfigurable, allowing the application to load desired weights and activation function during configuration, and to change these values whenever desired. During computation operation, the network performs the algorithm desired by the application. The architecture is expandable and can be minimally modified if desired to support more layers and more neurons by paying the extra hardware penalty.

Figure 21 shows the topology of the network with 9 aggregator nodes. The ingress, egress, and aggregator nodes are all connected using packet switches. The architecture supports deadlock free X–Y routing, and uses wormhole/packet switched communication, as described in this chapter. Each packet is composed of a header and four data fields, as this allows each packet to carry data to four neurons simultaneously. The general network topology is a 2D torus, allowing packets leaving one ‘edge’ of the network to enter the opposite ‘edge’ of the network, significantly reducing congestion hotspots. If desired, the feed-forward network can be mapped in a near-systolic manner, allowing for minimal network congestion. The heaviest congestion occurs near the input nodes, as the input layer of a neural network typically sees the highest amount of data, receiving a much larger number of inputs than any other layer. Ingress and egress nodes are responsible for congestion control between operations, inserting inputs for a new computation only after a period of time sufficient to allow the network traffic to lower. Internally, the network uses congestion signals to prevent the loss of any packets, as a single packet loss can disrupt the computation heavily.

There are significant network benefits to clustering neuron nodes around aggregators, rather than simply placing neuron nodes around the network to communicate with the activation function via packets. As an example, if we assume layers of 20 neurons each, the communication between two layers requires 20 neurons in the first layer to send their accumulated sum to the activation function. Each activation

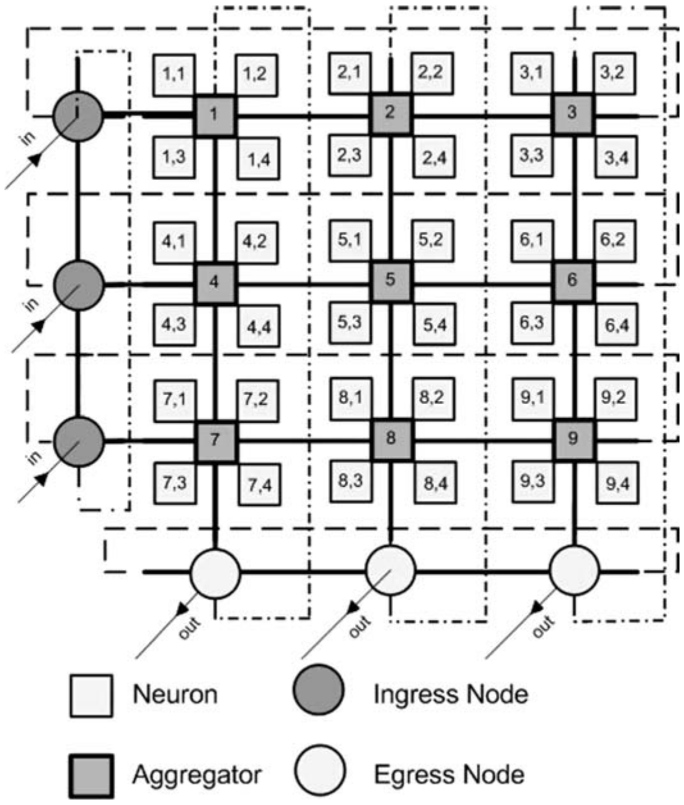


FIG. 21. System architecture.

function must then send each output data value to each of the 20 neurons in the second layer. This requires 420 packets be sent across the network. Clustering in this fashion requires each group of four neurons send only one (slightly larger) packet to every other group of four neurons. This results in only 16 packets being sent across the network, a reduction in network traffic of more than 96%, and obvious benefits in terms of network performance and power consumption.

8.2.6 Comparison to Traditional, Point-to-Point Implementations

In order to illustrate the benefits of NoC architecture over traditional ASIC implementations, we present the performance of a NoC implementation with the perfor-

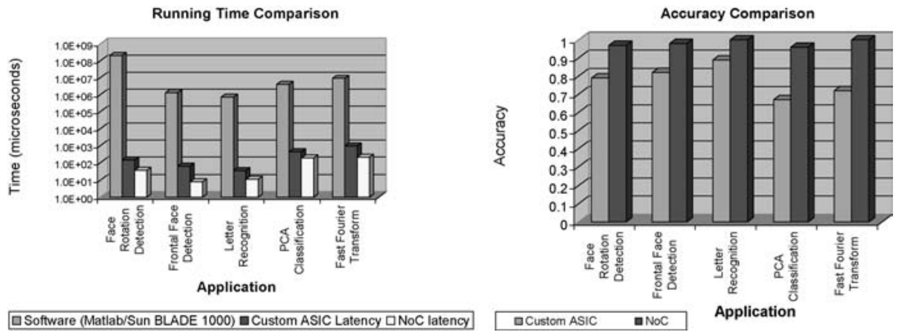


FIG. 22. Timing and accuracy comparison of NoC implementation vs. software and traditional point-to-point hardware implementations.

mance of that of a ASIC implementation using traditional point-to-point interconnects [82], using the same technology and neural network structure. We compared the two implementations using 5 neural network applications. Figure 22 shows the performance comparison in terms of network accuracy and latency, when comparing the traditional ASIC implementation and the NoC implementation (and, for purposes of comparison, a double precision software implementation). It is seen that the NoC architecture provides a much better ground for both speed and accuracy, as the regular structure, full parallelism and reconfigurability of the network allow for adaptive reconfiguration of the network to the application's demands, but most importantly, allow a large data bandwidth over layers, enabling the use of wide data words. Consequently the results in Figure 22 show the superiority of the NoC architecture over a traditional point-to-point connection architecture in terms of the accuracy, with accuracy approaching the double-precision software accuracy. The ability to share MAC units among more neurons while maintaining a parallel operation and utilizing more bits per data word due to the regular NoC implementation, aids in the increased accuracy—traditional point-to-point connection architecture on the other hand consumes much more wiring area, restricting both the multiplier precision, as well as the amount of computation resources placed on the same chip area.

9. Conclusion

This chapter presented a novel architecture for on-chip interconnects. On chip networks are expected to eventually replace traditional buses, as they offer plenty of advantages compared to existing bus architectures. NoC architecture offers a predictive wiring model, where traditionally long wires pave way for shorter, easier to model wires, with drivers and repeaters designed to match accurately the wire.

Network on chip is based on connecting system components to a backbone network instead of a traditional bus. On-chip routers are used to route data between components, and each component interfaces to the network via simple network interface hardware. Applications are mapped based on the application task flow graph, and components are placed in such a way as to optimize network performance. Data is routed using traditional network protocols, adapted to meet on-chip constraints. The whole process allows for hierarchical design flow and layered approaches in both simulation and evaluation stages.

In a traditional bus, every new unit attached adds parasitic capacitance; therefore electrical performance degrades with the bus growth. Bus timing is also difficult to model accurately in deep sub-micron processes. Bus testability is problematic and slow, and the bus arbiter delay grows with the number of master busses merging into a single bus. The arbiter is also instance-specific. Bandwidth is limited and shared by all units attached to the bus. On the other hand, NoC offers only point-to-point connects, using unidirectional wires, for any network size and topology. Wires can be in fact “pipelined” via routers, making the global communication protocol “asynchronous”—a signal can take two or three cycles to reach a destination, where as another signal can take only one cycle. NoC testability is faster and complete as it involves independent components. Routing decisions can be distributed or centralized, and routers can be reused once designed, without any effect to the size and topology of the network. The network bandwidth scales with the network size, making the network scalable and reconfigurable.

There are some cases where buses have the edge [14]. Bus advantages include minimal latency, and near zero cost for a small number of communicating blocks. Also, any bus model is directly compatible with most existing IP cores. Traditional busses enjoy simple concepts and are easier to design as well. Networks on the other hand imply additional latency in terms of the protocol operations and the network congestion. Similarly, NoC infrastructure typically consumes more area than a traditional bus. Existing IP cores and components need network interface hardware to attach to the network.

NoC is a still a relatively new concept, still at the early stages of development; therefore the field offers excellent research opportunities for both academics and professionals. There are plenty of performance trade offs that have to be researched and evaluated based on different application constraints.

Acknowledgements

This work was supported in part by NSF CAREER Award CCR 0093085 and a grant from DARPA/MARCO GSRC.

REFERENCES

[1] Benini L., De Micheli G., “Networks on chips: a new SoC paradigm”, *IEEE Computer* **35** (January 2002) 70–78.

[2] Dally W., Towles B., “Route packets, not wires: on-chip interconnection networks”, in: *Proceedings of 38th Design Automation Conference*, June 2001.

[3] Kumar S., Jantsch A., Millberg M., Berg J., Soininen J.P., Forsell M., Tiensyrj K., Hemani A., “A network on chip architecture and design methodology”, in: *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, April 2002.

[4] “International technology roadmap for semiconductors”, http://public.itrs.net/, June 2004.

[5] Hemani A., Jantsch A., Kumar S., Postula A., Öberg J., Millberg M., Lindqvist D., “Network on chip: an architecture for billion transistor era”, in: *Proceedings of NorChip 2000, Turku, Finland*, November 2000.

[6] Jantsch A., “Networks on chip”, in: *Proceedings of the Conference Radio Vetenskap och Kommunication, Stockholm*, June 2002.

[7] Bhojwani P., Mahapatra R., “Interfacing cores with on-chip packet-switched networks”, in: *Proceedings of VLSI Design 2003*, January 2003.

[8] Sgroi M., Sheets M., Mihal A., Keutzer K., Malik S., Rabaey J., Sangiovanni-Vencentelli A., “Addressing the system-on-a-chip interconnect woes through communication-based design”, in: *Proceedings of the 38th Design Automation Conference*, June 2001, pp. 667–672.

[9] Magarshack P., Paulin P., “System-on-chip beyond the nanometer wall”, in: *Proceedings of the 40th Design Automation Conference*, June 2003.

[10] Brebner G., Levi B., “Networking on chip with platform FPGAs”, in: *Proceedings of the IEEE ICFPT*, December 2003.

[11] Millberg M., Nilsson E., Thid R., Kumar S., Jantsch A., “The nostrum backbone—a communication protocol stack for networks on chip”, in: *Proceedings of the VLSI Design Conference*, January 2004.

[12] Soininen J.-P., Jantsch A., Forsell M., Pelkonen A., Kreku J., Kumar S., “Extending platform-based design to network on chip systems”, in: *Proceedings of the International Conference on VLSI Design*, January 2003.

[13] Zeferino C.A., Susin A.A., “SoCIN: a parametric and scalable network-on-chip”, in: *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design, SBCCI’03*, March 2003.

[14] Wielage P., Goossens K., “Networks on silicon: blessing or nightmare?”, in: *Proceedings of the Euromicro Symposium on Digital System Design (DSD’02)*, September 2002.

[15] Jantsch A., Tenhunen H., *Networks on Chip*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2003.

[16] Wiklund D., Liu D., “Design of a system-on-chip switched network and its design support”, in: *Proceedings of the International Conference of Communications, Circuits and Systems*, June 2002.

[17] Bolotin E., Cidon I., Ginosar R., Kolodny A., “QNoC: QoS architecture and design process for network on chip”, *Journal of Systems Architecture* **50** (2004) 105–128.

[18] Rijpkema E., Goossens K.G.W., Radulescu A., Dielissen J., van Meerbergen J., Wielage P., Waterlander E., “Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip”, in: *Proceedings of DATE 2003*, February 2003.
[19] Pande P.P., Grecu C., Ivanov A., Saleh R., “Design of a switch for network on chip application”, in: *IEEE International Symposium on Circuits and Systems, ISCAS, Bangkok, Thailand*, vol. V, 2003, pp. 217–220.
[20] Liu J., Zheng L., Tenhunen H., “A guaranteed-throughput switch for network-on-chip”, in: *Proceedings of the International Symposium on System-on-Chip 2003*, November 2003.
[21] Zeverino C., Kreutz M., Susin A., “RASoC: a router soft-core for networks-on-chip”, in: *Proceedings of DATE 2004*, February 2004.
[22] Xu J., Wolf W., “A wave-pipelined on-chip interconnect structure for networks-on-chips”, in: *Proceedings of the 11th Symposium on High-Performance Interconnects*, March 2003.
[23] Liu J., Zeng L.R., Pamunuwa D., Tenhunen H., “A global wire planning scheme for network-on-chip”, in: *Proceedings of ISCAS 2003*, vol. 4, May 2003, pp. 892–895.
[24] Pamunuwa D., Öberg J., Zheng L.R., Millberg M., Jantsch A., Tenhunen H., “Layout, performance and power trade-offs in mesh-based network-on-chip architectures”, in: *IFIP International Conference on Very Large Scale Integration (VLSI-SOC)*, December 2003.
[25] Yeo E., Nikolic B., Anantharam V., “Architectures and implementations of low-density parity check decoding algorithms”, in: *IEEE International Midwest Symposium on Circuits and Systems*, August 4–7, 2002.
[26] Nilsson E., Millberg M., Öberg J., Jantsch A., “Load distribution with the proximity congestion awareness in a network on chip”, in: *Proceedings of the Design Automation and Test Europe (DATE)*, March 2003, pp. 1126–1127.
[27] Radulescu A., Goossens K., “Communication services for networks on chip”, in: Bhattacharyya S., Deprettere E., Teich J. (Eds.), *Domain-Specific Embedded Multiprocessors*, Marcel Dekker, New York, 2003.
[28] Jain K., Mao J., Mohiuddin K.M., “Artificial neural networks: a tutorial”, *IEEE Computer* **29** (March 1996) 31–44.
[29] Sigüenza Tortosa D., Nurmi J., “Proteo: a new approach to network-on-chip”, in: *Proceedings of IASTED International Conference on Communication Systems and Networks (CSN’02), Malaga, Spain*, September 9–12, 2002.
[30] Xu J., Wolf W., Henkel J., Chankradhar S., Lv T., “A case study in networks-on-chip design for embedded video”, in: *Proceedings of DATE 2004*, February 2004.
[31] Marescaux T., Mignolet J.-Y., Bartic A., Moffat W., Verkest D., Vernalde S., Lauwereins R., “Networks on chip as hardware components of an OS for reconfigurable systems”, in: *Proceedings of FPL 2003*, August 2003.
[32] Jantsch A., “NoC’s: a new contract between hardware and software”, Invited keynote in: *Proceedings of the Euromicro Symposium on Digital System Design*, September 2003.
[33] Grunewald M., Niemann J., Porrmann M., Rückert U., “A mapping strategy for resource-efficient network processing on multiprocessor SoCs”, in: *Proceedings of DATE 2004*, February 2004.

[34] Salminen T., Soininen J.-P., “Evaluating application mapping using network simulation”, in: *International Symposium on System-on-Chip*, November 19–21, 2003.

[35] Hu J., Marculescu R., “Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures”, in: *Proceedings of Design Automation & Test in Europe (DATE’03)*, February 2003.

[36] Hu J., Marculescu R., “Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints”, in: *Proceedings of Design, Automation and Test in Europe Conference*, February 2004.

[37] Lei T., Kumar S., “A two-step genetic algorithm for mapping task graphs to a network on chip architecture”, in: *Proceedings of the Euromicro Symposium on Digital Systems Design (DSD’03)*, September 2003.

[38] Lei T., Kumar S., “Optimizing network on chip architecture size for applications”, in: *Proceedings of the 5th International Conference on ASIC, Beijing, China*, October 21–24, 2003.

[39] Murali S., De Micheli G., “Bandwidth constraint mapping of cores onto NoC architectures”, in: *Proceedings of Design, Automation and Test in Europe Conference*, February 2004.

[40] Wiklund D., “Implementation of a behavioral simulator for on-chip switched networks”, in: *Proceedings of the Swedish System-on-Chip Conference (SSoCC), Falkenberg, Sweden*, March 2002.

[41] Sun Y., Kumar S., Jantsch A., “Simulation and evaluation for a network-on-chip architecture using Ns-2”, in: *Proceeding of 20th IEEE Norchip Conference, Copenhagen*, November 11–12, 2002.

[42] Jalabert A., Murali S., Benini L., De Micheli G., “xpipesCompiler: a tool for instantiating application specific networks-on-chip”, in: *Proceedings of Design, Automation and Test in Europe Conference*, February 2004.

[43] Dall’Osso M., Biccari G., Giovannini L., Bertozzi D., Benini L., “xpipes: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs”, in: *Proceedings of International Conference on Computer Design*, October 2003, pp. 536–539.

[44] Coppola M., Curaba S., Grammatikakis M.D., Maruccia G., Papariello F., “OCCN: a network-on-chip modeling and simulation framework”, in: *Proceedings of Design, Automation and Test in Europe Conference*, February 2004.

[45] Lei T., Kumar S., “Algorithms and tools for network on chip based system design”, in: *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI’03)*, September 2003.

[46] Murgan T., Ortiz A.G., Petrov M., Glesner M., “A stochastic framework for communication architecture evaluation in networks-on-chip”, in: *Proceedings of the IEEE International Symposium on Signals, Circuits and Systems, Iasi, Romania*, July 10–11, 2003.

[47] Cota E., Kreutz M.E., Zeferino C.A., Carro L., Lubaszewski M., Susin A.A., “The impact of NoC reuse on the testing of core-based systems”, in: *Proceedings of the 21st VLSI Test Symposium (VTS’2003), USA*, IEEE Comput. Soc., Los Alamitos, CA, April 2003, pp. 128–133.

[48] Siegmund R., Muller D., “Efficient modelling and synthesis of on-chip communication protocols for network-on-chip design”, in: *International Symposium on Circuits and Systems (ISCAS)*, 2003.

[49] Madsen J., Mahadevan S., Virk K., Gonzalez M., “Network-on-chip modeling for system-level multiprocessor simulation”, in: *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, December 2003.

[50] Pestana S.G., Rijpkema E., Radulescu A., Goossens K., Gangwal O.P., “Cost-performance trade-offs in networks on chip: a simulation-based approach”, in: *Proceedings of Design, Automation and Test in Europe Conference*, February 2004.

[51] Chan J., Parameswaran S., “NoCGEN: a template based reuse methodology for networks on chip architecture”, in: *Proceedings of the 17th International Conference on VLSI Design, Mumbai, India*, January 2004.

[52] Thid R., Millberg M., Jantsch A., “Evaluating NoC communication backbones with simulation”, in: *Proceedings of the IEEE NorChip Conference*, November 2003.

[53] Ching D., Schaumont P., Verbauwhede I., “Integrated modeling and generation of a reconfigurable network-on-chip”, in: *Proceedings of the 2004 Reconfigurable Architectures Workshop (RAW 2004)*, April 2004.

[54] Vermeulen B., Dielissen J., Goossens K., Ciordas C., “Bridging communication networks-on-chip: test and verification implications”, *IEEE Communications Magazine* (December 2003).

[55] Pinto A., Carloni L., Sangiovanni-Vencentelli A., “Efficient synthesis of networks-on-chip”, in: *Proceedings of the 21st International Conference on Computer Design*, October 2003.

[56] Thid R., “A network-on-chip simulator”, Masters Thesis, Royal Institute of Technology, Sweden, August 2002.

[57] Brière M., Carrel L., Michalke T., Mieyeville F., O’Connor I., Gaffiot F., “Design and behavioral modeling tools for optical network-on-chip”, in: *Proceedings of Design, Automation and Test in Europe Conference*, February 2004.

[58] Benini L., De Micheli G., “Powering networks on chips”, in: *Proceedings of the 14th International Symposium on Systems Synthesis*, October 2001, Keynote.

[59] Simunic T., Boyd S.P., Glynn P., “Managing power consumption in networks on chips”, *IEEE Transactions on VLSI* **12** (1) (January 2004) 96–107.

[60] Banerjee N., Vellanki P., Chatha K.S., “A power and performance model for network-on-chip architectures”, in: *Proceedings of Design, Automation and Test in Europe Conference*, February 2004.

[61] Bona A., Zaccaria V., Zafalon R., “System level power modeling and simulation of high-end industrial network-on-chip”, in: *Proceedings of Design, Automation and Test in Europe Conference*, February 2004.

[62] Worm F., Lenne P., Thiran P., De Micheli G., “An adaptive low-power transmission scheme for on-chip networks”, in: *International Symposium of System Synthesis*, October 2002.

[63] Wang H., Peh L., Malik S., “Power-driven design of router microarchitectures in on-chip networks”, in: *Proceedings of the 36th International Conference on Microarchitecture*, December 2003.

[64] Raghunathan V., Srivastava M.B., Gupta R., "A survey of techniques for energy efficient on-chip communication", in: *Proceedings of the 40th Design Automation Conference*, June 2003.

[65] Marculescu R., "Networks-on-chip: the quest for on-chip fault-tolerant communication", in: *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, February 2003.

[66] Zimmer H., Jantsch A., "A fault model notation and error-control scheme for switch-to-switch buses in a network-on-chip", in: *Proceedings of CODES + ISSS Conference*, October 2003.

[67] Bolotin E., Cidon I., Ginosar R., Kolodny A., "Cost considerations in network on chip", *Integration—The VLSI Journal* (August 2003).

[68] Cuviello M., Dey S., Bai X., Zhao Y., "Fault modeling simulation for crosstalk in system-on-chip interconnects", in: *IEEE/ACM ICCAD 1999*, 1999, pp. 297–303.

[69] Shepard K.L., Narayanan V., "Noise in deep submicron digital design", in: *IEEE/ACM ICCAD 1996*, 1996, pp. 524–531.

[70] Chen H.H., Ling D.D., "Power supply noise analysis methodology for deep-submicron VLSI chip design", in: *34th DAC*, 1997, pp. 638–643.

[71] Dally W.J., Poulton J.W., *Digital Systems Engineering*, Cambridge University Press, Cambridge, UK, 1998.

[72] Bertozzi D., Benini L., de Micheli G., "Low power error resilient encoding for on-chip data buses", in: *DATE 2002*, 2002, pp. 102–109.

[73] Li L., Vijaykrishnan N., Kandemir M., Irwin M.J., "Adaptive error protection for energy efficiency", in: *Proceedings of the International Conference on Computer Aided Design (ICCAD'03)*, November 2003.

[74] Li L., Vijaykrishnan N., Kandemir M., Irwin M.J., "A crosstalk aware interconnect with variable cycle transmission", in: *DATE'04*, February 2004.

[75] Rabaey J.M., Chandrakasan A., Nikolic B., *Digital Integrated Circuits*, Prentice Hall, New York, 2003.

[76] Theocharides T., Link G.M., Vijaykrishnan N., Irwin M.J., Srikantam V., "A generic, reconfigurable neural network processor implemented as a network-on-chip", in: *Proceedings of the IEEE SOC Conference*, September 2004.

[77] Pirretti M., Link G.M., Brooks R.R., Vijaykrishnan N., Kandemir M., Irwin M.J., "Fault tolerant algorithms for network-on-chip interconnect", in: *Proceedings of the IEEE Symposium on VLSI*, February 2004.

[78] Dally W.J., "Express cube: improving the performance of k-ary n-cube interconnection networks", *IEEE Transactions on Computers* **C-40** (9) (September 1991) 1016–1023.

[79] Choi K., Adams W.S., "VLSI Implementation of a 256×256 crossbar interconnection network", in: *Proceedings of the 6th International Parallel Processing Symposium*, March 1992, pp. 289–293.

[80] Geethanjali E., Vijaykrishnan N., Irwin M.J., "An analytical power estimation model for crossbar interconnects", in: *Proceedings of the 15th Annual IEEE International ASIC/SOC Conference*, September 2002, pp. 119–123.

[81] Whelihan D., Schmit H., "The CMU network-on-chip simulator", http://www.ece.cmu.edu/~djw2/NOCsim/, June 2004.

[82] Theocharides T., Link G., Vijaykrishnan N., Irwin M.J., Wolf W., "Embedded hardware face detection", in: *Proceedings of the 17th International Conference on VLSI Design, Mumbai, India*, January 2004.

Characterizing Resource Allocation Heuristics for Heterogeneous Computing Systems

SHOUKAT ALI

*Department of Electrical and Computer Engineering,
University of Missouri–Rolla, Rolla, MO 65409-0040, USA
shoukat@umr.edu*

TRACY D. BRAUN

*3rd Dental Battalion, US Naval Dental Center, Okinawa, Japan
tdbraun@computer.org*

HOWARD JAY SIEGEL[1] AND
ANTHONY A. MACIEJEWSKI

*Colorado State University, Electrical and Computer Engineering
Department, Fort Collins, CO 80523-1373, USA
hj@colostate.edu
aam@colostate.edu*

NOAH BECK

*Advanced Micro Devices, MS 83-29, 90 Central St.,
Boxboro, MA 01719, USA
noah.beck@amd.com*

LADISLAU BÖLÖNI

*Department of Computer Engineering,
University of Central Florida, Orlando, FL 32816-2362, USA
lboloni@cpe.ucf.edu*

MUTHUCUMARU MAHESWARAN

School of Computer Science, McGill University,
Montreal, QC H3A 2A7 Canada
maheswar@cs.mcgill.ca

ALBERT I. REUTHER[2]

School of Electrical and Computer Engineering,
Purdue University, West Lafayette, IN 47907-1285, USA
reuther@ecn.purdue.edu

JAMES P. ROBERTSON

NVIDIA Corp, 12331 Riata Trace Pkwy, Austin, TX 78727, USA
jrobertson@nvidia.com

MITCHELL D. THEYS

Department of Computer Science (MC 152), University of Illinois
at Chicago, Chicago, IL 60607-7053, USA
mtheys@uic.edu

BIN YAO

Networking Platforms Research Department, Bell Laboratories,
Holmdel, New Jersey 07733, USA
byao@lucent.com

Abstract
In many distributed computing environments, collections of applications need to be processed using a set of heterogeneous computing (HC) resources to maximize some performance goal. An important research problem in these environments is how to assign resources to applications (matching) and order the execution of the applications (scheduling) so as to maximize some performance criterion without violating any constraints. This process of matching and scheduling is called mapping.

[1]Howard Jay Siegel holds a joint appointment in the Computer Science Department as well.
[2]Albert I. Reuther is currently with MIT Lincoln Laboratory, Lexington, MA.

To make meaningful comparisons among mapping heuristics, a system designer needs to understand the assumptions made by the heuristics for (1) the model used for the application and communication tasks, (2) the model used for system platforms, and (3) the attributes of the mapping heuristics. This chapter presents a three-part classification scheme (*3PCS*) for HC systems. The 3PCS is useful for researchers who want to (a) understand a mapper given in the literature, (b) describe their design of a mapper more thoroughly by using a common standard, and (c) select a mapper to match a given real-world environment.

1. Introduction

In today's interconnected world, many industry, laboratory, and military sites each use a shared set of networked computers of different types and ages. In such environments, there are large applications or collections of applications to be processed. The workload can be distributed over the set of heterogeneous computing resources to maximize some performance goal. To accomplish this, the system administrator or designer must include some method for determining which application tasks to assign to different machines in the network to exploit effectively the heterogeneity of the system platform. Such methods are applicable to the allocation of resources for many different types of computing and communication tasks in many types of environments, including parallel, distributed, cluster, grid, Internet, embedded, wireless, and reconfigurable systems.

The system designer may go to the literature to find information about techniques for assigning and scheduling (collectively referred to as "mapping") tasks in a network of heterogeneous machines. For example, in the literature, the system designer may find a technique A that is claimed to be better than another technique B. However, further investigation may show that A can only map independent (non-communicating) tasks while B can map both independent tasks and communicating subtasks within tasks. Thus, A may not be appropriate for the designer's system if the tasks therein include communicating subtasks (a workload model issue).

The system designer may find a third technique C that also handles communicating subtasks and is claimed to be better than B. It may be the case that C can incorporate information about both network contention and network link bandwidths to estimate communication times, whereas B (probably unreasonably) assumes a "fully connected" network, and is thus unable to use any contention information to estimate communication times. If a simulation study compares B and C for a system platform that is fully connected (i.e., no contention), it may turn out that B generates better mappings. However, if B and C are compared for a system platform that is not fully connected, we may find that B is not better than C, probably because B is not accounting for the network contention (a platform model issue).

Another important issue is the performance metric that the mapper tries to maximize. If both C and a fourth mapping technique D are appropriate for a particular HC environment, then different criteria may be used to determine which technique is "better." For example, if the performance metric is minimizing the average task execution time (not including the time waiting to begin execution), we might choose the technique (say D) which tries to select the fastest machine for each task. However, if the performance metric is minimization of the overall completion time of the workload, we may be more interested in selecting the technique (say C) that, for a given task, will choose an alternate machine if the fastest machine has a large queue of tasks waiting to execute (a mapping strategy issue).

Figure 1 is an illustration of the procedure used above to select a mapping heuristic based on information about the workload model, platform model, and mapping strategy. The figure uses the heuristics A, B, C, and D mentioned above.

From the above discussion, we observe that to make meaningful comparisons among mapping heuristics, a system designer needs to understand (a) the model used for workload, i.e., applications, (b) the model used for system platforms, and (c) the attributes of the mapping strategies. To accomplish this, we present a three-part classification scheme (*3PCS*) for heterogeneous computing systems (Figure 2). The 3PCS builds on the previous work done in [24,29,30,44,50,51,62,66,75,80]. All three parts of the 3PCS are needed to determine which resource allocation heuristics would be most appropriate for a given environment, and also to facilitate fair comparisons among different heuristics. We now describe some terms related to heterogeneous computing systems to set the stage for describing the 3PCS.

Performing computing and communication tasks on parallel and distributed *heterogeneous computing* (*HC*) systems involves the coordinated use of different types of machines, networks, interfaces, and other resources (e.g., [5,31,36,72]). An HC system may be used to perform a variety of different tasks that have diverse computational requirements. The resources should be allocated to the tasks in a way that

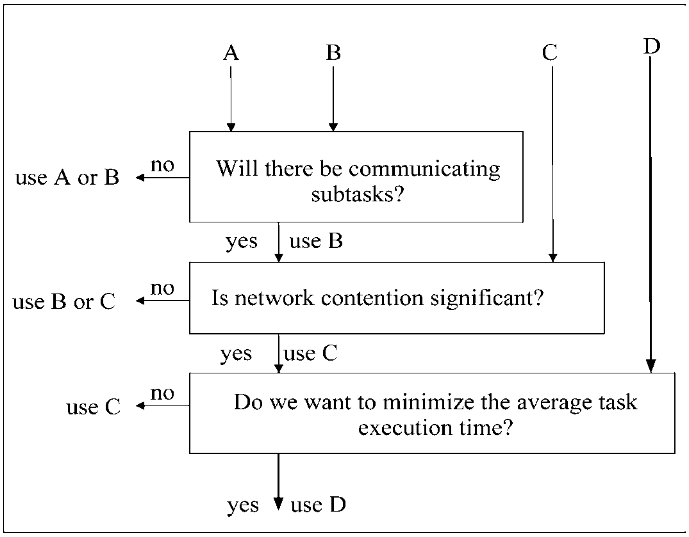

A
B
C
D
Will there be communicating subtasks?
no
use A or B
yes
use B
Is network contention significant?
no
use B or C
yes
use C
Do we want to minimize the average task execution time?
no
use C
yes
use D

FIG. 1. The choice of a mapping technique for a given system depends on the characteristics of the workload model, the characteristics of the platform model, and the mapping strategy. In this example, A, B, C, and D are four different mapping techniques discussed in the text.

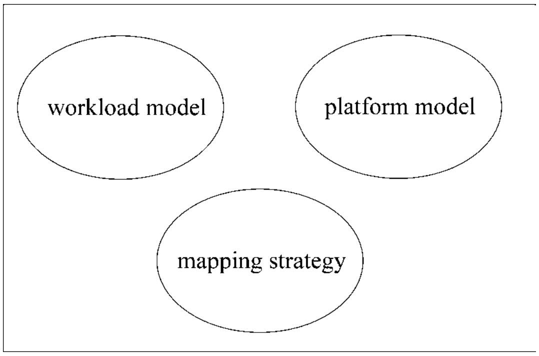


FIG. 2. The three parts of the 3PCS: workload model, platform model, and the mapping strategy.

exploits the heterogeneity to maximize some system performance measure. A cluster composed of machines of different ages and types is an example of an HC system. Alternatively, a cluster could be treated as a single machine in an HC suite. An HC system could also be part of a larger computational grid [34].

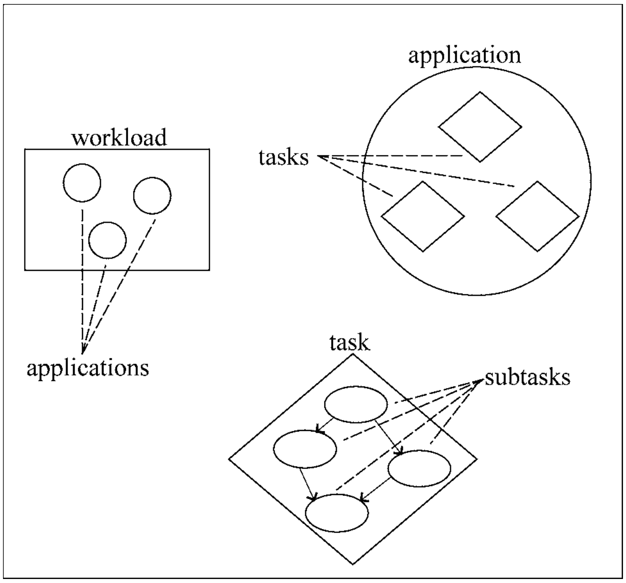


FIG. 3. The workload for a system consists of applications. Applications may be decomposed into tasks. Tasks may, in turn, be decomposed into two or more communicating subtasks. Some tasks may not have any subtasks.

The *workload* to be processed on an HC system consists of one or more *applications* (Figure 3). An application is assumed to be composed of one or more independent (i.e., non-communicating) *tasks*. It is also assumed that some tasks may be further decomposed into two or more communicating *subtasks*. The subtasks have data dependencies among them, but can be assigned to different machines for execution. If there are communicating subtasks within an application, inter-machine data transfers need to be performed when such subtasks are assigned to different machines.

A key factor in achieving the best possible performance from HC environments is the ability to effectively and efficiently *match* (assign) the applications to the machines and *schedule* (order the execution of) the applications on each machine (as well as ordering any inter-machine communications). The matching and scheduling of applications is defined as *mapping*. The mapping problem is also called resource management (or resource allocation) and a mapper is also called a resource management system.

The problem of mapping applications and communications onto multiple machines and networks in an HC environment has been shown, in general, to be NP-complete [33], requiring the development of near-optimal heuristic techniques. In recent years, numerous studies have been conducted on the mapping problem and mapping heuristics (e.g., [1,9,10,12–14,16,17,19,20,31,37,49,59,63,65,81,84,87]).

A mapper can be used in different ways. For example, we can use it to optimize a performance metric given a particular configuration of the system (i.e., particular sets of machines, networks, and protocols). Conversely, we can also use a mapper to optimize the operational *cost* of the system that is needed to reach a target value of performance. For example, a mapper could be used to select particular sets of machines, networks, and protocols that meet a certain level of performance while minimizing the dollar cost of the system (e.g., [27,64]).

The 3PCS for HC systems is useful for understanding a mapper. We propose that when one studies a mapper given in the literature, one should try to "check" the mapper in the context of the 3PCS. One should ask the questions raised in the 3PCS regarding the characteristics of the workload model, the platform model, and the mapping strategy.

The 3PCS is also useful for researchers who want to describe their mapper more thoroughly by using a common standard. When describing a mapper, one should indicate the appropriate values for all features in the three parts of the 3PCS. In this regard, the 3PCS may also help researchers see design and environment alternatives that they might not have otherwise considered during the development of new heuristics.

Additionally, the 3PCS is useful when one wants to select a mapper to match a given real-world environment. One can use the 3PCS to characterize the environment in terms of the platform model, workload model, and the mapping strategy. Then one can characterize different available mappers using the 3PCS, and select one that is most appropriate for the given environment (possibly with some modification).

Lastly, in the future, the 3PCS for HC systems could focus research towards the development of a standard set of benchmarks for evaluating resource allocation heuristics for HC environments. Classes for benchmarks could be defined based on specified workload, platform, and mapping strategy characteristics.

The work in this chapter was supported in part by the DARPA Quorum Program project *MSHN* (Management System for Heterogeneous Networks), by the DARPA *AICE* (Agile Information Control Environment) Program, and by the DARPA *BADD* (Battlefield Awareness and Data Dissemination) Program. One technical objective of the MSHN project was to design, prototype, and refine a mapping system as part of a distributed resource management system that leverages the heterogeneity of resources and workload to deliver the requested qualities of service [20,59]. One aspect of the AICE and BADD programs involved designing a scheduling system for forwarding data items prior to their use as inputs to a local application in a wide area network distributed computing environment [79,78]. The AICE, BADD, and MSHN

environments are similar in that, in some situations, not all applications or communication requests can be completed with their most preferred qualities of service by their deadlines. Thus, the goal of the mapper in these environments is to satisfy a set of application tasks or communication requests in a way that has the greatest collective perceived value. The 3PCS pertains to both the task environment in MSHN, and the communication request environment in BADD and AICE. In some cases, an application mentioned in this chapter may also refer to a communication request in the AICE and BADD context.

The rest of the chapter is organized as follows. Section 2 describes the 3PCS. Section 3 characterizes six heuristics from the literature in terms of the 3PCS. Section 4 gives an overview of some related taxonomy studies. A summary of the chapter is presented in Section 5.

2. Proposed Characterization Scheme

2.1 Overview

A “mixed-machine” HC system is composed of different machines, with possibly multiple execution models (as in the *MEMM* classification [30]). Such a system is defined to be *heterogeneous* if any features vary among machines enough to result in different execution performance among those machines. Such features could be processor type, processor speed, mode of computation, memory size, cache structure, number of processors (within parallel machines), inter-processor network (within parallel machines), etc.

The proposed 3PCS for describing mapping heuristics for mixed-machine HC systems is defined by three major components: (1) workload model, (2) platform model, and (3) mapping strategy. To properly analyze and compare mapping heuristics for current and future HC environments, information about all three parts is needed.

The 3PCS attempts to *qualitatively* define aspects of the environment that can affect mapping decisions and performance. (Doing this *quantitatively* in a thorough, rigorous, complete, and uniform manner is a long term goal of the HC field.) The 3PCS design was based on the existing mapping heuristic literature, as well as our group’s previous research and experience in the field of HC. Each part of the scheme can, of course, be investigated in more detail.

2.2 Workload Model Characterization

The first category of the 3PCS defines the model used for the workload characterization. In our context, workload characterization is limited to defining application computational and communication characteristics that may impact mapping decisions and relative mapper performance. This workload characterization defines a model for the applications to be executed on the HC system, and for the communications to be scheduled on the inter-machine network. Furthermore, the 3PCS includes application traits that may not be realistic, but do correspond to assumptions a given researcher may have made when designing and analyzing mapping heuristics in the literature. Typically, such assumptions are made to simplify the mapping problem in some way. For example, many researchers assume a given subtask must receive all of its input data from other subtasks before it can begin executing, when in reality the subtask may be able to begin with only a subset of data. As another example, some researchers may assume that the workload is divisible into arbitrary size portions that could be allocated to different machines [81]. The goal of the 3PCS is to reflect the environment assumed by the mapping heuristic, so that the workload model can capture any assumptions made (even if they are unrealistic). The defining traits of such an application model are explained below and illustrated in the organization chart given in Figure 4.

workload composition: Does the workload consist of any communicating entities, or, in other words, do the tasks within a given application have (communicating) subtasks? A workload that consists of (non-communicating) tasks only is sometimes termed as a "bag-of-tasks" or "meta-task" (e.g., Braun et al. [20], Maheswaran et al. [59]) and has a less complicated mapping problem associated with it.

Are the applications continuously executing, as in some real-time systems (e.g., [6,7])? Continuously executing applications are different from "one-shot" applications that process just one data set and then stop executing in that the former are usually part of a sensor-fed computing system. For example, the system in [6] consists of heterogeneous sets of sensors, subtasks, machines, and actuators. Each sensor produces data periodically at a certain rate, and the resulting data streams are input into subtasks. The subtasks process the data and send the output to other subtasks or to actuators (see Figure 5). Some of the items mentioned later in this characterization scheme apply only to the case where the workload contains communicating subtasks (whether continuously executing or "one-shot"). Such items will be indicated with the letter "S" written next to them.

Note that the mapping problem for a system that consists entirely of continuously executing subtasks often reduces to a problem of allocating subtasks to machines, i.e., scheduling is not involved. For example, in [6], each machine is capable of multi-tasking, executing the applications assigned to it in a round robin fashion. Similarly, a given network link is multi-tasked among all data transfers using that link.

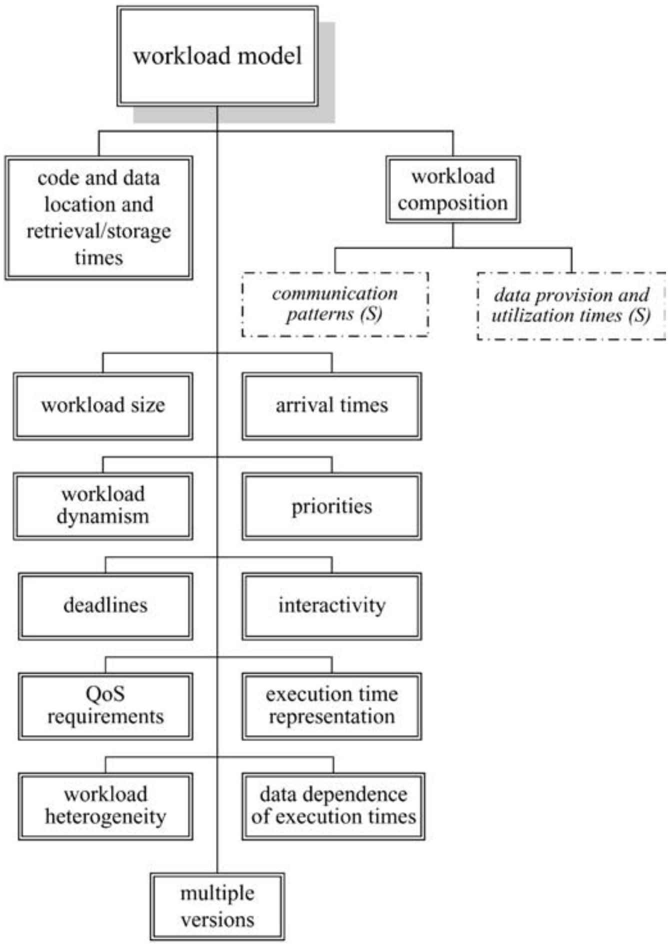

workload model
code and data location and retrieval/storage times
workload composition
communication patterns (S)
data provision and utilization times (S)
workload size
arrival times
workload dynamism
priorities
deadlines
interactivity
QoS requirements
execution time representation
workload heterogeneity
data dependence of execution times
multiple versions

FIG. 4. The different features of the workload model. The items “communication patterns” and “data provision and utilization times” are shown under “workload composition” because these items are applicable only if the workload is composed of communicating subtasks.

communication patterns (S): Does the application have any particular data communication pattern with respect to the source and destination subtasks for each data item to be transferred? Can the workload be represented as a directed acyclic graph, or a graph with loops, possibly including a set of inter-communicating co-routines? For example, assume that for a given application with n subtasks, $n - 1$ subtasks communicate only with a “master” subtask, and otherwise do not com-

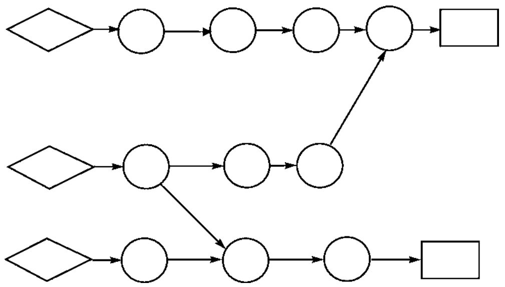


FIG. 5. The subtasks (denoted by "circles") in this system are continuously executing to process the data streams being generated by the sensors (denoted by "diamonds"). The output from subtasks is used to control the actuators (denoted by "rectangles").

municate with each other. Then, the communication times among subtasks can be reduced much more by mapping this application on a parallel machine with a "star" network topology than a machine with a hypercube network topology.

data provision and utilization times (S): Can a source subtask release data to consumer subtasks before it completes? Can a consumer subtask begin execution before receiving all of its input data? For example, the clustering non-uniform directed graph heuristic [32] assumes that a consumer subtask has to wait for the completion of the parent subtask if there is data dependency. The time at which input data needed by a subtask or output data generated by a subtask can be utilized may vary in relation to subtask start and finish times, and can help the mapper overlap the execution of inter-dependent subtasks.

code and data location and retrieval/storage times: Do tasks require data from special servers? Are data retrieval and storage times considered? Is the time required to fetch task code and initial data, as well as storing the final results of a task, considered part of the task execution time?

workload size: If a mapping heuristic was evaluated by simulation or experimentation, what size workload was used? The workload size is quantified by

(a) the number of "mappable" entities (tasks and subtasks) present in the workload (often considered relatively to the number of machines in the HC suite),
(b) the average size of "service demand," e.g., average machine utilization required by the mappable entities, average computation time, and
(c) the rate of arrival, i.e., the rate at which the mappable entities become available for assignment.

The workload size for which a given heuristic is evaluated can impact the relative performance of different heuristics for a given metric. For example, research in [59] shows that a certain class of mapping heuristics performs increasingly better than another class for larger workload sizes. Another example is given in the discussion for workload heterogeneity below.

workload dynamism: Is the complete workload to be mapped known *a priori* (static workload), or do the applications arrive in a real-time, non-deterministic manner (dynamic workload), or is it a combination of the two? Depending on this classification of the workload, we may need a “static” or a “dynamic” mapping strategy (defined in the mapping strategy characterization).

arrival times: Are arrival times for applications taken from a real system, or generated artificially for simulation studies? When taken from a real system, arrival times will be known exactly. For simulation studies, arrival times can be sampled from an anticipated probability distribution.

deadlines: Do the applications have deadlines? This property could be further refined into soft and hard deadlines, if required (e.g., [23,48]). Applications completed by a soft deadline provide the most valuable results. An application that completes after a soft deadline but before a hard deadline is still able to provide some useful data. After a hard deadline has passed, the output from the application is useless. The “worth” of the execution of a task may decrease as its completion time increases from the soft to hard deadline.

priorities: Do the applications have priorities (e.g., [48])? Environments that would require priorities include military systems and machines where time-sharing must be enforced. Priorities are generally assigned by the user (within some allowed range), but the relative weights given to each priority are usually determined by another party (e.g., a system administrator). Priorities and their relative weights are especially important if the mapping strategy is preemptive (defined in the mapping strategy characterization).

multiple versions: Do the applications have multiple versions, with different resource requirements, that could be executed (e.g., [23,70])? If yes, what are the relative “values” to the user of the different versions? Many applications have different versions. For example, an application that requires a Fast Fourier Transform might be able to perform the Fast Fourier Transform with either of two different procedures that have different precisions, different execution times, and different resource requirements.

QoS requirements: Do any applications specify any Quality of Service (QoS) requirements (other than deadlines and priorities mentioned above)? Most QoS requirements, like security, can affect mapping decisions (e.g., not mapping a particular application onto a machine of the wrong security classification).

interactivity: Are applications user-interactive (i.e., do they depend on real-time user input)? Such applications must be executed on machines for which the user has access or clearance.

workload heterogeneity: For each machine in the HC suite, how greatly and with what properties (e.g., probability distribution (e.g., [11])) do the execution times of the different tasks (or subtasks) vary for any given machine? For subtasks, the above question applies to message sizes as well.

Workload heterogeneity can affect the failure rate of a heuristic (failure rate is defined in the mapping strategy characterization). It has been shown in [4] that, for a particular system where the goal was to design static resource allocation heuristics that balance the utilization of the computation and network resources, an increase in workload heterogeneity increased the difference in failure rates among some of the heuristics being considered.

Workload heterogeneity also can impact the suitability of a performance metric for a given system. For example, the research in [4] shows that the need to measure system robustness increases as the system “complexity” increases. Based on [57,58,67,76], system complexity is a function of the system heterogeneity and the size. Figure 6 shows how the graph of robustness against slack changes in appearance as the system complexity increases. Slack is a proxy measure of robustness, and is simpler to calculate than the robustness measure introduced in [4]. The underlying systems for the first two graphs (from the left) are identical in size but different in workload heterogeneity. The third system (right-most) has the same heterogeneity as the second system but is bigger in size. It can be seen that for the first system, robustness is tightly correlated with slack. For this system, there is almost no need to calculate robustness values because the slack values can be used to approximate robustness. For the second system, however, the need to measure robustness explicitly increases. For the third system, the need is even more acute.

execution time representation: How are the estimated execution times of tasks and subtasks on each of the different machines in the HC suite determined? Most mapping techniques require an estimate of the execution time of each task and subtask on each machine (e.g., [53,85,86]).

The two choices most commonly used for making these estimates from historic or direct information are deterministic and distribution modeling. Deterministic modeling uses a fixed (or expected) value (e.g., [35]), e.g., the average of ten previous executions of a task or subtask. Alternatively, the application developer or the application user could specify the estimated execution time of the applications’ tasks and subtasks on each machine in the suite. Distribution modeling statistically processes historic knowledge to arrive at a probability distribution for task or subtask execution times. This probability distribution is then used to make mapping decisions (e.g., [12,55]).

The execution time for a given application may be determined by task profiling. Task profiling specifies the types of computations present in an application based on the code for the task (or subtask) and the data to be processed (e.g., [38,60]).

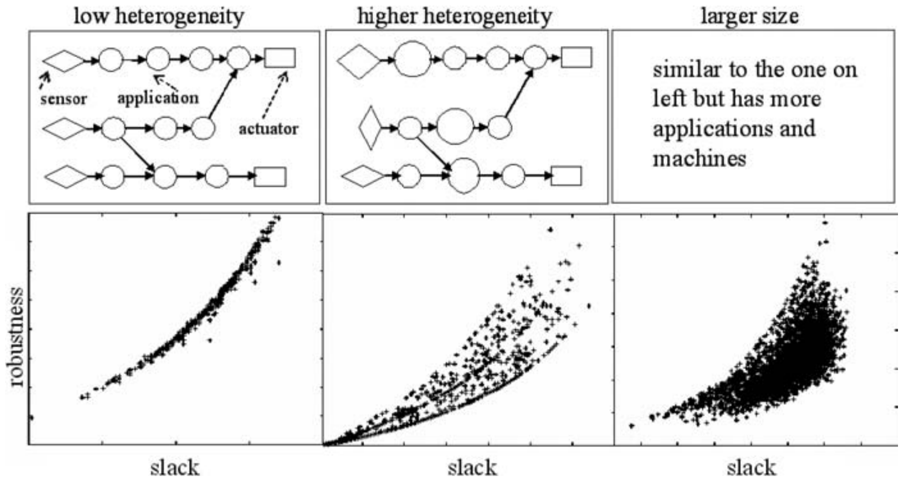


FIG. 6. The need to measure the system robustness increases as the system complexity increases. For the system with small heterogeneity, the robustness and slack are tightly coupled, thereby suggesting that robustness measurements are not needed if slack is known. As the system heterogeneity increases, the robustness and slack become less correlated, indicating that the robustness measurements can be used to distinguish between mappings that are similar in terms of the slack. As the system size increases, the correlation between the slack and the robustness decreases even further.

This information may be used by the mapping heuristic, in conjunction with analytical benchmarking (defined in the platform model characterization), to estimate task or subtask execution time.

In simulation studies, estimated execution times can be derived from probability distributions with a given mean and a given degree of heterogeneity (e.g., Ali et al. [11]).

The same issues apply to subtask communications (e.g., [79]).

data dependence of execution times: Is the execution time of a given application on a given machine independent of the data content of the application inputs (e.g., in image smoothing) or is it data dependent (e.g., in object recognition) (e.g., [73])?

2.3 Platform Model Characterization

The second category of the 3PCS defines the models used for target platforms available within HC systems. The target platform traits listed are those that may impact mapping decisions and relative mapper performance. The target platform is defined by the hardware, network properties, and software that constitute the

HC suite. Several existing heuristics make simplifying (but unrealistic) assumptions about their target platforms (e.g., assuming that an infinite number of machines are available [74]). Therefore, the 3PCS is not limited to a set of realistic target platforms. Instead, a framework for classifying the *models* used for target platforms is provided. Such a framework allows the 3PCS to reflect the environment assumed by a mapping heuristic (even if the environment is unrealistic). The defining traits of the platform model are explained below and illustrated in the organization chart given in Figure 7.

number of machines: Is the number of machines finite or infinite? Is the number of machines fixed or variable (e.g., new machines can come on-line)? Furthermore, a given heuristic may treat a finite, fixed number of machines as a parameter that can be changed from one mapping to another, e.g., when trying to find a minimum dollar-cost set of machines to meet a performance requirement (e.g., [27,64]).

system control: Does the mapping strategy control and allocate all resources in the environment (dedicated), or are external users also consuming resources (shared)?

application compatibility: Is each machine in the environment able to perform each application, or, for some applications, are there any special capabilities that are only available on certain machines? These capabilities could involve issues such as database software, I/O devices, memory space, and security.

machine heterogeneity and architecture: For each task or subtask, how greatly and with what properties (e.g., probability distribution (e.g., [11])) do the execution times vary across different machines in the HC suite? For subtasks, the above question applies to communication link speeds as well. For each machine, various architectural features that can impact performance must be considered, e.g., processor type, processor speed, external I/O bandwidth, mode of computation (e.g., shared memory, distributed shared memory, NUMA, UMA), memory size, number of processors (within parallel machines), and inter-processor network (within parallel machines). The machines in the HC suite may be evaluated on analytical benchmarks to aid in later estimating task or subtask execution times. Analytical benchmarking provides a measure of how well each available machine in the HC platform performs on each given type of computation [60]. This information may be used by the mapping heuristic, in conjunction with task profiling, to estimate task or subtask execution times. Machine heterogeneity may affect the performance of a mapping heuristic in some cases. For example, it is shown in [39] that a particular algorithm, CDA, is outperformed by another algorithm, PSP, when resource heterogeneity is increased.

code and data access and storage times: How long will it take each machine to access the code and input data it needs to execute a given task or subtask? How long will it take each machine to store any resulting output data for a given task or

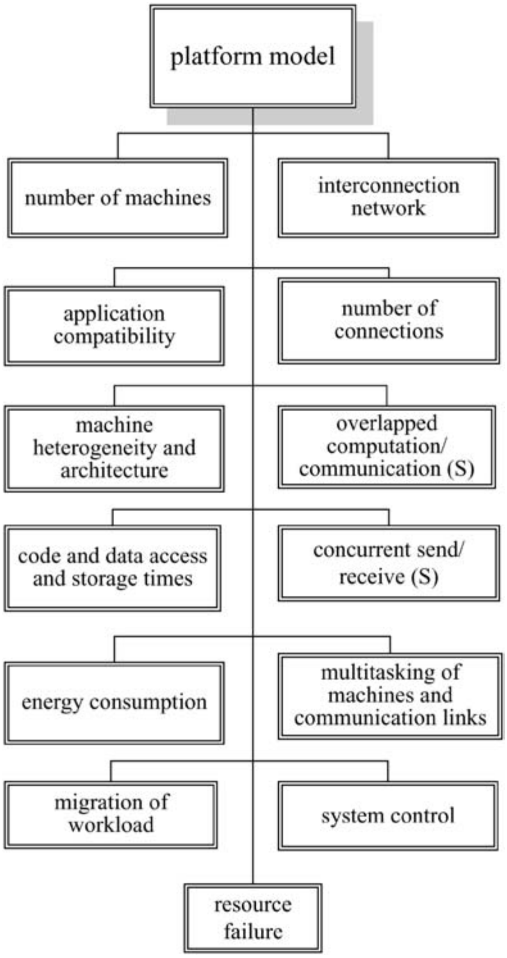

platform model
number of machines
interconnection network
application compatibility
number of connections
machine heterogeneity and architecture
overlapped computation/ communication (S)
code and data access and storage times
concurrent send/ receive (S)
energy consumption
multitasking of machines and communication links
migration of workload
system control
resource failure

FIG. 7. The different features of the platform model. The items marked with an “S” next to them are only applicable when the workload contains communicating subtasks.

subtask? For subtasks, the above questions do not apply to communication activity to/from another subtask.

interconnection network: What are the various properties of the inter-machine network? Many network characteristics can affect mapping decisions and system performance, including bandwidth, latency, switching control, and topology. Most of these network properties are also functions of the source and destination machines. Volumes of literature (e.g., [26,28,56,71]) already exist on the topic of interconnection networks, therefore, interconnection networks are not classified here.

number of connections: How many connections does each machine have to the interconnection network structure or directly to other machines?

concurrent send/receive (S): Can each machine perform concurrent sends and receives of data to other machines (assuming enough network connections)?

overlapped computation/communication (S): Can machines overlap computation and inter-machine communication?

energy consumption: How is consumption of energy in battery-based systems calculated? At what rate is energy consumed when performing computation, sending data, receiving data, or when system is idle (e.g., [41,46,69,70])? Does the system allow conservation of energy by reducing the clock speed (e.g., [46])? How many different voltage levels are allowed and how does each level impact clock rate and energy consumption (e.g., [83])? These questions are especially relevant for *ad hoc* grid computing systems, where some of the devices are mobile, battery-based, and use wireless communications (e.g., [61,69]).

multitasking of machines and communication links: Does the system allow machines and communications links to be multi-tasked (e.g., [10])? If so, how does the multitasking impact the execution time of tasks and subtasks, and the communication time of subtasks?

migration of workload: Do the machines support the migration of applications, tasks, or subtasks? Migration may be used by a "dynamic" mapping strategy (explained in the mapping strategy characterization) to re-map an application (or a task or a subtask) from a machine that has failed or is overloaded to some other suitable machine. Migration affects the communication patterns among subtasks, and may reduce the advantage of any mapping decision based on pre-migration communication patterns.

resource failure: Is the failure of machines, links, storage devices, and other resources modeled?

2.4 Mapping Strategy Characterization

The third category of the 3PCS defines the characteristics used to describe mapping strategies. Because the general HC mapping problem is NP-complete, it is assumed that the mapping strategies being classified are heuristics that attempt to produce near-optimal mappings. The different features of the mapping strategy characterization are explained below and illustrated in the organization chart given in Figure 8.

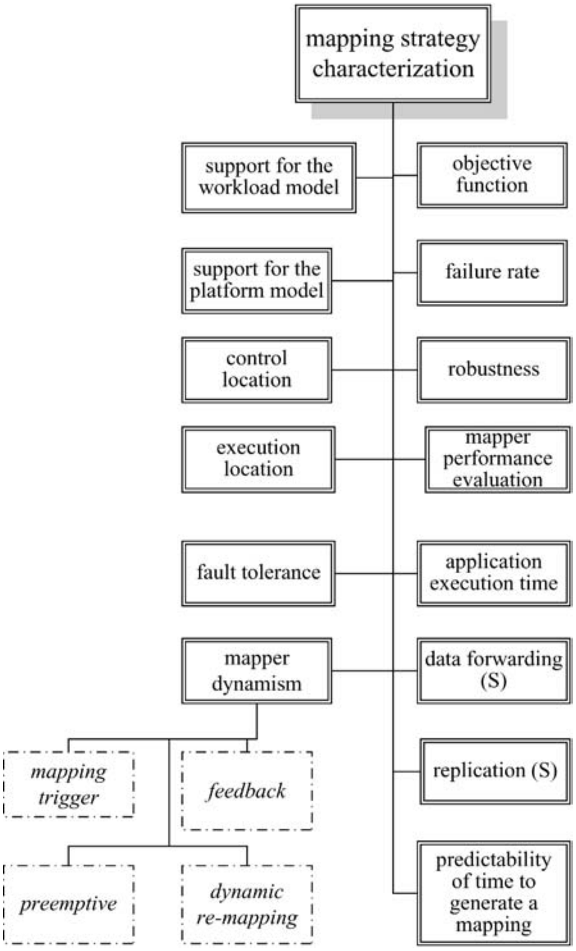

mapping strategy characterization
support for the workload model
objective function
support for the platform model
failure rate
control location
robustness
execution location
mapper performance evaluation
fault tolerance
application execution time
mapper dynamism
data forwarding (S)
mapping trigger
feedback
replication (S)
preemptive
dynamic re-mapping
predictability of time to generate a mapping

FIG. 8. The different features of the mapping strategy characterization. The items marked with an "S" next to them are only applicable when the workload contains communicating subtasks. The items in dotted boxes under "mapper dynamism" are only applicable for a dynamic or on-line mapping strategy.

support for the workload model: Can the mapping strategy use information about a given trait of the application (as modeled in the section on workload model characterization)? For example, a mapping strategy that cannot make use of the fact that a given task has multiple versions may be outperformed by one that does use this information in making mapping decisions.

support for the platform model: Can the mapping strategy use information about a given trait of the platform (as modeled in the section on platform model characterization)? For example, can the mapping strategy take advantage of any support that the platform provides for the migration of subtasks or tasks?

control location: Is the mapping strategy centralized or distributed? Distributed strategies can further be classified as cooperative or non-cooperative (independent) approaches.

execution location: Can a machine within the suite be used to execute the mapping strategy, or is an external machine required?

fault tolerance: Is fault tolerance considered by the mapping strategy? This may take several forms, such as assigning applications to machines that can perform checkpointing, or executing multiple, redundant copies of an application [1].

objective function: What quantity is the mapping strategy trying to optimize? Are there associated QoS constraints, e.g., minimizing average energy used by a mobile device while still completing the application in a given time (e.g., [69])? This varies widely among strategies, and can make some approaches inappropriate in some situations. The objective function, i.e., performance metric, can be as simple as the total execution time for the workload, or a more complex function that includes priorities, deadlines, QoS, etc. [47].

failure rate: What is the failure rate for the mapping heuristic? Before a mapping heuristic is employed in a real system, one would often evaluate its performance using simulations. In such a case, an important property of a mapping heuristic is its failure rate. A mapping heuristic failure occurs if the heuristic cannot find a mapping that allows the system to meet its QoS constraints (e.g., fails to find a resource allocation that completes a set of applications within a requested time constraint). One way of determining the failure rate is to repeat the simulation for a number of trials, where each trial uses the same probability distributions for simulation parameters (but re-samples execution times, arrival times, data sizes, etc.). The failure rate is then given by the ratio of the number of trials in which the heuristic could not find a mapping to the total number of trials. It has been shown in [10] that some heuristics may produce mappings that perform very similarly with respect to a given performance metric, but these heuristics differ very significantly in their failure rates.

robustness: For a given mapping, what is the smallest departure from the assumed conditions of system operation that will cause the objective function to degrade below some acceptable threshold? That is, what is the robustness of the mapping [9]. Parallel and distributed systems may operate in an environment where certain system performance features degrade due to unpredictable circumstances, such as sudden machine failures, higher than expected system load, or inaccuracies in the estimation of system parameters (e.g., [18,19,40,42,43,54,68]). Therefore, designing heuristics that produce robust mappings is an important design issue.

mapper performance evaluation: How is the mapper being evaluated? This issue is different from the evaluation of a mapping, which is done using the objective function. The mapper evaluation is performed before the mapper is employed in a real system, and is an attempt to find how close to optimal is the mapping found by the mapper. Because all non-trivial mapping problems are likely to be NP-hard, the mapper evaluation is as hard as finding the optimal mapping! One frequent approach is to determine an upper bound on performance. Of course the upper bound should be as tight as possible (being equal to optimal in the tightest case). Another approach is to simulate a system where a particular artificial condition makes it easy to find either the optimal performance or a tight upper bound. In yet another approach, the mapper is simply compared with another well-known mapper from literature. As a special case of the last approach, an on-line mapper may be evaluated against a static mapper with full *a priori* knowledge of the information not available to the on-line mapper. As another special case, a fault tolerant mapper may be evaluated against an otherwise identical fault intolerant mapper [1]. The relative amount of time it takes different mappers to generate mappings also may be a consideration.

application execution time: How are execution times determined by the mapper, e.g., are they estimated or produced from a probability distribution? (This was discussed in workload model and platform model characterizations.)

mapper dynamism: Is the mapping technique dynamic or static? Dynamic mapping techniques operate in real-time (as workload arrives for immediate execution), and make use of real-time information (e.g., [25,59,84,87]). Dynamic techniques require inputs from the environment, and may not have a definite end. For example, dynamic techniques may not know the entire workload to be mapped when the technique begins executing; new applications may arrive at unpredictable intervals. Similarly, new machines may be added to the suite. If a dynamic technique has feedback, the application executing on a machine may be reassigned because of the loss of the machine. Similarly, the applications waiting to be executed on a machine may be reassigned because of the currently executing applications on various machines taking significantly longer or shorter than expected.

In contrast, static mapping techniques take a fixed set of applications, a fixed set of machines, and a fixed set of application and machine attributes as inputs and generate a single, fixed mapping (e.g., [10,15,20,22]). These heuristics typically can use more time to determine a mapping because it is being done off-line, e.g., for production environments; but these heuristics must then use estimated values of some parameters such as when a machine will be available. Static mapping techniques have a well-defined beginning and end, and each resulting mapping is not modified due to changes in the HC environment or feedback. These techniques may be used to plan the execution of a set of tasks for a future time period (e.g., the production tasks to execute on the following day). Static mapping also is used in "what-if" predictive studies. For example, a system administrator might want to know the benefits of adding a new machine to the HC suite before purchasing it. By generating a static mapping (using estimated execution times for tasks and subtasks on the proposed new machines), and then deriving the estimated system performance for the set of applications, the impact of the new machine can be approximated. Static mapping also can be used to do a post-mortem analysis of dynamic mappers, to see how well they are performing. Dynamic mappers must be able to process applications as they arrive into the system, without knowledge of what applications will arrive next. When performing a post-mortem analysis, a static mapper can have the knowledge of all of the applications that had arrived over an interval of time (e.g., the previous day). Also static mappers derive mappings off-line, and thus can take much more time to determine a mapping than a dynamic mapper can. Therefore, the system performance for a static mapping should provide a good way to evaluate the system performance for a dynamic mapping of the same applications.

Some of the items mentioned later in this section apply only to dynamic mapping. They will be indicated with the letter "D" written next to them.

dynamic re-mapping (D): Can the mapping heuristic re-map an initial mapping? What event triggers the re-mapping? For example, a dynamic heuristic with feedback can re-map a previous mapping. The previous mapping could be a static or a dynamic mapping. Re-mapping is usually needed when the state of the system has changed enough so that the mapping found initially performs unacceptably low under the changed scenario. The trigger in this case could be the performance metric falling below a pre-declared threshold. Re-mapping can involve dynamically deriving a new mapping or selecting from previously stored mappings that were determined statically for different situations [52].

mapping trigger (D): What event triggers the dynamic mapping heuristic? Does it map a task as soon as it arrives (immediate mode dynamic mapping) or does it collect arriving tasks into a small batch and then perform the mapping (batch mode dynamic mapping)? Batch mode and immediate mode dynamic mapping techniques perform differently under different conditions (e.g., task arrival rate [59]). The mapping trigger could also depend on the status of the system, e.g., when the workload input queue of any machine falls below a certain threshold.

preemptive (D): What assumptions does the mapping strategy make about preemption of applications (e.g., can applications be interrupted and restarted)? Preemptive mapping strategies can interrupt applications that have already begun execution to free resources for more important applications. Applications that were interrupted may be reassigned (i.e., migrated), or may resume execution upon completion of the more important application. Preemptive techniques must be dynamic by definition. Application "importance" must be specified by some priority assignment and weighting scheme, as already discussed in the section on workload model characterization.

feedback (D): Does the mapping strategy incorporate real-time feedback from the platform (e.g., machine availability times) or applications (e.g., actual task execution times of completed tasks) into its decisions? If yes, how is the state estimated in a distributed system? In other words, as given in Rotithor [66], is state estimation:

(1) Centralized or decentralized? That is, which machines are responsible for collecting state information and constructing an estimate of the system state?
(2) Complete or partial? That is, during any state information exchange, how many machines are involved?
(3) Voluntary or involuntary? That is, how does a machine choose to disseminate state information?
(4) Periodic or aperiodic? That is, at what instant does a machine choose to initiate information dissemination.

data forwarding (S): Is data forwarding considered during mapping? That is, could a subtask executing on a given machine receive data from an intermediate machine sooner than from the original source (e.g., [77,82])? For example, assume that subtasks A, B, and C are mapped on machines X, Y, and Z, respectively, and that A needs to send the same data item to B and C. Further assume that this data item from subtask A on machine X has already been sent to subtask B on machine Y. For subtask C to receive the data item on machine Z, it may choose to receive the data item from machine X (the original source) or intermediate machine Y (the forwarder).

replication (S): Can a given subtask be duplicated and executed on multiple machines to reduce communication overhead? In a replication-based mapping, a subtask may be redundantly executed on more than one processor so as to eliminate the communication time [2,63]. Note that replication also can be used for fault tolerance, and as such has been covered under "fault tolerance" in the mapping strategy characterization.

predictability of time to generate a mapping: Is the time taken by the mapping strategy to generate a mapping predictable? For some heuristics, the mapping generation time can be accurately predicted. For example, in the greedy approaches in [10], the mapping heuristics perform a fixed, predetermined number of steps with a known amount of computation in each step before arriving at a mapping decision. In contrast, some heuristics are iterative in the sense that the mapping is continually refined until some stopping criterion is met, resulting in a number of steps that is not known *a priori*, or in a known number of steps with an unknown amount of work in each step (e.g., genetic algorithms [74,82]). The mapping generation times of different mapping strategies vary greatly, and are an important property during the comparison or selection of mapping techniques. For example, the choice between two mapping heuristics whose performance is comparable may be made based on the heuristics' mapping generation time [7,20].

3. Example Mapper Evaluations

This section characterizes six heuristics from the literature in terms of the 3PCS. The characterizations are presented in Tables I through IX. Each table contains two heuristics. The selection of heuristics to pair in each table was based on a desire to

TABLE I
WORKLOAD MODEL EXAMINATION FOR DFTS [1] AND THE BOTTOMS UP HEURISTIC [69]

Characteristic	DFTS	Bottoms up
workload composition	set of applications decomposable into tasks	one task with communicating subtasks
communication patterns (S)	N/A	directed acyclic graph
data provision and utilization times (S)	N/A	N/S
data retrieval/storage	N/S	N/C
workload size	on-line load specified with arrival and resource demand distributions	1024 subtasks with specified resource demand distribution
workload dynamism	dynamic	static
arrival times	simulated	N/A
deadlines	N/C	N/C
priorities	N/C	N/C
multiple versions	N/C	N/C
QoS requirements	N/C	task must be completed within a given time constraint
interactivity	N/C	N/C
workload heterogeneity	hyper-exponential distribution	Gamma distribution
execution time representation	distribution derived empirically	assumed distribution
data dependence of execution times	N/C	N/C

TABLE II
PLATFORM MODEL EXAMINATION FOR DFTS [1] AND THE BOTTOMS UP HEURISTIC [69]

Characteristic	DFTS	Bottoms up
number of machines	64	8
system control	shared	dedicated
application compatibility	no restrictions	no restrictions
machine heterogeneity and architecture	N/S	modeled
code and data access and storage times	N/A	assumed zero
interconnection network (S)	N/A	wireless
number of connections (S)	N/A	N/A
concurrent send/receives (S)	N/A	1 send/receive
overlapped computation/ communication (S)	N/A	yes
energy consumption	N/C	modeled
multitasking of machines and communication links	yes	N/C
migration of workload	N/C	N/C
resource failure	considered	N/C

show contrasting characteristics. The heuristics have been examined with respect to each of the three parts of the 3PCS, i.e., the workload model, platform model, and mapping strategy characterization. Readers should see the references for detailed descriptions of the heuristics themselves. The following notation is used in Tables I through IX. A field is marked “N/A” if that particular feature is not applicable to the heuristic being examined. A field is marked “N/S” if that particular feature is applicable to the heuristic but its value is not specified in the paper. Finally, “N/C” stands for “not considered,” and refers to a feature that has not been considered in the given paper.

4. Related Work

Taxonomies related in various degrees to this work have appeared in the literature. In this section, overviews of some related taxonomy studies are given.

Ahmad et al. [3] survey algorithms that allocate a parallel program represented by an edge-weighted directed acyclic graph (DAG) to a set of homogeneous processors, with the objective of minimizing the completion time. They propose a taxonomy of scheduling with four groups. The first group includes algorithms that schedule the DAG to a bounded number of processors directly. These algorithms are called the

TABLE III
MAPPING STRATEGY EXAMINATION FOR DFTS [1] AND THE BOTTOMS UP HEURISTIC [69]

Characteristic	DFTS	Bottoms up
support for the workload model	yes	yes
support for the platform model	yes	yes
control location	centralized	centralized
execution location	N/S	N/S
fault tolerance	yes, through task replication	N/C
objective function	mean response time	energy consumption
failure rate	N/C	N/C
robustness	N/C	N/C
mapper performance evaluation	comparison with a fault intolerant heuristic	comparison with a lower bound
dynamism	on-line	static
dynamic re-mapping (D)	triggered when number of healthy replicas falls below a threshold	N/A
mapping trigger (D)	task arrival	N/A
preemptive (D)	yes	N/A
feedback (D)	yes	N/A
data forwarding (S)	N/A	N/C
replication	yes	N/C
time to generate a mapping	predictable	predictable

bounded number of processors scheduling algorithms. The algorithms in the second group “cluster” the subtasks, where a cluster is a set of subtasks formed to reduce or eliminate communication times. Because the number of clusters can be unbounded, these algorithms are called the unbounded number of clusters scheduling algorithms. The algorithms in the third group schedule the DAG using task duplication and are called the task duplication based scheduling algorithms. The algorithms in the fourth group perform allocation and mapping on arbitrary processor network topologies. These algorithms are called the arbitrary processor network scheduling algorithms. The authors discuss the design philosophies and principles behind these algorithms classes, and then analyze and classify 21 scheduling algorithms.

A scheme for classifying static scheduling techniques used in general-purpose distributed computing systems is presented in [24]. The classification of workload and platform was outside the scope of this study. The taxonomy in [24] does combine well-defined hierarchical characteristics with more general flat characteristics to differentiate a wide range of scheduling techniques. Several examples of different scheduling techniques from the published literature are also given, with each classified by the taxonomy. In HC systems, however, scheduling is only half of the

TABLE IV
WORKLOAD MODEL EXAMINATION FOR THE OASS ALGORITHM [45] AND THE HRA MAX–MIN HEURISTIC [8]

Characteristic	OASS	HRA max–min
workload composition	communicating subtasks	communicating subtasks, continuously executing
communication patterns (S)	undirected graph	DAG
data provision and utilization times (S)	N/S	N/S
data retrieval/storage	N/S	N/S
workload size	28 subtasks, resource demands calculated with a devised procedure	40 subtasks, resource demands sampled from a Gamma distribution
workload dynamism	static	static
arrival times	N/A	N/A
deadlines	N/C	N/C
priorities	N/C	N/C
multiple versions	N/C	N/C
QoS requirements	N/C	N/C
interactivity	N/C	N/C
workload heterogeneity	N/S	Gamma distribution
execution time representation	N/S	assumed distribution
data dependence of execution times	N/C	N/C

mapping problem. The matching of tasks to machines also greatly affects execution schedules and system performance. Therefore, the 3PCS also includes categories for platform and workload models, both of which influence matching (and scheduling) decisions.

Several different taxonomies are presented in [30]. The first is the EM^3 taxonomy, which classifies all computer systems into one of four categories, based on execution mode and machine model [30]. The 3PCS proposed here assumes heterogeneous systems from either the SEMM (single execution mode, multiple machine models) or the MEMM (multiple execution modes, multiple machine models) categories. A "modestly extended" version of the taxonomy from [24] is also presented in [30]. The modified taxonomy introduces new descriptors and is applied to heterogeneous resource allocation techniques. Aside from considering different parallelism characteristics of applications, [30] did not explicitly consider workload model characterization and platform model characterization.

A taxonomy for comparing heterogeneous subtask matching methodologies is included in [44]. The taxonomy focuses on static subtask matching approaches, and classifies several specific examples of optimal and sub-optimal techniques. This is a

TABLE V
PLATFORM MODEL EXAMINATION FOR THE OASS ALGORITHM [45] AND THE HRA MAX–MIN HEURISTIC [8]

Characteristic	OASS	HRA max–min
number of machines	finite, fixed	finite, fixed
system control	N/S	shared
application compatibility	no restrictions	restricted
machine heterogeneity and architecture	N/S	modeled
code and data access and storage times	N/C	N/C
interconnection network (S)	point-to-point	non-blocking switched network
number of connections (S)	1	1
concurrent send/receives (S)	N/S	N/S
overlapped computation/ communication (S)	yes	yes
energy consumption	N/C	N/C
multitasking of machines and communication links	N/S	N/S
migration of workload	N/C	N/C
resource failure	N/C	N/C

single taxonomy, without the three distinct parts of the 3PCS. However, the "optimal-restricted" classification in [44] includes algorithms that place restrictions on the underlying program and/or multicomputer system.

Krauter et al. [50] give a taxonomy and a survey of grid resource management systems. Their taxonomy covers resource and resource manager models. For example, they include classifications like resource discovery (query based or agent based), resource dissemination (periodic or on-demand), QoS support (soft, hard, or none), scheduler organization (centralized, decentralized, or hierarchical), and rescheduling (periodic or even driven). The 3PCS includes the workload model in addition to the models discussed in [50]. Furthermore, the platform model and mapping strategy characterizations in the 3PCS include parameters not included in [50].

Kwok and Ahmad [51] provide a taxonomy for classifying various scheduling algorithms into different categories according to their assumptions and functionalities. They also propose a set of benchmarks to allow a comprehensive performance evaluation and comparison of these algorithms. With very much the same motivation as the study given in this chapter, Kwok and Ahmad [51] argue that while many scheduling heuristics proposed in literature are individually reported to be efficient, it is not clear how effective they are and how well they compare against each other, partially because these scheduling algorithms are based upon radically different assumptions.

TABLE VI
MAPPING STRATEGY EXAMINATION FOR THE OASS ALGORITHM [45] AND THE HRA MAX–MIN HEURISTIC [8]

Characteristic	OASS	HRA max–min
support for the workload model	yes	yes
support for the platform model	yes	yes
control location	centralized	centralized
execution location	N/S	N/S
fault tolerance	N/C	N/C
objective function	makespan	load balancing
failure rate	N/C	considered
robustness	N/C	N/C
mapper performance evaluation	comparison with an existing heuristic	comparison with a lower bound
dynamism	static	static
dynamic re-mapping (D)	N/A	N/A
mapping trigger (D)	N/A	N/A
preemptive (D)	N/A	N/A
feedback (D)	N/A	N/A
data forwarding (S)	N/A	N/A
replication	N/C	N/C
time to generate a mapping	predictable	predictable

Kwok and Ahmad [51] evaluate 15 scheduling algorithms, and compare them using the proposed benchmarks. They interpret the results and discuss why some algorithms perform better than the others. However, the taxonomy in [51] is based on the problem of scheduling a weighted directed acyclic graph to a set of homogeneous processors, whereas the 3PCS is for heterogeneous systems with a broader range of application types.

Noronha and Sarma [62] survey several existing intelligent planning and scheduling systems for providing a guide to the main artificial intelligence techniques. They give a taxonomy of planning and scheduling problems in an attempt to reconcile the differences in usage of the terms planning and scheduling between the AI and operations research communities. Some of the more successful planning and scheduling systems are surveyed, and their features are highlighted, e.g., deterministic versus stochastic, algorithm complexity (polynomial versus NP-hard), dynamism of the scheduler (on-line versus off-line), precedence constraints, resource constraints, objective function, scheduler type (optimizing versus feasible, i.e., satisfying the problem requirements).

Rotithor [66] presents a taxonomy of dynamic task scheduling schemes in distributed computing systems. The author argues that system state estimation and decision

TABLE VII
WORKLOAD MODEL EXAMINATION FOR THE KPB HEURISTIC [59] AND THE MIN–MIN HEURISTIC [21]

Characteristic	KPB	Min–min
workload composition	independent tasks	communicating subtasks
communication patterns (S)	N/A	DAG
data provision and utilization times (S)	N/A	consumer subtask must wait until all input data has been received
data retrieval/storage	N/C	N/C
workload size	2000 tasks, resource demands sampled from a truncated Gaussian distribution	1000 tasks and 1000 subtasks, resource demands sampled from a Gamma distribution
workload dynamism	dynamic	static
arrival times	Poisson distribution	N/S
deadlines	N/C	hard deadline on each task and subtask
priorities	N/C	tasks and subtasks classified in four priority classes
multiple versions	N/C	three versions per task
QoS requirements	N/C	N/C
interactivity	N/C	N/C
workload heterogeneity	truncated Gaussian distribution	Gamma distribution
execution time representation	assumed distribution	assumed distribution
data dependence of execution times	N/C	N/C

making are the two major components of dynamic task scheduling in a distributed computing system, and that the combinations of solutions to each individual component constitute solutions to the dynamic task scheduling problem. Based on this argument, the author presents a taxonomy of dynamic task scheduling schemes that is synthesized by treating state estimation and decision making as orthogonal problems. Solutions to estimation and decision making are analyzed and the resulting solution space of dynamic task scheduling is shown. The author shows the applicability of the proposed taxonomy by means of examples that encompass solutions proposed in the literature. The state estimation classification scheme sits under the "feedback" box in Figure 8, and is incorporated in our discussion of "feedback" in the mapping strategy characterization section.

Stankovic et al. [75] give an excellent discussion of a set of real-time scheduling results. The paper presents a simple classification scheme for the real-time scheduling theory, which it divides in two groups based on the platform model: uniprocessor and multiprocessor. For the uniprocessor scheduling, the authors further differentiate

TABLE VIII
PLATFORM MODEL EXAMINATION FOR THE KPB HEURISTIC [59] AND THE MIN–MIN HEURISTIC [21]

Characteristic	KPB	Min–min
number of machines	finite, variable	finite, fixed
system control	dedicated	dedicated
application compatibility	no restrictions	no restrictions
machine heterogeneity and architecture	modeled	modeled
code and data access and storage times	N/C	N/C
interconnection network (S)	N/A	N/S
number of connections (S)	N/A	N/S
concurrent send/receives (S)	N/A	N/S
overlapped computation/ communication (S)	N/A	N/S
energy consumption	N/C	N/C
multitasking of machines and communication links	N/C	N/C
migration of workload	N/C	N/C
resource failure	N/C	N/C

TABLE IX
MAPPING STRATEGY EXAMINATION FOR THE KPB HEURISTIC [59] AND THE MIN–MIN HEURISTIC [21]

Characteristic	KPB	Min–min
support for the workload model	yes	yes
support for the platform model	yes	yes
control location	centralized	centralized
execution location	external	N/S
fault tolerance	N/C	N/C
objective function	makespan	makespan
failure rate	N/C	N/C
robustness	N/C	N/C
mapper performance evaluation	comparison with an existing heuristic	comparison with a lower bound
dynamism	dynamic	static
dynamic re-mapping (D)	yes	N/A
mapping trigger (D)	task arrival	N/A
preemptive (D)	no	N/A
feedback (D)	yes	N/A
data forwarding (S)	N/A	N/S
replication	N/C	N/C
time to generate a mapping	predictable	predictable

between dedicated and shared resources. For the multiprocessor schedulers, the authors differentiate between static and dynamic techniques, and further examine each kind for preemptive and non-preemptive techniques. The focus of the effort in [75] is on real-time systems.

T'kindt and Billaut [80] present a survey of multi-criteria scheduling, and discuss some basic results of multi-criteria optimization literature. One of their aims is to contextualize the performance evaluation of a given multi-criteria scheduling scheme to be able to answer questions like: Are trade-offs among criteria allowed? Is it possible to associate a weight to each criterion? Is it possible to associate a particular goal value to each criterion? Does an upper bound exist for each criterion? In some of these respects, their work is similar to ours, but is intended for job shop community.

The 3PCS uses these studies as a foundation, and extends their concepts. Relevant ideas from these studies are incorporated into the unique structure of the 3PCS, allowing for more detailed classifications of HC mapping heuristics.

5. Summary

Heterogeneous computing (HC) is the coordinated use of different types of machines, networks, and interfaces to meet the requirements of widely varying application mixtures and to maximize the overall system performance or cost-effectiveness. An important research problem in HC is mapping, i.e., how to assign resources to applications, and schedule the applications on a given machine, so as to maximize some performance criterion without violating any quality of service constraints. This chapter proposes a three-part classification scheme (*3PCS*) for understanding, describing, and selecting a mapper for HC systems. The 3PCS allows for fair comparison of different heuristics.

The 3PCS for HC systems can help researchers who want to understand a mapper by giving them a “check list” of various characteristics of the workload model, platform model, and strategy attributes of the mapper. As such, it can help extend existing mapping work, and recognize important areas of research by facilitating understanding of the relationships that exist among previous efforts. The 3PCS also is useful for researchers who want to describe their mapper more thoroughly by using a common standard. In this regard, the 3PCS also may help researchers see the design and environment alternatives that they might not have otherwise considered during the development of new heuristics. Additionally, the 3PCS is useful when one wants to select a mapper to match a given real-world environment. All three parts of the 3PCS are needed to determine which heuristics would be appropriate for a given environment, and also to facilitate fair comparison of different heuristics.

ACKNOWLEDGEMENTS

This research was supported by the DARPA/ITO Quorum Program under NPS subcontract numbers N62271-98-M-0217 and N62271-98-M-0448, GSA subcontract number GS09K99BH0250, and, the Office of Naval Research Grant No. N00014-00-1-0599; by the DARPA/ITO AICE Program under contract numbers DABT63-99-C-0010 and DABT63-99-C-0012; by the DARPA/ISO BADD Program and the Office of Naval Research under grant number N00014-97-1-0804; by the Colorado State University George T. Abell Endowment; and by the Colorado State University Center for Robustness in Computer Systems (funded by the Colorado Commission on Higher Education Technology Advancement Group through the Colorado Institute of Technology).

The authors thank Ishfaq Ahmad, Richard F. Freund, Veljko Milutinovic, Janet Siegel and Jay Smith for their comments. We presented a preliminary version of portions of this material at the Workshop on Advances in Parallel and Distributed Systems.

REFERENCES

[1] Abawajy J.H., “Fault-tolerant scheduling policy for grid computing systems”, in: *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, April 2004, pp. 238–244.

[2] Ahmad I., Kwok Y.-K., “On exploiting task duplication in parallel program scheduling”, *IEEE Transactions on Parallel and Distributed Systems* **9** (9) (September 1998) 872–892.

[3] Ahmad I., Kwok Y.-K., Wu M.-Y., “Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors”, in: *2nd International Symposium on Parallel Architectures, Algorithms, and Networks*, vol. 6, June 1996, pp. 207–213.

[4] Ali S., “Robust resource allocation in dynamic distributed heterogeneous computing systems”, Ph.D. Thesis, School of Electrical and Computer Engineering, Purdue University, August 2003.

[5] Ali S., Braun T.D., Siegel H.J., Maciejewski A.A., “Heterogeneous computing”, in: Urban J., Dasgupta P. (Eds.), *Encyclopedia of Distributed Computing*, Kluwer Academic Publishers, Norwell, MA, 2004, in press.

[6] Ali S., Kim J.-K., Yu Y., Gundala S.B., Gertphol S., Siegel H.J., Maciejewski A.A., Prasanna V., “Greedy heuristics for resource allocation in dynamic distributed real-time heterogeneous computing systems”, in: *2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2002)*, vol. II, June 2002, pp. 519–530.

[7] Ali S., Kim J.-K., Yu Y., Gundala S.B., Gertphol S., Siegel H.J., Maciejewski A.A., Prasanna V., “Utilization-based techniques for statically mapping heterogeneous applications onto the HiPer-D heterogeneous computing system”, in: *Parallel and Distributed Computing Practices*, 2004, in press.

[8] Ali S., Kim J.-K., Yu Y., Gundala S.B., Gertphol S., Siegel H.J., Maciejewski A.A., Prasanna V., "Utilization-based heuristics for statically mapping real-time applications onto the HiPer-D heterogeneous computing system", in: *11th IEEE Heterogeneous Computing Workshop (HCW 2002) in the Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, April 2002.

[9] Ali S., Maciejewski A.A., Siegel H.J., Kim J.-K., "Measuring the robustness of a resource allocation", *IEEE Transactions on Parallel and Distributed Systems* **15** (7) (July 2004) 630–641.

[10] Ali S., Maciejewski A.A., Siegel H.J., Kim J.-K., "Robust resource allocation for distributed computing systems", in: *2004 International Conference on Parallel Processing (ICPP 2004)*, 2004.

[11] Ali S., Siegel H.J., Maheswaran M., Hensgen D., Sedigh-Ali S., "Representing task and machine heterogeneities for heterogeneous computing systems", *Tamkang Journal of Science and Engineering* **3** (3) (November 2000) 195–207, invited.

[12] Armstrong R., Hensgen D., Kidd T., "The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions", in: *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, March 1998, pp. 79–87.

[13] Aydin H., Melhem R., Mossé D., Mejía-Alvarez P., "Power-aware scheduling for periodic real-time tasks", *IEEE Transactions on Computers* **53** (5) (May 2004) 584–600.

[14] Bajaj R., Agrawal D.P., "Improving scheduling of tasks in a heterogeneous environment", *IEEE Transactions on Parallel and Distributed Systems* **15** (2) (February 2004) 107–118.

[15] Banino C., Beaumont O., Carter L., Ferrante J., Legrand A., Robert Y., "Scheduling strategies for master–slave tasking on heterogeneous processor platforms", *IEEE Transactions on Parallel and Distributed Systems* **15** (4) (April 2004) 319–330.

[16] Baskiyar S., SaiRanga P.C., "Scheduling directed a-cyclic task graphs on heterogeneous network of workstations to minimize schedule length", in: *2003 International Conference on Parallel Processing Workshops (ICPPW03)*, October 2003, pp. 97–103.

[17] Beaumont O., Legrand A., Robert Y., "Scheduling strategies for mixed data and task parallelism on heterogeneous clusters and grids", in: *11th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, February 2003, pp. 209–216.

[18] Berry P.M., "Uncertainty in scheduling: Probability, problem reduction, abstractions and the user", IEE Computing and Control Division Colloquium on Advanced Software Technologies for Scheduling, Digest No. 1993/163, April 26, 1993.

[19] Bölöni L., Marinescu D.C., "Robust scheduling of metaprograms", *Journal of Scheduling* **5** (5) (September 2002) 395–412.

[20] Braun T.D., Siegel H.J., Beck N., Bölöni L.L., Maheswaran M., Reuther A.I., Robertson J.P., Theys M.D., Yao B., Hensgen D., Freund R.F., "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems", *Journal of Parallel and Distributed Computing* **61** (6) (June 2001) 810–837.

[21] Braun T.D., Siegel H.J., Maciejewski A.A., "Heterogeneous computing: Goals, methods, and open problems", in: Monien B., Prasanna V.K., Vajapeyam S. (Eds.), *High Peformance Computing—HiPC 2001*, in: *Lecture Notes in Computer Science*, vol. 2228, Springer-Verlag, Berlin, 2001, pp. 307–318.

[22] Braun T.D., Siegel H.J., Maciejewski A.A., "Heterogeneous computing: Goals, methods, and open problems" (invited keynote presentation for the 2001 International Multiconference that included PDPTA 2001), in: *2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*, vol. I, June 2001, pp. 1–12.

[23] Braun T.D., Siegel H.J., Maciejewski A.A., "Static mapping heuristics for tasks with dependencies, priorities, deadlines, and multiple versions in heterogeneous environments", in: *16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, April 2002.

[24] Casavant T.L., Kuhl J.G., "A taxonomy of scheduling in general-purpose distributed computing systems", *IEEE Transactions on Software Engineering* **14** (2) (February 1988) 141–154.

[25] Chandra A., Gong W., Shenoy P., "Dynamic resource allocation for shared data centers using online measurements", in: *2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2003, pp. 300–301.

[26] Dally W., Towles B., *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, San Francisco, CA, 2003.

[27] Dhodhi M.K., Hielscher F.H., Storer R.H., "SHEMUS: Synthesis of heterogeneous multiprocessor systems", *Microprocessor and Microsystems* **19** (6) (August 1995) 311–319.

[28] Duato J., Yalmanchili S., Ni L., *Interconnection Networks: An Engineering Approach*, IEEE Computer Society Press, Los Alamitos, CA, 1997.

[29] Ekmečić I., Tartalja I., Milutinović V., "A taxonomy of heterogeneous computing", *IEEE Computer* **28** (12) (December 1995) 68–70.

[30] Ekmečić I., Tartalja I., Milutinović V., "A survey of heterogeneous computing: Concepts and systems", *Proceedings of the IEEE* **84** (8) (August 1996) 1127–1144.

[31] Eshaghian M.M. (Ed.), *Heterogeneous Computing*, Artech House, Norwood, MA, 1996.

[32] Eshaghian M.M., Wu Y.-C., "A portable programming model for network heterogeneous computing", in: Eshaghian M.M. (Ed.), *Heterogeneous Computing*, Artech House, Norwood, MA, 1996, pp. 155–195.

[33] Fernandez-Baca D., "Allocating modules to processors in a distributed system", *IEEE Transaction on Software Engineering* **SE-15** (11) (November 1989) 1427–1436.

[34] Foster I., Kesselman C. (Eds.), *The Grid 2: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, San Francisco, CA, 2004.

[35] Freund R.F., Gherrity M., Ambrosius S., Campbell M., Halderman M., Hensgen D., Keith E., Kidd T., Kussow M., Lima J.D., Mirabile F., Moore L., Rust B., Siegel H.J., "Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet", in: *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, March 1998, pp. 184–199.

[36] Freund R.F., Siegel H.J., "Heterogeneous processing", *IEEE Computer* **26** (6) (June 1993) 13–17.

[37] Gertphol S., Yu Y., Gundala S.B., Prasanna V.K., Ali S., Kim J.-K., Maciejewski A.A., Siegel H.J., "A metric and mixed-integer-programming-based approach for resource allocation in dynamic real-time systems", in: *16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, April 2002.

[38] Ghafoor A., Yang J., "Distributed heterogeneous supercomputing management system", *IEEE Computer* **26** (6) (June 1993) 78–86.

[39] Gomoluch J., Schroeder M., "Performance evaluation of market-based resource allocation for grid computing", *Concurrency and Computation: Practice and Experience* **16** (5) (April 2004) 469–475.

[40] Gribble S.D., "Robustness in complex systems", in: *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001, pp. 21–26.

[41] Hong I., Qu G., Potkonjak M., Srivastava M., "Synthesis techniques for low-power hard real-time systems on variable voltage processors", in: *19th IEEE Real-Time Systems Symposium (RTSS '98)*, December 1998, pp. 95–105.

[42] Jen E., "Stable or robust? What is the difference?", Santa Fe Institute Working Paper No. 02-12-069, 2002.

[43] Jensen M., "Improving robustness and flexibility of tardiness and total flowtime job shops using robustness measures", *Journal of Applied Soft Computing* **1** (1) (June 2001) 35–52.

[44] Kafil M., Ahmad I., "Optimal task assignment in heterogeneous computing systems", in: *6th IEEE Heterogeneous Computing Workshop (HCW '97)*, April 1997, pp. 135–146.

[45] Kafil M., Ahmad I., "Optimal task assignment in heterogeneous distributed computing systems", *IEEE Concurrency* **6** (3) (July–September 1998) 42–51.

[46] Kim J.-K., "Resource management in heterogeneous computing systems: Continuously running applications, tasks with priorities and deadlines, and power constrained mobile devices", Ph.D. Thesis, School of Electrical and Computer Engineering, Purdue University, August 2004.

[47] Kim J.-K., Hensgen D.A., Kidd T., Siegel H.J., John D.S., Irvine C., Levin T., Porter N.W., Prasanna V.K., Freund R.F., "A flexible multi-dimensional QoS performance measure framework for distributed heterogeneous systems", in: *Cluster Computing*, 2005, in press. Special issue on *Cluster Computing in Science and Engineering*.

[48] Kim J.-K., Shivle S., Siegel H.J., Maciejewski A.A., Braun T., Schneider M., Tideman S., Chitta R., Dilmaghani R.B., Joshi R., Kaul A., Sharma A., Sripada S., Vangari P., Yellampalli S.S., "Dynamic mapping in a heterogeneous environment with tasks having priorities and multiple deadlines", in: *12th IEEE Heterogeneous Computing Workshop (HCW 2003) in the Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, April 2003.

[49] Ko K., Robertazzi T.G., "Equal allocation scheduling for data intensive applications", *IEEE Transactions on Aerospace and Electronic Systems* **40** (2) (April 2004) 695–705.

[50] Krauter K., Buyya R., Maheswaran M., "A taxonomy and survey of grid resource management systems", Tech. rep., University of Mannitoba, Canada and Monash University, Australia, TR 2000-80, November 2000.

[51] Kwok Y.-K., Ahmad I., "Static scheduling algorithms for allocating directed task graphs to multiprocessors", *ACM Computing Surveys* **31** (4) (1999) 406–471.

[52] Kwok Y.-K., Maciejewski A.A., Siegel H.J., Ghafoor A., Ahmad I., "Evaluation of a semi-static approach to mapping dynamic iterative tasks onto heterogeneous computing systems", in: *1999 International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN '99)*, June 1999, pp. 204–209.

[53] Leangsuksun C., Potter J., Scott S., “Dynamic task mapping algorithms for a distributed heterogeneous computing environment”, in: *4th IEEE Heterogeneous Computing Workshop (HCW ’95),* April 1995, pp. 30–34.

[54] Leon V.J., Wu S.D., Storer R.H., “Robustness measures and robust scheduling for job shops”, *IEE Transactions* **26** (5) (September 1994) 32–43.

[55] Li Y.A., Antonio J.K., Siegel H.J., Tan M., Watson D.W., “Determining the execution time distribution for a data parallel program in a heterogeneous computing environment”, *Journal of Parallel and Distributed Computing* **44** (1) (July 1997) 35–52.

[56] Liszka K.J., Antonio J.K., Siegel H.J., “Problems with comparing interconnection networks: Is an alligator better than an armadillo?”, *IEEE Concurrency* **5** (4) (October–December 1997) 18–28.

[57] Lloyd S., “Complex systems: A review, ESD-WP-2003-01.16”, in: *ESD Internal Symposium, Working Paper Series,* Massachusetts Institute of Technology, May 2002.

[58] Magee C.L., de Weck O.L., “An attempt at complex system classification, ESD-WP-2003-01.02”, in: *ESD Internal Symposium, Working Paper Series,* Massachusetts Institute of Technology, May 2002.

[59] Maheswaran M., Ali S., Siegel H.J., Hensgen D., Freund R.F., “Dynamic mapping of a class of independent tasks onto heterogeneous computing systems”, *Journal of Parallel and Distributed Computing* **59** (2) (November 1999) 107–131.

[60] Maheswaran M., Braun T.D., Siegel H.J., “Heterogeneous distributed computing”, in: Webster J.G. (Ed.), *Encyclopedia of Electrical and Electronics Engineering,* vol. 8, John Wiley, New York, 1999, pp. 679–690.

[61] Marinescu D.C., Marinescu G.M., Ji Y., Bölöni L., Siegel H.J., “Ad hoc grids: Communication and computing in a power constrained environment”, in: *Workshop on Energy-Efficient Wireless Communications and Networks 2003 (EWCN 2003) in the Proceedings of the 22nd International Performance, Computing, and Communications Conference (IPCCC 2003),* April 2003.

[62] Noronha S.J., Sarma V.V.S., “Knowledge-based approaches for scheduling problems”, *IEEE Transactions on Knowledge and Data Engineering* **3** (2) (June 1991) 160–171.

[63] Park C.-I., Choe T.-Y., “An optimal scheduling algorithm based on task duplication”, *IEEE Transactions on Computers* **51** (4) (April 2002) 444–448.

[64] Prakash S., Parker A.C., “SOS: Synthesis of application-specific heterogeneous multiprocessor systems”, *Journal of Parallel and Distributed Computing* **16** (4) (December 1992) 338–351.

[65] Ravindran B., Li P., “DPR, LPR: Proactive resource allocation algorithms for asynchronous real-time distributed systems”, *IEEE Transactions on Computers* **53** (2) (February 2004) 201–216.

[66] Rotithor H.G., “Taxonomy of dynamic task scheduling schemes in distributed computing systems”, *IEE Proceedings on Computer and Digital Techniques* **141** (1) (January 1994) 1–10.

[67] Schuster P., “How does complexity arise in evolution: Nature’s recipe for mastering scarcity, abundance, and unpredictability”, *Complexity* **2** (December 1996) 22–30.

[68] Sevaux M., Sörensen K., “Genetic algorithm for robust schedules”, in: *8th International Workshop on Project Management and Scheduling (PMS 2002),* April 2002, pp. 330–333.

[69] Shivle S., Castain R., Siegel H.J., Maciejewski A.A., Banka T., Chindam K., Dussinger S., Pichumani P., Satyasekaran P., Saylor W., Sendek D., Sousa J., Sridharan J., Sugavanam P., Velazco J., "Static mapping of subtasks in a heterogeneous ad hoc grid environment", in: *13th IEEE Heterogeneous Computing Workshop (HCW 2004) in the Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, April 2004.

[70] Shivle S., Siegel H.J., Maciejewski A.A., Banka T., Chindam K., Dussinger S., Kutruff A., Penumarthy P., Pichumani P., Satyasekaran P., Sendek D., Sousa J.C., Sridharan J., Sugavanam P., Velazco J., "Mapping of subtasks with multiple versions in a heterogeneous ad hoc grid environment", in: *3rd International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar 2004) in the Proceedings of the 3rd International Symposium on Parallel and Distributed Computing in association with HeteroPar'04 (ISPDC/HeteroPar 2004)*, July 2004.

[71] Siegel H.J., *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, second ed., McGraw-Hill, New York, 1990.

[72] Siegel H.J., Dietz H.G., Antonio J.K., "Software support for heterogeneous computing", in: Tucker J.A.B. (Ed.), *The Computer Science and Engineering Handbook*, CRC Press, Boca Raton, FL, 1997, pp. 1886–1909.

[73] Siegel H.J., Siegel L.J., Kemmerer F., Mueller P.T. Jr., Smalley H.E. Jr., Smith S.D., "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition", *IEEE Transactions on Computers* **C-30** (12) (December 1981) 934–947.

[74] Singh H., Youssef A., "Mapping and scheduling heterogeneous task graphs using genetic algorithms", in: *5th IEEE Heterogeneous Computing Workshop (HCW '96)*, April 1996, pp. 86–97.

[75] Stankovic J.A., Spuri M., Natale M.D., Buttazzo G.C., "Implications of classical scheduling results for real-time systems", *IEEE Computer* **28** (6) (June 1995) 16–25.

[76] Sussman J.M., "Collected views on complexity in systems, ESD-WP-2003-01.01", in: *ESD Internal Symposium, Working Paper Series*, Massachusetts Institute of Technology, May 2002.

[77] Tan M., Siegel H.J., Antonio J.K., Li Y.A., "Minimizing the application execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system", *IEEE Transactions on Parallel and Distributed Systems* **8** (8) (August 1997) 857–871.

[78] Theys M.D., Siegel H.J., Chong E.K.P., "Heuristics for scheduling data requests using collective communications in a distributed communication network", *Journal of Parallel and Distributed Computing* **61** (9) (September 2001) 1337–1366.

[79] Theys M.D., Tan M., Beck N.B., Siegel H.J., Jurczyk M., "A mathematical model and scheduling heuristics for satisfying prioritized data requests in an oversubscribed communication network", *IEEE Transactions on Parallel and Distributed Systems* **11** (9) (September 2000) 969–988.

[80] T'kindt V., Billaut J.-C., "Some guidelines to solve multicriteria scheduling problems", in: *1999 IEEE International Conference on Systems, Man, and Cybernetics*, vol. 6, October 1999, pp. 463–468.

[81] Veeravalli B., Min W.H., “Scheduling divisible loads on heterogeneous linear daisy chain networks with arbitrary processor release times”, *IEEE Transactions on Parallel and Distributed Systems* **15** (3) (March 2004) 273–288.
[82] Wang L., Siegel H.J., Roychowdhury V.P., Maciejewski A.A., “Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach”, *Journal of Parallel and Distributed Computing* **47** (1) (November 1997) 8–22.
[83] Weiser M., Welch B., Demers A., Shenker S., “Scheduling for reduced CPU energy”, in: *USENIX Symposium on Operating Systems Design and Implementation*, November 1994, pp. 13–23.
[84] Xiao L., Chen S., Zhang X., “Dynamic cluster resource allocations for jobs with known and unknown memory demands”, *IEEE Transactions on Parallel and Distributed Systems* **13** (3) (March 2002) 223–240.
[85] Xu D., Nahrstedt K., Wichadakul D., “QoS and contention-aware multi-resource reservation”, *Cluster Computing* **4** (2) (April 2001) 95–107.
[86] Yang J., Ahmad I., Ghafoor A., “Estimation of execution times on heterogeneous supercomputer architecture”, in: *1993 International Conference on Parallel Processing (ICPP ’93)*, vol. I, August 1993, pp. 219–225.
[87] Zomaya A.Y., Teh Y.-H., “Observations on using genetic algorithms for dynamic load-balancing”, *IEEE Transactions on Parallel and Distributed Systems* **12** (9) (September 2001) 899–911.

Power Analysis and Optimization Techniques for Energy Efficient Computer Systems

WISSAM CHEDID AND CHANSU YU

Department of Electrical and Computer Engineering
Cleveland State University
2121 Euclid Avenue
Stilwell Hall 340
Cleveland, OH 44115
USA
wissam@chedid.com
c.yu91@csuohio.edu

BEN LEE

School of Electrical Engineering and Computer Science
Oregon State University
Owen Hall 302
Corvallis, OR 97331
USA
benl@eecs.orst.edu

Abstract
Reducing power consumption has become a major challenge in the design and operation of today's computer systems. This chapter describes different techniques addressing this challenge at different levels of system hardware, such as CPU, memory, and internal interconnection network, as well as at different levels of software components, such as compiler, operating system and user applications. These techniques can be broadly categorized into two types: Design time power analysis versus run-time dynamic power management. Mechanisms in the first category use analytical energy models that are integrated into existing simulators to measure the system's power consumption and thus help engineers to test power-conscious hardware and software during design time. On the other hand, dynamic power management techniques are applied during run-time, and

ISSN: 0065-2458/DOI 10.1016/S0065-2458(04)63004-X

are used to monitor system workload and adapt the system's behavior dynamically to save energy.

1. Introduction

Information processing engines are really seen as just "heat engines."
C. Mead and L. Conway, "*Introduction to VLSI Systems*," 1980

Innovations and improvements have long been made in computer and system architectures to essentially increase the computing power truly observing the Moore's Law for more than three decades. Improvements in semiconductor technology make it possible to incorporate millions of transistors on a very small die and to clock them at very high speeds. Architecture and system software technology also offer tremendous performance improvements by exploiting parallelism in a variety of forms. While the demand for even more powerful computers would be hindered by the physics of computational systems such as the limits on voltage and switching speed [39], a more critical and imminent obstacle is the power consumption and the corresponding thermal and reliability concerns [27]. This applies not only to low-end portable systems but also to high-end system designs.

Since portable systems such as laptop computers and cell phones draw power from batteries, reducing power consumption to extend their operating times is one of the most critical product specifications. This is also a challenge for high-end system designers because high power consumption raises temperature, which deteriorates performance and reliability. In some extreme cases, this requires an expensive, separate

Abbreviations

ACPI:	Advanced Configuration and Power Interface	**Flops:**	Floating-point Operation Per Second
APM:	Advanced Power Management	**GOP:**	Group Of Pictures
AVS:	Automatic Voltage Scaler	**I/O:**	Input/Output
BIOS:	Basic Input/Output Services	**IVS:**	Independent Voltage Scaling
CMOS:	Complementary Metal Oxide Semiconductor	**JVM:**	Java Virtual Machine
		LCD:	Liquid–Crystal Display
COPPER:	Compiler-controlled continuous Power-Performance	**MPEG:**	Moving Pictures Expert Group
		OS:	Operating System
CPU:	Central Processing Unit	**PLL:**	Phase-Locked Loop
CVS:	Coordinated Voltage Scaling	**POSE:**	Palm Operating System Emulator
DLS:	Dynamic Link Shutdown		
DPM:	Dynamic Power Management	**RTL:**	Register Transfer Language
DVS:	Dynamic Voltage Scaling	**RWEC:**	Remaining Worst-case Execution Cycles
DVS-DM:	DVS with Delay and Drop rate Minimizing	**SPEC:**	Standard Performance Evaluation Corporation
DVS-PD:	DVS with Predicting Decoding time	**VOVO:**	Varry-On/Varry-Off

power facility, as in the *Earth Simulator* [18], which achieves a peak performance of 40 Tflops but dissipates 5 MWatts of power.

This chapter provides a comprehensive survey of power analysis and optimization techniques proposed in the literature. Techniques for power efficient computer systems can be broadly categorized into two types: *Offline power analysis* and *dynamic power management* techniques. *Offline power analysis* techniques are based on analytical *energy models* that are incorporated into existing performance-oriented simulators to obtain power and performance information and help system architects select the best system parameters during design time. *Dynamic power management* (*DPM*) schemes monitor system workload and adapt the system's behavior to save energy. These techniques are dynamic, run-time schemes operating at different levels of a computer system. They include *Dynamic Voltage Scaling* (*DVS*) schemes that adjust the supply voltage and operating frequency of a processor to save power when it is idle [27,68]. A similar idea can be applied to I/O devices by monitoring their activities and turning them off or slowing them down when the demand on these devices is low [7,14,28]. Another possibility to conserve energy at run-time is when a system has more than one resource of the same kind, which is typically found in parallel and networked cluster systems. In this case, applying a DPM scheme in a coordinated way rather than applying it to individual resources independently can better manage the entire system. For instance, the DVS technique can be extended to a cluster system of multiple nodes by coordinating multiple DVS decisions [19].

This chapter is organized as follows. Section 2 discusses several energy models and the corresponding power analysis and optimization techniques integrated into existing simulation environments. These energy models cover various levels of a system with a varying degree of granularity and accuracy, which includes CPU-level, system-level, and parallel system-level power analysis techniques. Section 3 presents various hardware and software DPM techniques that also differ in granularity as well as accuracy. Fine-grained monitoring and power management is possible at a smaller scale but it may not be feasible at a larger scale because of the corresponding overhead of gathering information and making power-related decisions. Therefore, this section presents the various CPU-level, system-level, and parallel system-level DPM techniques. Section 4 provides a conclusion and discusses possible future research.

2. Power Analysis and Optimization Using Energy Models

Power dissipation has emerged as a major constraint in the design of processors and computer systems. Power optimization, just as with performance, requires careful design at several levels of the system architecture. The first step toward optimizing power consumption is to understand the sources of energy consumption at different levels. Various energy models have been developed and integrated with existing simulators or measurement tools to provide accurate power estimation, which can be used to optimize the system design.

Section 2.1 describes processor-based energy models that estimate power consumption at cycle- or instruction-level. Section 2.2 discusses system-based energy models that study power consumption of both hardware and software components. Finally, Section 2.3 targets multiprocessor–based or cluster-based energy models. These studies in particular focus on the system interconnect since energy performance of individual processors or nodes can be estimated based on techniques described in Sections 2.1 and 2.2. Table I summarizes these energy model-based offline approaches.

2.1 CPU-Level Energy Models

Power consumed by the CPU is a major part of the total power consumption of a computer system and thus has been the main target of power consumption analysis [9,10,49,65,70]. Several power models have been developed and integrated into existing performance simulators in order to investigate power consumption of CPU either on a functional unit basis or processor as a whole. These analyses are based on

TABLE I
TAXONOMY OF POWER ANALYSIS TECHNIQUES USING ENERGY MODELS

Type	Level of detail	Energy models	Simulation tools	Section
CPU	Cycle level or RTL	Power density-based or capacitance-based model for cycle-level simulation	*PowerTimer* [9], *Wattch* [10] and *SimplePower* [70]	2.1.1
	Instruction level	Instruction-based energy model with the measurement of instruction counts	Power profiles for *Intel 486DX2, Fujitsu SPARClite'934* [65] and *PowerPC* [49]	2.1.2
System	Hardware component level	State-based model (e.g., sleep/doze/ busy) for functional simulation	*POSE* (Palm OS Emulator) [16]	2.2.1
	Software component level	Process-based model with time-driven and energy-driven sampling	Time driven sampling, *PowerScope* [20], and energy driven sampling [12]	2.2.2
	Hardware and software component level	Component-specific energy models for complete system simulation	*SoftWatt* built upon *SimOS* system simulator [27]	2.2.3
Parallel system	Interconnection network architecture level	Bit energy model for bit-level simulation	*Simulink*-based tool [71]	2.3
		Message-based energy model for simulating interconnection network	*Orion*, the simulator for power-performance interconnection networks [67]	2.3

two abstraction levels; *cycle-level* (or *register-transfer level*) and *instruction-level* as described in the following two subsections, respectively.

2.1.1 Cycle-Level CPU Energy Model

Energy consumption of a processor can be estimated by using cycle-level architecture simulators. This is done by identifying the active (or busy) microarchitecture-level units or blocks during every execution cycle of the simulated processor [9,10, 70]. These cycle-by-cycle resource usage statistics can then be used to estimate the power consumption. An energy model describing how each unit or block consumes energy is a key component in any power-aware cycle-level simulators. Figure 1 illustrates a high-level block diagram of power-aware cycle-level simulators.

Brooks et al. presented two types of energy models for their *PowerTimer* simulator [9]:

(i) Power density-based energy model is used for components when detailed power and area measurements are available; and
(ii) analytical energy models are used for the rest of the components in a CPU.

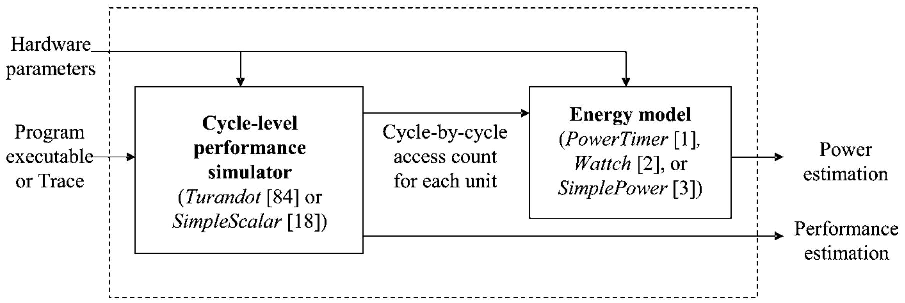

Hardware parameters
Program executable or Trace
Cycle-level performance simulator (*Turandot* [84] or *SimpleScalar* [18])
Cycle-by-cycle access count for each unit
Energy model (*PowerTimer* [1], *Wattch* [2], or *SimplePower* [3])
Power estimation
Performance estimation

FIG. 1. Block diagram of a power-aware, cycle-level simulator.

Analytical equations formulate the energy characteristics in terms of microarchitecture-level design parameters such as cache size, pipeline length, number of registers, etc. These two types of energy models were used in conjunction with a generic, parameterized, out-of-order superscalar processor simulator called *Turandot* [44]. Using *PowerTimer*, it is possible to study the power-performance trade-offs for different system configurations with varying resource sizes of caches, issue queues, rename registers, and branch predictor tables, which will help in building power-aware microarchitectures.

Wattch [10] and *SimplePower* [70] are two other CPU-level power-monitoring tools based on *SimpleScalar* [11], which is the most popular microarchitecture simulator. In *Wattch*, the energy models depend on the internal capacitances of the circuits that make up each unit of the processor. Each modeled unit falls into one of the following four categories: Array structures, memories, combinational logic and wires, and the clocking network. A different power model is used for each category and integrated in the *SimpleScalar* simulator to provide a variety of metrics such as power, performance, energy, and energy-delay product. Table II shows the energy expenditure of various components from measurements as well as from the *Wattch* simulator.

SimplePower, on the other hand, is based on *transition-sensitive energy model*, where each modeled functional unit has its own switch capacitance for every possible input transition [70]. This is then used to calculate the power consumed in a particular functional unit based on the input transition while executing a given instruction. *SimplePower* is used to evaluate the impact of an architectural modification as well as the effect of a high-level compiler optimization technique on system power. Example uses of *SimplePower* include selective gated pipeline technique to reduce the datapath switch capacitance, loop and data transformation to reduce the memory system power, and register relabeling to conserve power on the data buses [70].

TABLE II
COMPARISON OF POWER BREAKDOWNS BETWEEN MEASUREMENT (ALPHA 21264) AND ANALYTICAL ENERGY MODEL IN THE WATTCH SIMULATOR [10]

Hardware structure	Measurement (Alpha 21 264)	Analytical model (Wattch)
Caches	16.1%	15.3%
Out-of-order issue logic	19.3%	20.6%
Memory	8.6%	11.7%
Memory management unit	10.8%	11.0%
Floating point execution unit	10.8%	11.0%
Clocking network	34.4%	30.4%

2.1.2 Instruction-Level CPU Energy Model

In contrast to the fine-grain *cycle-level* techniques, coarse-grain *instruction-level* power analysis techniques estimate the total energy cost of a program by adding the energy consumed while executing instructions of a program [65,9]. Instruction-by-instruction energy costs, called *base costs*, can be measured for individual instructions for a target processor. However, there is extra power consumption due to "interaction" between successive instructions caused mainly by pipeline and cache effects. The base costs of individual instructions and the power cost of *inter-instruction effects* are determined based on the experimental procedure using a program containing several instances of the targeted instruction (for base cost measurement) and an alternating sequence of instructions (for inter-instruction effects costs). Table III illustrates a subset of the base costs for Intel 486DX2 and Fujitsu SPARClite'934 [65]. A similar study has also been conducted for PowerPC microprocessor [49].

Once the instruction-by-instruction energy model is constructed for a particular processor, the total energy cost, E_P, of any given program, P, is given by:

$$E_P = \sum_i (Base_i * N_i) + \sum_{i,j} (Inter_{i,j} * N_{i,j}) + \sum_k E_k \tag{1}$$

where $Base_i$ is the base cost of instruction i and N_i is the number of executions of instruction i. $Inter_{i,j}$ is the inter-instruction power overhead when instruction i is followed by instruction j, and $N_{i,j}$ is the number of times the (i, j) pair is executed. Finally, E_k is the energy contribution of other inter-instruction effects due to pipeline stalls and cache misses.

TABLE III
BASE COSTS FOR INTEL 486DX2 AND FUJITSU SPARCLITE ‘934 PROCESSORS [65]. (CYCLES AND ENERGY NUMBERS IN THE TABLE ARE PER-INSTRUCTION VALUES)

Intel 486DX2				Fujitsu SPARClite ‘934			
Instruction	Current (mA)	Cycles	Energy (10^{-8} J)	Instruction	Current (mA)	Cycles	Energy (10^{-8} J)
nop	276	1	2.27	nop	198	1	3.26
mov dx,[bx]	428	1	3.53	ld [10],i0	213	1	3.51
mov dx,bx	302	1	2.49	or g0,i0,10	198	1	3.26
mov [bx],dx	522	1	4.30	st i0,[10]	346	2	11.4
add dx,bx	314	1	2.59	add i0,o0,10	199	1	3.28
add dx,[bx]	400	2	6.60	mul g0,r29,r27	198	1	3.26
jmp	373	3	9.23	srl i0,1,10	197	1	3.25

2.2 Complete System-Level Energy Models

There is little benefit in studying and optimizing only the CPU core if other components have significant effect on or even dominate the energy consumption. Therefore, it is necessary to consider other critical components to reduce the overall system energy. Section 2.2.1 discusses the *hardware state-level models*, where the total energy consumption of the entire system is estimated based on the state each device is in or transitioning to/from. Here, it is assumed that each device is capable of switching into one of several power-saving states, such as sleep state, depending on the demand on that particular device [16]. This capability is usually provided in portable systems to extend their lifetimes as longer as possible. *Software-based* approaches presented in Section 2.2.2 identify energy hotspots in applications and operating system procedures and thus allow software programmers to remove bottlenecks or modify the software to be energy-aware. Finally, a *complete system level* simulation tool, which models the hardware components, such as CPU, memory hierarchy, and a low power disk subsystem as well as software components, such as OS and application, is presented in Section 2.2.3.

2.2.1 Hardware State-Based Energy Model

Cignetti et al. presented a system-wide energy optimization technique with a hardware state-based energy model [16]. This power model encapsulates low-level details of each hardware subsystem by defining a set of power states (e.g., sleep, doze or busy for CPU) for each device. Each power state is characterized by the power consumption of the hardware during the state, which is called *steady state power*. In addition, each transition between states is assigned an energy consumption cost,

TABLE IV
STEADY STATE AND TRANSIENT POWER OF A PALM DEVICE FROM IBM. (STEADY STATE POWER SHOWN IS THE RELATIVE VALUE TO THE DEFAULT STATE; CPU DOZE, LCD ON, BACKLIGHT OFF, PEN AND BUTTON UP. STATE TRANSITION IS CAUSED BY SYSTEM CALLS, WHICH ARE SHOWN ON THE RIGHT-HAND SIDE)

Steady state power		
Device	State	Power (mW)
CPU	Busy	104.502
	Idle	0.0
	Sleep	−44.377
LCD	On	0.0
	Off	−20.961
Backlight	On	94.262
	Off	0.0
Button	Pushed	45.796
Pen	On Screen	82.952
	Graffitti	86.029

Transient energy	
System Call	Transient energy (mJ)
CPU Sleep	2.025
CPU Wake	11.170
LCD Wake	11.727
Key Sleep	2.974
Pen Open	1.935

called *transient energy*. Since transitions between states occur as a result of system calls, the corresponding energy can be measured by keeping track of system calls. The total energy consumed by the system is then determined by adding the power of each device state multiplied by the time spent in that state plus the total energy consumption for all the transitions.

The above mentioned state-based energy model was implemented as an extension to the *Palm OS Emulator* (*POSE*) [48], which is a Windows based application that simulates the functionalities of a Palm device. POSE emulates *Palm OS* and instruction execution of the *Motorola Dragonball* microprocessor [43]. To quantify the power consumption of a device and to provide parameters to the simulator, measurements were taken in order to capture transient energy consumption as well as steady state power consumption as presented in Table IV [16]. A Palm device from IBM was connected to a power supply with an oscilloscope measuring the voltage across a small resistor. The power consumption of the basic hardware subsystems, such as CPU, LCD, backlight, buttons, pen, and serial link, was measured using measurement programs called *Power* and *Millywatt* [40].

2.2.2 Process-Based Energy Model

Since software is the main determinant for the activities of hardware components, such as the processor core, memory system and buses, there is a need for investigating energy-oriented software techniques and their interaction and integration with performance-oriented software design. This subsection presents process-based power measurement techniques for system optimization [12,20]. Using specially designed monitoring tools, these measurement-based techniques target the power consumption of the entire system and try to point out the hotspots in applications and operating system procedures. It is noted that these techniques are process-based in the sense that they assume different processes consume different amount of energy not only because they execute for different amount of time or different number of instructions but also because they use different sets of resources in different sequences.

In *PowerScope* [20], a *time-driven statistical sampler* is used to determine what fraction of the total energy is consumed, during a certain time period, due to specific processes in the system. This technique can be further extended to determine the energy consumption of different procedures within a process. By providing such a fine-grained feedback, *PowerScope* helps focus on those system components responsible for the bulk of energy consumption. Chang et al. presented a similar tool but it is based on *energy-driven statistical sampling*, which uses energy consumption to drive sample collection [12]. The *multimeter* [20] (or the *energy counter* [12]) monitors the power consumption of the system and the software under test by generating an interrupt for each time interval [20] (or each energy quanta [12]). This interrupt will prompt the system to record the process ID of the currently running process as well as to collect a current [20] (or energy [12]) sample. After the experiment, the collected data, i.e., process IDs and current/energy sample, is analyzed offline to match the processes with the energy samples to create the energy profile.

The result from this study showed that a non-trivial amount of energy was spent by the operating system compared to other user processes. In addition, there are often significant differences between time-driven and energy-driven profiles and therefore, it is necessary to carefully combine both sampling methods to obtain more accurate energy profile information.

2.2.3 Component-Specific Energy Model

Power profiling techniques mentioned above provide energy cost for executing a certain program but without understanding the overall system behaviors in sufficient detail to capture the interactions among all the system components. A complete system power simulator, *SoftWatt* [27] overcomes this problem by modeling hardware components such as CPU, memory hierarchy, and disk subsystem, and quantifying the power behavior of both application software and operating system. *SoftWatt* was built on top of *SimOS* infrastructure [55], which provides detailed simulation of both the hardware and software including the *IRIX* operating system [30]. In order to capture the complete system power behavior, *SoftWatt* integrates different analytical power models available from other studies into the different hardware components of *SimOS*. The modeled units in *Softwatt* include cache-structure, datapath, clock generation and distribution network, memory, and hard drive.

Experience with *Softwatt* running *JVM98* benchmark suite [59] from *SPEC* (*Standard Performance Evaluation Corporation*) [62] emphasized the importance of a complete system simulation to analyze the power impact of both architecture and OS on the execution of applications. From a system hardware perspective, the disk is the single largest power consumer of the entire system. However, with the adoption of a low-power disk, the power hotspot was shifted to the CPU clock distribution and generation network (similar results are shown in Table II). Also, the cache subsystem was found to consume more power than the processor core. From the software point of view, the user mode consumes more power than the kernel mode. However, certain kernel services are called so frequently that they accounted for significant energy consumption in the processor and memory hierarchy. Thus, taking into account the energy consumption of the kernel code is critical for reducing the overall energy cost. Finally, transitioning the CPU and memory subsystem to a low-power mode or even halting the processor when executing an idle process can considerably reduce power consumption.

2.3 Interconnect-Level Energy Models in Parallel Systems

After presenting energy models at the CPU-level (Section 2.1) and the system-level (Section 2.2), this section describes energy models at the parallel system-level with the focus on interconnection networks. With the ever-increasing demand for computing power, processors are becoming more and more interconnected to create large clusters of computers communicating through interconnection networks. Wang et al. showed that the power consumption of these communication components is becoming more critical, especially with increase in network bandwidth and capacity to the gigabit and terabit domains [67]. Thus, power analysis in this area usually targets the building blocks inside a network router and a switch fabric.

Bit energy model [71] considers the energy consumed for each bit, moving inside the switch fabric from the input to the output ports, as the summation of the bit energy consumed on each of the following three components:

(i) the internal node switches that direct a packet from one intermediate stage to the next until it reaches the destination port;
(ii) the internal buffer queues that store packets with lower priorities when contention occurs; and
(iii) the interconnect wires that dissipate power when the bit transmitted on the wire flips polarity from the previous bit.

Different models were employed for each one of these components based on their characteristics. For example, the bit energy of a node switch is state-dependent; it depends on the presence or absence of packets on other input ports. On the other hand, power consumption of the internal buffer can be expressed as the sum of data access energy (read and write) and the memory refreshing operation. Finally, the bit energy of interconnect wires depends on the wire capacitance, length, and coupling between adjacent wires. The bit energy model was incorporated into a *Simulink* [56] based bit-level simulation platform to trace the dataflow of every packet in the network to summarize the total energy consumption in the interconnect.

As opposed to the bit-level approach mentioned above, an architecture-level network power-performance simulator, *Orion*, was presented in [67]. Orion models an interconnection network as comprising of message generating (such as sources), transporting (router buffers, crossbars, arbiters, and link components), and consuming (sinks) agents. Each of these agents is a building block of the interconnection network, and is represented by an architecture-level energy model. This energy model is based on the switch capacitance of each component including both gate and wire capacitances. These capacitance equations are combined with the switching activity estimation to compute the energy consumption per component operation. Orion can be used to plug-and-play router and link components to form different network fabric architectures, run varying communication workloads, and study their impact on overall network power and performance.

3. Dynamic Power Management (DPM) Techniques

While the simulation and measurement techniques described in Section 2 aim to optimize power performance at design time, DPM techniques target energy consumption reduction at run-time by selectively turning off or slowing down components when the systems is idle or serving light workloads. As in Section 2, DPM techniques are applied in different ways and at different levels. For example, *Dynamic Voltage Scaling* (*DVS*) technique operates at the CPU-level and changes processor's supply voltage and operating frequency at run-time as a method of power management [68]. A similar technique, called *Dynamic Link Shutdown* (*DLS*), operates at the interconnect-level and puts communication switches in a cluster system into a low-power mode to save energy [32]. DPM techniques can also be used for shutting down idle I/O devices [49], or even nodes of server clusters [19,50].

As summarized in Table V, this section discusses DPM techniques that are classified based on the implementation level. Section 3.1 discusses DPM techniques applied at the *CPU-level*. In Section 3.2, *system-level* DPM approaches that consider other system components (memory, hard drive, I/O devices, display, etc.) than CPU are discussed. Finally, Section 3.3 presents DPM techniques proposed for *parallel systems*, where multiple nodes collaborate to save the overall power while collectively performing a given parallel task.

TABLE V
TAXONOMY OF DYNAMIC POWER MANAGEMENT TECHNIQUES

Type	Implemen-tation level	Monitoring mechanism	Control mechanism	Section
CPU	CPU-level	Monitor internal bus activity to reduce switching activity	Different encoding schemes [29,61,69], compiler-based scheduling [31,63,66]	3.1.1
		Monitor CPU instruction in execution to control clock supply to each component	Clock gating for CPU components [21,26]	3.1.2
		Monitor CPU workload to adjust supply voltage to CPU	DVS with interval-based or history-based scheduler [25,52,58,68], compiler-based scheduler [4,5,22,54]	3.1.3
System	Hardware device-based	Monitor device activities to shut it or slow it down	Timeout, predictive or stochastic policies [7,8,14,15,17,24,28,51,57,60]	3.2.1
	Software-based	Monitor device activity via application or system software to shut it or slow it down	Prediction of future utilization of device [24,35,36,38], ACPI [1,2,47]	3.2.2
Parallel system	Hardware-based	Monitor multiple CPU's workloads to cooperatively adjust supply voltages	CVS (Coordinated DVS) [19]	3.3.1
		Monitor switch/router activity to rearrange connectivity or put into reduced power mode	History-based DVS on switch/router [53], Dynamic Link Shutdown [32]	3.3.1
	Software-based	Monitor synchronization activities to power down spinning nodes	Thrifty barrier [34]	3.3.2
		Monitor workload distribution to shut off some nodes	Load unbalancing [50]	3.3.2

3.1 CPU-Level DPM

The intuition behind power saving at the CPU-level comes from the basic energy consumption characteristics of digital static CMOS circuits, which is given by

$$E \propto C_{\mathrm{eff}} V^2 f_{\mathrm{CLK}} \tag{2}$$

where C_{eff} is the effective switching capacitance of the operation, V is the supply voltage, and f_{CLK} is the clock frequency [25]. The DPM techniques presented in this section reduce the power consumption by targeting one or more of these parameters.

Section 3.1.1 discusses techniques to reduce the switching activity of the processor, mainly at the datapath and buses. In Section 3.1.2, *clock gating* techniques are discussed, which reduce power consumption by turning off the idle component's clock, i.e., $f_{\mathrm{CLK}} = 0$. Finally, Section 3.1.3 presents one of the most promising, and also the most complicated, CPU-level DPM technique based on DVS. DVS scales both V and f_{CLK} to serve the processor workload with the minimum required power. If applied properly, DVS allows substantial energy saving without affecting performance.

3.1.1 Reducing Switching Activity

As discussed earlier, reducing switching activity plays a major role in reducing power consumption. A number of such optimization techniques have been proposed to reduce switching activity of internal buses [29,61,69] and functional units [31,63, 66] of a processor. In case of buses, energy is consumed when wires change states (between 0 and 1). Different techniques are used to reduce the switching activity on buses by reducing the number of wire transitions. Stan and Burleson proposed *bus-invert coding* where the bus value is inverted when more than half the wires are changing state [61]. In other words, when the new value to be transmitted on the bus differs by more than half of its bits from the previous value, then all the bits are inverted before transmission. This reduces the number of state changes on the wire, and thus, saves energy.

Henkel and Lekatsas proposed a more complicated approach where cache tables are used on the sending and receiving sides of the channel to further reduce transitions [29]. That is, when a value "hit" is observed at the input of the channel, the system will only send the index of the cache entry instead of the whole data value, which will reduce the number of transitions. Finally, Wen et al. used *bus transcoding* to reduce bus traffic and thus power based on data compression on bus wires [69]. As an enhancement to this technique, *transition coding* was also proposed where the encoding of data represents the wire changes rather than the absolute value, which simplifies the energy optimization problem.

On the other hand, the processor's switching activity can also be reduced by using power-aware compiler techniques. Although applied at compile time, these are considered as DPM techniques because their effect is closely tied to the system's run-time behavior. For example, in *instruction scheduling* technique [63,66], instructions are reordered to reduce the switching activity between successive instructions. More specifically, it minimizes the switching activity of a data bus between the on-chip cache and main memory when instruction cache misses occur [66]. *Cold scheduling* [63] prioritizes the selection of the next instruction to execute based on the energy cost of placing that instruction into the schedule. Another compiler based technique called *register assignment* [31] focuses on reducing the switching activity on the bus by re-labeling the register fields of the compiler-generated instructions. A simulator, such as *SimplePower* [70], is used to parameterize the compiler with sample traces. In other words, it records the transition frequencies between register labels in the instructions executed in consecutive cycles and this information is then used to obtain a better encodings for the registers such that the switching activity and consequently the energy consumption on the bus is reduced.

3.1.2 Clock Gating

Clock gating involves freezing the clock of an idle component. Energy is saved because no signal or data will propagate in these frozen units. Clock gating is widely used because it is conceptually simple; the clock can be restarted by simply deasserting the clock-freezing signal. Therefore, only a small overhead in terms of additional circuitry is needed, and the component can transit from an idle to an active state in only a few cycles. This technique has been implemented in several commercial processors such as Alpha 21264 [26] and PowerPC 603 [21]. The Alpha 21264 uses a hierarchical clocking architecture with gated clocks. Depending on the instruction to be executed, each CPU unit (e.g., floating point unit) is able to freeze the clock to its subcomponents (e.g., adder, divider and multiplier in floating point unit).

The PowerPC 603 processor supports several sleep modes based on clock gating. For this purpose, it has two types of clock controllers: *global* and *local*. Clocks to some components are globally controlled while others are locally controlled. For example, consider *PLL* (*Phase Locked Loop*) that acts mainly as a frequency stabilizer and does not depend on global clock. Even though clocks to all units are globally disabled and the processor is in sleep state, the PLL can continue to function which makes a quick wake-up (within ten clock cycles) possible. On the other hand, if the PLL is also turned off, maximum power saving would be achieved but the wake-up time could be as long as 100 μs, to allow the PLL to relock to the external clock.

3.1.3 Dynamic Voltage Scaling (DVS)

In contrast to clock gating, which can only be applied to idle components, DVS targets components that are in active state, but serving a light workload. It has been proposed as a means for a processor to deliver high performance when required, while significantly reducing power consumption during low workload periods. The advantage of DVS can be observed from the power consumption characteristics of digital static CMOS circuits (2) and the clock frequency equation:

$$delay \propto \frac{V}{(V - V_k)^{\alpha}} \quad \text{and} \quad f_{\mathrm{CLK}} \propto \frac{(V - V_k)^{\alpha}}{V} \qquad (3)$$

where V is the supply voltage, and f_{CLK} is the clock frequency. α ranges from 1 to 2, and V_k depends on threshold voltage at which *velocity saturation*[1] occurs [25].

Decreasing the power supply voltage would reduce power consumption quadratically as shown in equation (2). However, this would create a higher propagation delay and at the same time force a reduction in clock frequency as shown in equation (3). While it is generally desirable to have the frequency set as high as possible for faster instruction execution, the clock frequency and supply voltage can be reduced for some tasks where maximum execution speed is not required. Since processor activity is variable, there are idle periods when no useful work is being performed and DVS can be used to eliminate these power-wasting idle times by lowering the processor's voltage and frequency.

In order to clearly show the advantage of DVS techniques, Figure 2 compares DVS with the simple On/Off scheme, where the processor simply shuts down when it is idle (during time 2–4, 5–7 and 8.5–11 in the figure). DVS reduces the voltage and frequency, spreading the workload to a longer period, but more than quadratically reducing energy consumption. A quick calculation from Figure 2 shows about 82% reduction in power based on equation (2) because

$$\begin{aligned} E_{\text{DVS}}/E_{\text{On/Off}} &= \bigl(4\times(0.5)^3+3\times(0.33)^3+4\times(0.375)^3\bigr)\\ &\quad /\bigl(2\times 1^3+1\times 1^3+(1.5)\times 1^3\bigr)\\ &= 0.82/4.5 = 0.18. \end{aligned}$$

Note that each task workload, which is represented by the area inside the rectangle in Figure 2, remains the same for both the simple On/Off and DVS mechanisms.

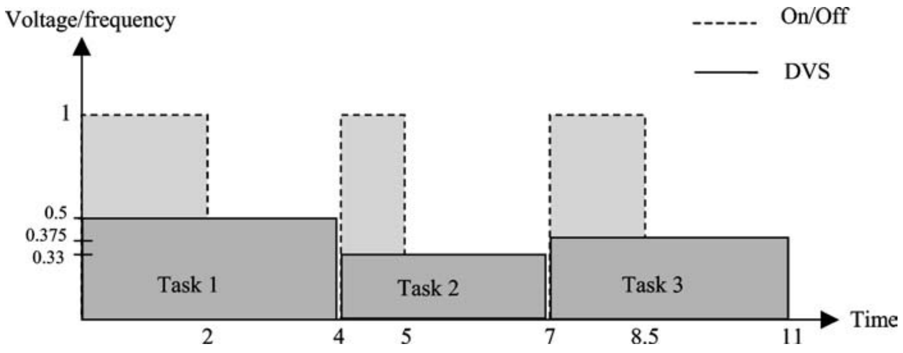


FIG. 2. Voltage scheduling graph with On/Off and DVS mechanisms.

[1] Velocity saturation is related to the semiconductor voltage threshold after which saturation occurs and the transistor's behavior becomes non-linear.

TABLE VI
CLOCK FREQUENCY VERSUS SUPPLY VOLTAGE FOR THE MOBILE INTEL PENTIUM III PROCESSOR [41]

Maximum performance mode			Battery optimized mode		
Frequency (MHz)	Voltage (V)	Max. power consumption (Watt)	Frequency (MHz)	Voltage (V)	Max. power consumption (Watt)
500	1.10	8.1	300	.975	4.5
600	1.10	9.7	300	.975	4.5
600	1.35	14.4	500	1.10	8.1
600	1.60	20.0	500	1.35	12.2
650	1.60	21.5	500	1.35	12.2
700	1.60	23.0	550	1.35	13.2
750	1.35	17.2	500	1.10	8.1
750	1.60	24.6	550	1.35	13.2
800	1.60	25.9	650	1.35	15.1
850	1.60	27.5	700	1.35	16.1
900	1.70	30.7	700	1.35	16.1
1000	1.70	34.0	700	1.35	16.1

Current custom and commercial CMOS chips are capable of operating reliably over a range of supply voltages [46,64] and there are a number of commercially available processors that support DVS mechanisms. Table VI shows the Mobile Intel Pentium III processor with 11 frequency levels and 6 voltage levels with two performance modes: *Maximum performance* mode and *battery optimized performance* mode [41]. The maximum performance mode is designed to provide the best performance while the battery optimized performance mode provides the balance between performance and battery lifetime. *Crusoe* processor from Transmeta, Inc. also has variable voltage and frequency as presented in Table VII.

TABLE VII
CLOCK FREQUENCY VERSUS SUPPLY VOLTAGE FOR THE TRANSMETA CRUSOE PROCESSOR [33]

Frequency (MHz)	Voltage (V)	Power consumption (Watt)
667	1.6	5.3
600	1.5	4.2
533	1.35	3.0
400	1.225	1.9
300	1.2	1.3

The main challenge in applying DVS is to know when and how to scale the voltage and frequency. In the following discussion, three different voltage schedulers are presented: *Interval-based*, *inter-task*, and *intra-task* scheduler. Interval-based scheduler is a time-based voltage scheduler that predicts the future workload using the workload history. Inter-task and intra-task schedulers target real-time applications with deadlines to meet for tasks. Inter-task scheduler changes speed at each task boundary, while intra-task scheduler changes speed within a single task with the help from compilers. Inter-task approaches make use of a prior knowledge of the application to produce predictions for the given task, while intra-task approaches try to take advantage of slack time that results from the difference in program execution path caused by conditional statements.

3.1.3.1 Interval-Based Scheduler.

Interval-based voltage schedulers [25,68] divide time into uniform length intervals and analyze CPU utilization of the previous intervals to determine the voltage/frequency of the next interval. Govil et al. discussed and compared seven such algorithms [25]:

(i) PAST uses the recent past as a predictor of the future.
(ii) FLAT simply tries to smooth the processor speed to a global average.
(iii) LONG_SHORT attempts to find a golden mean between the most recent behavior and a more long-term average.
(iv) AGED_AVERAGES employs an exponential-smoothing method, attempting to predict via a weighted average.
(v) CYCLE is a more sophisticated prediction algorithm that tries to take advantage of previous *run_percent* values that have cyclical behavior, where *run_percent* is the fraction of cycles in an interval during which the CPU is active.
(vi) PATTERN is a generalized form of CYCLE that attempts to identify the most recent *run_percent* values as a repeating pattern.
(vii) PEAK is a more specialized version of PATTERN and uses the following heuristics based on observation on narrow peaks: Increasing *run_percents* would fall but decreasing *run_percents* would continue falling [25].

According to their simulation studies, simple algorithms based on rational smoothing rather than complicated prediction schemes showed better performance. Their study also shows that further possibilities exist by improving predictions, such as sorting past information by process-type or providing useful information by applications [25].

3.1.3.2 Inter-Task Techniques for Real-Time Applications.

Interval-based scheduler is simple and easy to implement but it often incorrectly predicts future workloads and degrades the quality of service. In non-real-time applications, unfinished task from the previous interval would be completed in later intervals and does not cause any serious problems. However, in real-time applications, tasks are specified by the task start time, the computational resources required, and the task deadline. Therefore, the voltage/frequency scaling must be carried out under the constraint that no deadlines are missed. An optimal schedule is defined to be the one for which all tasks complete on or before deadlines and the total energy consumed is minimized.

For a set of tasks with the given timing parameters, such as deadlines, constructing the optimal voltage schedule requires super-linear algorithmic complexity. One simple heuristic algorithm is to identify the task with the earliest deadline and find the minimum constant speed needed to complete the task within the time interval before deadline. Repeating the same procedure for all tasks provides a voltage schedule. Quan and Fu suggested a more efficient inter-task scheduling algorithm for real-time applications [52]. This approach tries to find the critical intervals using the given timing parameters, such as start times and deadlines, which can be bottlenecks in executing a set of tasks. Then, a voltage schedule is produced for the set of critical intervals, and a complete low-energy voltage schedule is constructed based on the minimum constant speed found during any critical interval. Although this greedy approach guarantees minimum peak power consumption, it may not always produce the minimum-energy schedule.

Another inter-task DVS technique has been proposed for a specific real-time application, MPEG player [13,58]. The task here is to decode an MPEG frame or a *group of pictures* (*GOP*) [45]. Since different frames require an order of different computational overhead for decoding, it is more beneficial to change the supply voltage and operating frequency depending on frames rather than GOP. The main difficulty is to predict the next workload (e.g., decoding the next frame) in order to assign a proper voltage and frequency setting. If the next workload (frame decoding time) is underestimated, a voltage/frequency will be assigned that is lower than required, and the job will not meet its deadline causing either jitters or frames to be dropped and the video quality will degrade. On the other hand, if the next workload is overestimated, a voltage/frequency that is higher than required will be assigned, leading to more power consumption than necessary.

Son et al. proposed two heuristic DVS algorithms for MPEG decoding [58]: *DVS-DM* (*DVS with delay and drop rate minimizing algorithm*) and *DVS-PD* (*DVS with decoding time prediction*). DVS-DM is an interval-based DVS in the sense that it schedules voltage at every GOP boundary based on parameters (mainly delay and drop rate) obtained from previous decoding history. DVS-PD determines the voltage based on information from the next GOP (like frame sizes and frame types) as well as previous history. Since frames exhibit different characteristics depending on the

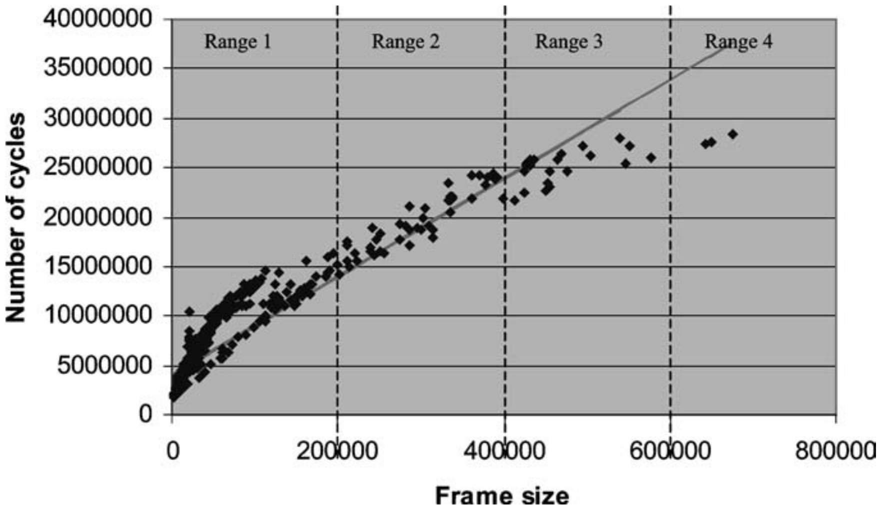


FIG. 3. Decode time as a function of frame size (based on the movie, "Undersiege").

frame type, DVS-PD offers higher prediction accuracy for future workload compared to DVS-DM [58].

Chedid proposed another set of techniques for power aware MPEG decoding [13]: *regression*, *range-avg* and *range-max*. The *regression* technique is based on the observation that the frame-size/decoding-time distribution follows a linear regression model [6] with high accuracy as shown in Figure 3. The regression line is built dynamically at run-time by calculating the slope of the frame-size/decoding-time relationship based on past history. The other two techniques, *range-avg* and *range-max*, alleviate the computational overhead found in the regression algorithm. These approaches divide the decoding-time/frame-size distribution into several ranges as in Figure 3 and make estimation decision based on the average decoding time (*range-avg*) or the maximum decoding time (*range-max*) in each range. The accuracy of these two techniques is only slightly worse than regression, but has the advantages of lower complexity and being able to dynamically increase or decrease the range size in order to better respond to any system requirement such as more power reduction or better video quality [13].

Table VIII summarizes the different inter-task DVS techniques for MPEG decoding discussed in the previous paragraphs.

3.1.3.3 Intra-Task Techniques for Real-Time Applications.

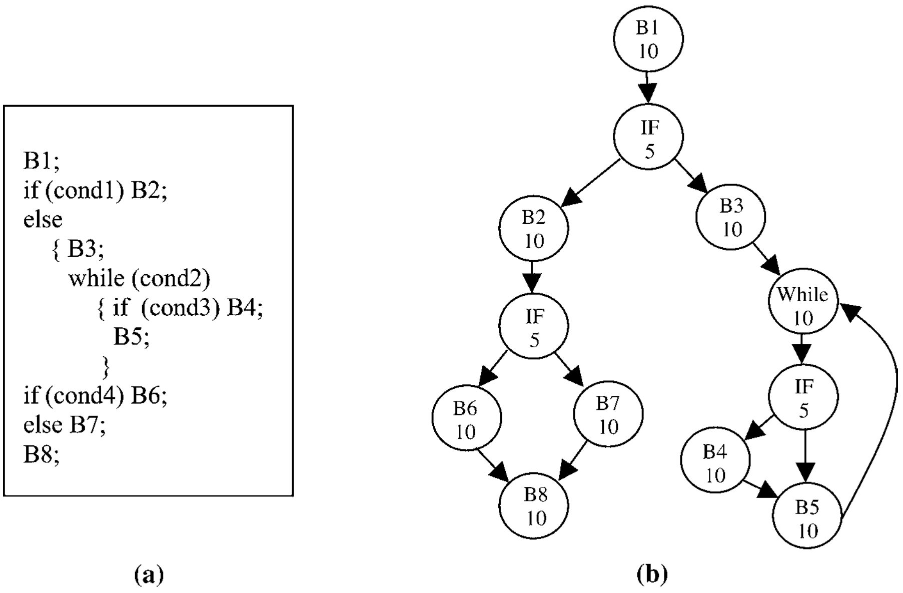
As opposed to the inter-task DVS techniques mentioned above, where voltage/frequency

TABLE VIII
INTER-TASK DVS TECHNIQUES FOR A REAL-TIME APPLICATION (MPEG PLAYER)

Technique	Implementation level	Method used to predict future workload	Advantages	Disadvantages
DVS-DM	GOP (Group of Pictures)	Previous history of delay and drop rate	Easy to implement	Inaccurate if decoding workload fluctuates
DVS-PD	GOP (Group of Pictures)	Weighted average of previous history and next GOP information	More accurate and less vulnerable to fluctuations than *DVS-DM*	Vulnerable to fluctuations between frames within each GOP
Regression	Picture frame	Dynamic regression of previous history and next frame information	Highly accurate prediction	Computationally expensive
Range-avg	Picture frame	Average workload of past picture frames with similar frame type and size	Easy to implement and flexible in balancing between power saving and video quality	Less accurate than *Regression*
Range-max	Picture frame	Maximum workload of past picture frames with similar frame type and size	Easy to implement and more flexible than *Range-avg*	Less accurate than *Regression* and *Range-avg*

changes occur between consecutive tasks, intra-task DVS techniques are applied during the execution of a task with the help of a power-aware compiler. The compiler identifies different possible execution paths within a task, each requiring a different amount of work and thus different voltage/frequency setting. Consider an example of a real-time task and its flow graph in Figure 4. In Figure 4(b), each node represents a basic block, B_i, of this task and the number in each node denotes the number of execution cycles required to complete the block. The total number of cycles varies for the same task depending on the chosen path and the resultant slack time is the target of optimization in the following intra-task techniques.

Azevedo et al. introduced an intra-task DVS technique using *program checkpoints* under compiler control [5]. Checkpoints indicate places in a program where the processor voltage/frequency should be re-calculated and scaled. The program is profiled, using a representative input data set, and information about minimum/maximum energy dissipated and cycle count between checkpoints is collected. This information is used in a run-time voltage scheduler to adjust the voltage in an energy efficient way, while meeting the deadline.



```
B1;
if (cond1) B2;
else
   { B3;
     while (cond2)
        { if  (cond3) B4;
          B5;
        }
if (cond4) B6;
else B7;
B8;
```

FIG. 4. Intra-task paths. (a) Example program and (b) its flow graph (each circle representing a basic block of a task and the number representing the cycles to execute the block).

Similarly, Shin and Kim proposed a compiler-based conversion tool, called *Automatic Voltage Scaler* (*AVS*), that converts DVS-unaware programs into DVS-aware ones [54]. The compiler profiles a program during compile-time and annotates the *Remaining Worst-case Execution Cycles* (*RWEC*) information, which represents the remaining worst-case execution cycles among all the execution paths that start from each corresponding checkpoint. It automates the development of real-time power-aware programs on a variable-voltage processor in a way completely transparent to software developers.

In the previously discussed approaches, voltage/frequency scaling must be computed and executed at every checkpoint, which may introduce an uncontrollable overhead at run-time. Ghazaleh et al. reported a similar compiler-based approach but requires collaboration between the compiler and the operating system [22]. As before, the compiler annotates the checkpoints with the RWEC temporal information. During program execution, the operating system periodically adapts the processor's voltage and frequency based on this temporal information. Therefore, this approach separates the checkpoints into two categories: The first one is only used to compute the temporal information and adjust the dynamic run-time information. The second one is used by the OS (which has more information on the overall application behavior) to execute the voltage/frequency change. This approach relies on the strengths of both the compiler and OS to obtain fine-grain information about an application's execution to optimally apply DVS.

COPPER (*Compiler-Controlled Continuous Power-Performance*) [4] is another compiler-based approach that also relies on the characteristics of the microarchitecture to optimize the power performance of the application. Among many possibilities, it focuses on combining dynamic register file reconfiguration with voltage/frequency scaling. During compile time, different versions of the given program code are produced under varying architectural parameters, mainly the number of available registers, and the corresponding power profiles are evaluated using energy simulator such as Wattch presented in Section 2.1.1. Since running a code version compiled for less number of registers may lead to lower energy consumption but higher execution delay, it is possible to tradeoff between the average power consumption and the execution time with code versioning. The run-time system selects a code version to help achieve performance goals within a given energy constraints.

3.2 Complete System-Level DPM

As discussed before, the CPU does not dominate the power consumption of the entire system. Other system components, such as disk drives and displays, have a much larger contribution. Therefore, it is necessary to consider all of the critical components of the system to effectively optimize power. A well-known system-level power management technique is shutting down hard drives and displays when they are idle. A similar idea can also be applied to other I/O devices to save energy. However, changing power states of hardware components incurs not only time delay but also energy overhead. Consequently, a device should be put to sleep only if the energy saved justifies the overhead. Thus, the main challenge in successfully applying this technique is to know when to shut down the devices and to wake them up.

A straightforward method is to have individual devices make such decisions by monitoring their own utilization. One clear advantage of this device-based scheme (Section 3.2.1) is transparency, i.e., energy saving is achieved without involving or changing application or system software. On the contrary, this scheme may perform poorly because it is unaware of the tasks requesting the service of the device. Software-based DPM techniques (Section 3.2.2) have been proposed to alleviate this problem. Application or system software takes full responsibility on power-related decisions assuming that devices can operate in several low power modes using control interfaces such as *Advanced Configuration and Power Interface* (*ACPI*) [2].

3.2.1 Hardware Device-Based DPM Policies

Hardware device-based policies observe hardware activities and workloads of the target device and change power states accordingly. They are usually implemented in hardware or device drivers without direct interaction with application or system software as illustrated in Figure 5. Based on prediction mechanisms for future device usage, these methods can be classified into three categories: *Time-out*, *Predictive*, and *Stochastic* policies [7,37].

For time-out policies, *break-even time*, T_{BE}, is defined as the minimum time length of an idle period during which shutting down a particular device will save power. When transition power, P_{TR}, and transition time, T_{TR}, (required when changing power states of the device) are negligible, T_{BE} is zero because there is no delay and energy overhead for shutting down and waking up the device. In practice, T_{BE} is calculated based on P_{TR} and T_{TR} as in Figure 6. Time-out policies work as follows: When an idle period begins, a timer is started with a predefined duration $T_{timeout}$, which is usually set as a certain fraction of T_{BE}. If the device remains idle after $T_{timeout}$, then the power manager makes the transition to the low-power or off state because it assumes that the device will remain idle for at least another T_{BE} seconds. These policies are relatively easy to implement but they have two major drawbacks. First, a large amount of energy is wasted waiting for the timeout to expire, during which the device is idle but still fully powered. Second, there is always a performance penalty to wakeup devices upon receiving a request signal. Devices typically have a long wakeup time, and thus the request to wake up may incur significant delays.

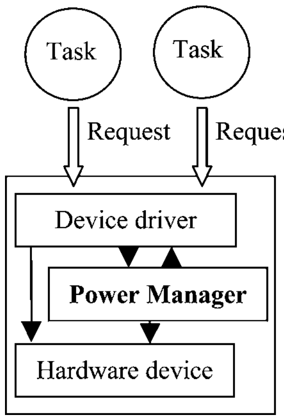


FIG. 5. Hardware device-based DPM.

T_{TR}:	transition time required to enter ($T_{\mathrm{On,Off}}$) and exit ($T_{\mathrm{Off,On}}$) the inactive state;
P_{TR}:	transition power;
P_{On}, P_{Off}:	power when device is On and Off;
T_{BE}:	break-even time, the minimum length of an idle period during which shutting down the device will save power.

$$T_{\mathrm{TR}} = T_{\mathrm{On,Off}} + T_{\mathrm{Off,On}},$$

$$P_{\mathrm{TR}} = (T_{\mathrm{On,Off}} P_{\mathrm{On,Off}} + T_{\mathrm{Off,On}} P_{\mathrm{Off,On}})/T_{\mathrm{TR}},$$

$$T_{\mathrm{BE}} = \begin{cases} T_{\mathrm{TR}}, & \text{if } P_{\mathrm{TR}} \leqslant P_{\mathrm{On}}, \\ T_{\mathrm{TR}} + T_{\mathrm{TR}}(P_{\mathrm{TR}} - P_{\mathrm{On}})/(P_{\mathrm{On}} - P_{\mathrm{Off}}) & \text{if } P_{\mathrm{TR}} > P_{\mathrm{On}}, \end{cases}$$

where "$T_{\mathrm{TR}}(P_{\mathrm{TR}} - P_{\mathrm{On}})/(P_{\mathrm{On}} - P_{\mathrm{Off}})$" represents the additional time needed to spend in the Off state to compensate the excess power consumed during state transition.

FIG. 6. Calculation of break-even time, T_{BE} [7].

Predictive policies counter the drawbacks of the time-out policies using techniques such as *predictive shutdown* [15,17,24,60] and *predictive wakeup* [28]. The predictive shutdown policy eliminates the time-out period by predicting the length of an idle period beforehand. Srivastava et al. suggested that the length of an idle period can be predicted by the length of the previous busy period [60]. Chung et al. observed the pattern of idle periods, and then the length of the current idle period is predicted by matching the current sequence that led to this idle period with the observed history of idle periods [15]. In [17,24], researchers suggested dynamically adjusting T_{timeout} based on whether the previous predictions were correct or not. The predictive wakeup policy reduces the performance penalty by waking up the device on time so that it becomes ready when the next request arrives. Hwang et al. employed the *exponential-average* (weighted-average with exponential weight values) approach to predict the wake-up time based on past history [28]. This policy may increase power consumption but will decrease the delay for serving the first request after an idle period.

One of the shortcomings of predictive policies is that they assume a deterministic arrival of device requests. *Stochastic policies* model the arrival of requests and device power-state changes as stochastic processes, e.g., *Markov processes*. Benini et al. modeled the arrival of I/O requests using stationary discrete-time Markov processes [8]. This model was used to achieve optimal energy saving by shutting down and waking up a device in the most efficient way in terms of energy as well as performance. In this model, time is divided into small intervals with the assumption that the system can only change its state at the beginning of a time interval. Chung et al. extended the model by considering non-stationary processes [14]. They pre-computed the optimal schedule for different I/O request patterns, and at run-time these schedules are used to more accurately estimate the next I/O request time.

However, for discrete-time Markov models, the power manager needs to send control signals to the components every time interval, which may result in heavy signal traffic and therefore more power dissipation. Qiu et al. used continuous-time Markov models to help prevent this “periodic evaluation” and instead used event-triggered evaluation [51]. They consider both request arrival and request service events, upon which the system determines whether or not to shut down a device. Finally, Simunic et al. suggested adding timeout to continuous-time Markov models so that a device would be shut down if it has been continuously idle for a predefined timeout duration [57]. In general, stochastic policies provide better performance than predictive and time-out policies. In addition, they are capable of managing multiple power states, making decisions not only *when* to perform state transition but also *which* transition should be made. The main disadvantage of these policies is that they require offline preprocessing and are more complicated to implement. For a detailed comparison of above mentioned device-based DPM schemes, please refer to [7,37].

3.2.2 Software-Based DPM Policies

While hardware device-based power management policies can optimize energy-performance of individual devices, they do not consider system-wide energy consumption due to the absence of global information. Therefore, software-based DPM policies have been proposed to handle system-level power management. *Advanced Configuration and Power Interface* (*ACPI*) [2], proposed as an industrial standard by Intel, Microsoft and Toshiba, provides a software-hardware interface allowing power managers to control the power of various system components. Application and operating system-based DPM techniques, which will be discussed later in this section, utilize such interface to conserve power. Although application-based schemes can be the most effective because future workloads are best known to applications, OS-based schemes have a benefit that existing applications do not need to be re-written for energy savings.

3.2.2.1 Advanced Configuration and Power Interface (ACPI).

ACPI [2] evolved from the older *Advanced Power Management* (*APM*) standard targeting desktop PCs [3]. Implemented at the BIOS (Basic I/O Services)-level, APM policies are rather simple and deterministic. Application and OS make normal BIOS calls to access a device and the APM-aware BIOS serve the I/O requests while conserving energy. One advantage of APM is that the whole process is transparent to application and OS software. ACPI is a substitute for APM at the system software level. Unlike APM, ACPI does not directly deal with power management. Instead, it exposes the power management interfaces for various hardware devices to the OS and the responsibility of power management is left to application or operating system

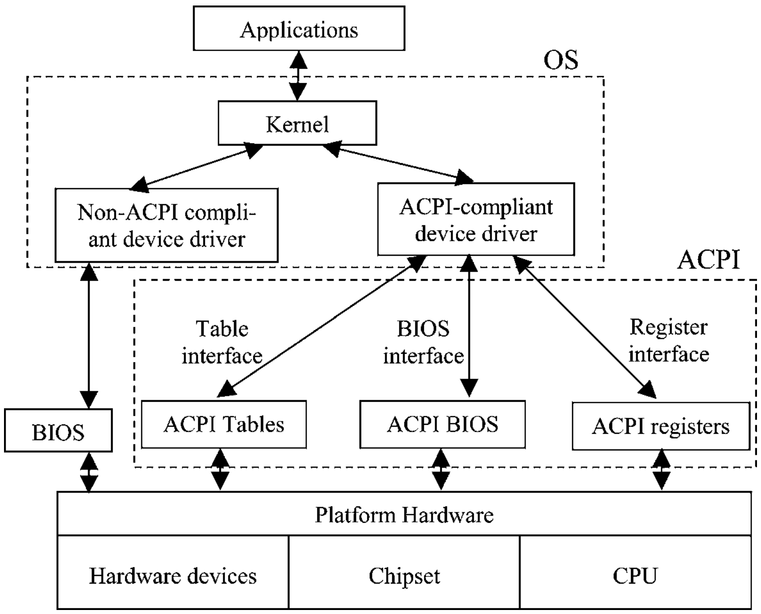


FIG. 7. Interaction among system components with ACPI [2].

software. Figure 7 overviews the interactions among system components in ACPI [2]. The front-end of the ACPI is the ACPI-compliant device driver. It maps kernel requests to ACPI commands and maps ACPI responses to I/O interrupts or kernel signals. Note that the kernel may also interact with non-ACPI-compliant hardware through other device drivers.

The ACPI specification defines five *global* system power states: G0 represents the working state, G1 to G3 represent the sleeping states, and lastly one legacy state for non-ACPI-compliant devices. The specification also refines the sleeping state by defining additional four *sleeping* states (S1–S4) within the state G1 as shown in Table IX. In addition to the global states, ACPI also defines four *device* (D0–D3) and four *processor* states (C0–C3). Different states differ in the power they consume and the time needed for wake up. For example, a deep sleep state, such as S4 state in Table IX, saves more power but takes longer to wake up.

3.2.2.2 Application-Based DPM.

Application-based DPM policies were made possible by the emergence of the ACPI standard state above. These policies move the power manager from the device or hardware level to the application level. The application, which is the source of most requests to the target device, is now in

TABLE IX
ACPI GLOBAL STATES [2]

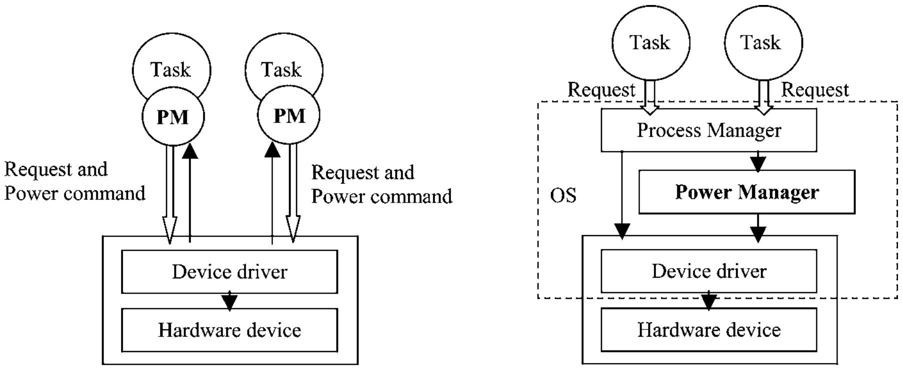
Global states		Description
G3		Mechanical off: no power consumption, system off.
G2		Soft off: full OS reboot needed to restore working state.
G1	*Sleeping states*	Sleeping: system appears to be Off, and will return to working state in an amount of time that increases with the inverse of power consumption.
	S4	Longest wake-up latency and lowest power. All devices are powered off.
	S3	Low wake-up latency. All system contexts are lost except system memory.
	S2	Low wake-up latency. Only CPU and system cache context is lost.
	S1	Lowest wake-up latency. No system context is lost.
G0		Working: system On and fully operational.
Legacy		Legacy: entered when system is non-ACPI compliant.

↑ Less Power More latency

↓ More Power Less latency

charge of commanding the power states of that device. These policies allow application programs to put a device in the fully working state, send it to sleep mode, wake it up, or receive notice about the device power-state changes. For example, Microsoft's *OnNow* [47] and *ACPI4Linux* [1] support power management for ACPI-compliant devices. Figure 8(a) illustrates the application-based DPM scheme.

Alternatively, Lu et al. proposed a software architecture that exports power management mechanisms to the application level through a template [38]. This scheme makes power state decisions based on information about the system parameters such



(a) Application-level power management (b) OS-level power management

FIG. 8. Software-based DPM policies.

as power at each state, transition energy, delay, and future workload. This software architecture differs from the ACPI-based techniques in that the power management is centralized to one application, which makes it safer and more efficient in a single application system.

3.2.2.3 Operating System-Based DPM.

Application-based power management has several drawbacks. First, they require modifications to existing applications, and thus implementing these policies will place additional burden on the programmers. Second, advances in technology constantly change hardware parameters, which make a policy optimized for a certain environment inefficient after a device is replaced. Finally, different programs may set the same device to different power states causing the system become unstable. OS-based DPM techniques use the operating system's knowledge of all the running processes and their interaction with the hardware to optimize energy performance. A number of OS-based DPM schemes have been proposed in the literature [23,35,36]. Figure 8(b) illustrates the implementation of OS-based power management.

Lu et al. proposed *task-based* power management, which uses process IDs at the OS-level to differentiate tasks that make I/O requests [35,36]. This has a major advantage over device-based policies in that it offers a better understanding of device utilization as well its future usage pattern. For example, an OS-based DPM scheme, called *power-oriented process scheduling*, schedules tasks by clustering idle periods to reduce the number of power-state transitions and state-transition overhead [35]. Finally, Gniady et al. proposed to use program counters to predict I/O activities in the operating system [23]. Their *program-counter access predictor* dynamically learns the access patterns of an application using path-based correlation to match a particular sequence of program counters leading to each idle period. This information is then used to predict future occurrences of this idle period and thus optimize power.

3.3 Parallel System-Level DPM

Most of the power related research topics are devoted to uni-processor systems. However, due to the co-operative nature of computation in a parallel computing environment, the most energy-efficient execution for each individual processor may not necessarily lead to the best overall energy-efficient execution. Reducing power consumption not only reduces the operation cost for cooling but also increases reliability, which is often critically important for these high-end systems. This section introduces DPM techniques for parallel systems proposed in the literature. First, hardware-based power optimization techniques such as coordinated DVS [19] and low-power interconnection networks [32,53] in cluster systems are presented in

Section 3.3.1. Second, software-based DPM techniques such as energy-aware synchronization for multiprocessors systems [34] are introduced in Section 3.3.2. This subsection also presents DPM techniques used in server clusters to reduce the energy consumption of the whole cluster by coordinating and distributing the workload among all available nodes [50].

3.3.1 Hardware-Based DPM Techniques

3.3.1.1 Coordinated Dynamic Voltage Scaling (CVS). Elnozahy et al. also proposed to use the DVS scheme discussed in Section 3.1.3 in a cluster system [19]. They presented five such policies for comparison. The first policy, *Independent Voltage Scaling* (*IVS*) simply uses voltage scaled processors, where each node independently manages its own power consumption. The second policy, called *Coordinated Voltage Scaling* (*CVS*), uses DVS in a coordinated manner so that all cluster nodes operate very close to the average frequency setting across the cluster in order to reduce the overall energy cost. This can be achieved by periodically computing the average frequency setting of all active nodes by a centralized monitor and broadcasting it to all the nodes in the cluster. The third policy, called *vary-on/vary-off* (*VOVO*), turns off some nodes so that only the minimum number of nodes required to support the workload are kept alive. The fourth policy, called *Combined Policy*, combines IVS and VOVO, while the fifth policy, called *Coordinated Policy*, uses a combination of CVS and VOVO. According to their evaluation, the last two policies offer the most energy savings. Among the two, the *Coordinated Policy* (CVS-VOVO) saves more energy at the expense of a more complicated implementation.

3.3.1.2 Network Interconnect-Based DPM. One of the most critical power drains in parallel systems is the communication links between nodes, which is an important differentiating factor compared to uni-processor systems. The communication facilities (switches, buses, network cards, etc.) consume a large amount of the power budget of a cluster system [32,53], which is particularly true with the increasing demand for network bandwidth in such systems. Shang et al. proposed to apply DVS to the internetworking links [53]. The intuition is that if network bandwidth could be tuned accurately to follow the link usage, huge power saving can be achieved. A *history-based DVS* policy uses past network traffic in terms of link utilization and receiver input buffer utilization to predict future traffic. It then dynamically adjusts the voltage and frequency of the communication links to minimize the network power consumption while maintaining high performance.

An alternative DPM scheme applied to interconnection links was suggested by Kim et al. [32]. They addressed the potential problem of performance degradation, particularly in low to medium workload situations. This may result in more buffer utilization which also increases the overall leakage energy consumption. Their method called *Dynamic Link Shutdown* (*DLS*) scheme attempts to alleviate the problem based on the fact that a subset of under-utilized links (with utilization under a certain threshold) could be completely shut down assuming another subset of highly used links can be found to provide connectivity in the network.

3.3.2 Software-Based DPM Techniques

3.3.2.1 Barrier Operation-Based DPM.

As discussed in the previous subsection, interconnection links may be the most critical bottleneck in parallel systems with respect to energy consumption as well as computing performance. From the perspective of application software, collective communications such as *barriers* are often considered the most critical bottleneck during the execution of a parallel application [42]. In a conventional multiprocessor system, an early arriving thread stops (typically by spin-waiting) at the barrier and waits for all slower threads to arrive before proceeding with the execution past the barrier. This barrier spin-waiting is highly inefficient since power is wasted performing unproductive computations.

Li et al. proposed *thrifty barrier* [34], where an early arriving thread tries to put its processor into a low power state instead of just spinning. When the last thread arrives, dormant processors are woken up and all the threads proceed past the barrier. However, as discussed earlier in Section 3.2.1, power state transitions should justify the power saving versus the delay time incurred in the process. Therefore, each thread that arrives early should predict the length of the pending stall time and decide whether to transit to low power state or not, and if so, choose the best low power state. At the same time, the wake up time must also be predicted to tradeoff power saving versus performance degradation. To tackle these problems, the thrifty barrier uses the past history of interval time between two consecutive barriers to predict the stall time at the barrier [34]. The main objective is to wake up dormant threads just in time for the proceeding execution, thereby achieving significant energy savings without causing performance degradation.

3.3.2.2 DPM with Load Balancing.

In a cluster system, load balancing is a technique used to evenly distribute the workload over all available nodes in a way that all idle nodes are efficiently utilized. Pinheiro et al. used the concept of *load unbalancing* to reduce power consumption of a cluster system [50]. Unlike load balancing, it concentrates work in fewer nodes while idling others that can be turned off, which will lead to power saving but at the same time may degrade performance. Their algorithm periodically evaluates whether some nodes should be removed from or added to the cluster based on the predicted power consumption and the given total workload imposed on the cluster with different cluster configurations. If nodes are underutilized, some of them will be removed, and if nodes are overused, new nodes should be added. In both cases, the algorithm redistributes the existing workload to the active nodes in the cluster. Significant power reduction was reported with only negligible performance degradation [50].

4. Conclusion

The need for robust power-performance modeling and optimization at all system levels will continue to grow with tomorrow's workload and performance requirements for both low-end and high-end systems. Such models, providing design-time or run-time optimizations, will enable designers to make the right choices in defining the future generation of energy efficient systems.

This chapter discussed different ideas and techniques proposed in the literature with the goal of developing power-aware computer systems. The various methods surveyed were differentiated by design time *power analysis* and run-time *dynamic power management* techniques. Power analysis techniques are mainly based on simulation, sometimes assisted by measurements. These techniques integrate various energy models into existing simulation tools and analyze and profile the power consumption on different levels of a computer system at design time in order to help build power efficient hardware and software systems. On the other hand, dynamic power management techniques are applied during run-time. They monitor the system workload to predict the future computational requirements and try to dynamically adapt the system behavior accordingly.

Successful design and evaluation of power management and optimization techniques are highly dependent on the availability of a broad and accurate set of power analysis tools, which will be crucial in any future study. While the focus should be more on the overall system behavior capturing the interaction between different system components, it is also important to have a better and more accurate understanding of particular hardware components or software programs with respect to power consumption. While power efficient system design is important in low-end portable systems, this is also true in high-end parallel and clustered systems. This is because high power consumption raises temperature and thus deteriorates performance and reliability. Therefore, a holistic approach that considers all the components in such systems will need to be further investigated.

References

[1] "ACPI4Linux", http://phobos.fs.tum.de/acpi/.
[2] "Advanced Configuration and Power Interface (ACPI)", http://www.acpi.info.

[3] “Advanced Power Management (APM)”, http://www.microsoft.com/whdc/archive/amp_12.mspx, Microsoft Corporation.

[4] Azevedo A., Cornea R., Issenin I., Gupta R., Dutt N., Nicolau A., Veidenbaum A., “Architectural and compiler strategies for dynamic power management in the COPPER project”, in: *IWIA 2001 International Workshop on Innovative Architecture*, 2001.

[5] Azevedo A., Cornea R., Issenin I., Gupta R., Dutt N., Nicolau A., Veidenbaum A., “Profile-based dynamic voltage scheduling using program checkpoints”, in: *Design, Automation and Test in Europe Conference and Exhibition (DATE’02)*, 2002.

[6] Bavier A., Montz A., Peterson L., “Predicting MPEG execution times”, in: *Proceedings of SIGMETRICS ’98/PERFORMANCE ’98*, 1998.

[7] Benini L., Bogliolo A., De Micheli G., “A survey of design techniques for system-level dynamic power management”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **8** (3) (June 2000).

[8] Benini L., Bogliolo A., Paleologo G.A., De Micheli G., “Policy optimization for dynamic power management”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **18** (6) (June 1999) 813–833.

[9] Brooks D., Bose P., Schuster S., Jacobson H., Kudva P., Buyuktosunoglu A., Wellman J., Zyuban V., Gupta M., Cook P., “Power aware microarchitecture: design and modeling challenges for next-generation microprocessors”, *IEEE Micro* (December 2000) 26–44.

[10] Brooks D., Tiwari V., Martonosi M., “Wattch: a framework for architectural-level power analysis and optimizations”, in: *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, June 2000.

[11] Burger D., Austin T., “The SimpleScalar tool set, version 2”, Tech. Report No. 1342, Computer Sciences Dept., Univ. of Wisconsin, June 1997.

[12] Chang F., Farkas K., Ranganathan P., “Energy-driven statistical profiling: detecting software hotspots”, in: *Proceedings of the Workshop on Power Aware Computing Systems*, February 2002.

[13] Chedid W., Yu C., “Dynamic voltage scaling techniques for power-aware MPEG decoding”, Master’s Thesis, ECE Dept., Cleveland State University, December 2003.

[14] Chung E.-Y., Benini L., Bogliolo A., Lu Y.-H., De Micheli G., “Dynamic power management for nonstationary service requests”, *IEEE Transactions on Computers* **51** (11) (November 2002) 345–1361.

[15] Chung E.-Y., Benini L., De Micheli G., “Dynamic power management using adaptive learning tree”, in: *Proceedings of the International Conference on Computer-Aided Design*, November 1999, pp. 274–279.

[16] Cignetti T., Komarov K., Ellis C., “Energy estimation tools for the Palm^{TM}”, in: *Proceedings of the ACM MSWiM’2000: Modeling, Analysis and Simulation of Wireless and Mobile Systems*, August 2000.

[17] Douglis F., Krishnan P., Bershad B., “Adaptive disk spin-down policies for mobile computers”, in: *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, 1995, pp. 381–413.

[18] “Earth Simulator”, http://www.es.jamstec.go.jp/esc/eng/index.html.

[19] Elnozahy E.N., Kistler M., Rajamony R., “Energy-efficient server clusters”, in: *Proceedings of the Second Workshop on Power Aware Computing Systems*, February 2002.

[20] Flinn J., Satyanarayanan M., “PowerScope: a tool for profiling the energy usage of mobile applications”, in: *Proceedings of the Workshop on Mobile Computing Systems and Applications (WMCSA)*, February 1999.
[21] Gary S., Ippolito P., Gerosa G., Dietz C., Eno J., Sanchez H., “PowerPC 603, a microprocessor for portable computers”, *IEEE Design & Test of Computers* **11** (4) (October 1994) 14–23.
[22] Ghazaleh N.A., Mossé D., Childers B., Melhem R., Craven M., “Collaborative operating system and compiler power management for real-time applications”, in: *9th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003.
[23] Gniady C., Hu Y.C., Lu Y.-H., “Program counter based techniques for dynamic power management”, in: *10th International Symposium on High Performance Computer Architecture (HPCA’04)*, February 2004.
[24] Golding R., Bosch P., Staelin C., Sullivan T., Wilkes J., “Idleness is not sloth”, in: *Proceedings of the USENIX Winter Conference*, 1995, pp. 201–212.
[25] Govil K., Chan E., Wasserman H., “Comparing algorithms for dynamic speed-setting of a low-power CPU”, *MobiCom* (1995).
[26] Gowan M., Biro L., Jackson D., “Power considerations in the design of the Alpha 21264 microprocessor”, in: *ACM Design Automation Conference*, June 1998, pp. 726–731.
[27] Gurumurthi S., Sivasbramaniam A., Irwin M.J., Vijaykrishnan N., Kandemir M., “Using complete machine simulation for software power estimation: the SoftWatt approach”, in: *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA-8)*, February 2002.
[28] Hwang C.-H., Wu A., “A predictive system shutdown method for energy saving of event-driven computation”, in: *International Conference on Computer-Aided Design,* November 1997, pp. 28–32.
[29] Henkel J., Lekatsas H., “A^{2}BC: adaptive address bus coding for low power deep sub-micron designs”, in: *ACM Design Automation Conference*, 2001.
[30] “IRIX OS”, http://www.sgi.com/developers/technology/irix/.
[31] Kandemir M., Vijaykrishnan N., Irwin M., “Compiler optimizations for low power systems”, in: *Workshop on Power Aware Computing Systems,* 2002, pp. 191–210.
[32] Kim E.J., Yum K.H., Link G.M., Das C.R., Vijaykrishnan N., Kandemir M., Irwin M.J., “Energy optimization techniques in cluster interconnects”, in: *International Symposium on Low Power Electronics and Design (ISLPED’03)*, August 2003.
[33] Klaiber A., “The technology behind Crusoe processors”, Transmeta Corporation, http://www.transmeta.com/pdfs/paper_aklaiber_19jan00.pdf, January 2000.
[34] Li J., J.F. Martíne, Huang M.C., “The thrifty barrier: energy-aware synchronization in shared-memory multiprocessors”, in: *10th International Symposium on High Performance Computer Architecture (HPCA’04)*, February 2004.
[35] Lu Y.-H., Benini L., De Micheli G., “Low-power task scheduling for multiple devices”, in: *International Workshop on Hardware/Software Codesign*, May 2000.
[36] Lu Y.-H., Benini L., De Micheli G., “Operating-system directed power reduction”, in: *International Symposium on Low Power Electronics and Design*, July 2000.
[37] Lu Y.-H., De Micheli G., “Comparing system-level power management policies”, *IEEE Design & Test of Computers* **18** (2) (March 2001) 10–19.

[38] Lu Y.-H., Simunic T., De Micheli G., "Software controlled power management", in: *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES99)*, May 1999, pp. 157–161.

[39] Mead C., Conway L., *Introduction to VLSI Systems*, Addison–Wesley, Reading, MA, 1980.

[40] "Millywatt", http://www.cs.duke.edu/ari/millywatt/.

[41] "Mobile Intel Pentium III processor in BGA2 and Micro-PGA2 packages datasheet", Intel Corporation.

[42] Moh S., Yu C., Lee B., Youn H.Y., Han D., Lee D., "4-ary tree-based barrier synchronization for 2-D meshes without nonmember involvement", *IEEE Transactions on Computers* **50** (8) (August 2001) 811–823.

[43] "Motorola Dragonball", http://www.motorola.com/dragonball/.

[44] Moudgill M., Bose P., Moreno J., "Validation of Turandot, a fast processor model for microarchitecture exploration", in: *Proceedings of IEEE International Performance, Computing and Communication Conference*, 1999, pp. 451–457.

[45] "MPEG-2: the basics of how it works", Hewlett Packard Lab.

[46] Namgoong W., Yu M., Meng T., "A high-efficiency variable-voltage CMOS dynamic DC-DC switching regulator", in: *IEEE International Solid-State Circuits Conference*, 1997, pp. 380–381.

[47] "OnNow", http://www.microsoft.com/hwdev/onnow/.

[48] "Palm OS emulator", http://www.palmos.com/dev/tools/emulator/.

[49] Patino O.A., Jimenez M., "Instruction level power profile for the powerPC microprocessor", in: *Computing Research Conference 2003*, University of Puerto Rico-Mayagüez, April 2003, pp. 120–123.

[50] Pinheiro E., Bianchini R., Carrera E.V., Heath T., "Load balancing and unbalancing for power and performance in cluster-based systems", Technical Report DCS-TR-440, Dept. Computer Science, Rutgers University, May 2001.

[51] Qiu Q., Pedram M., "Dynamic power management based on continuous-time Markov decision processes", in: *Proceedings of the Design Automation Conference*, June 1999, pp. 555–561.

[52] Quan G., Hu X., "Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors", in: *Design Automation Conference*, 2001.

[53] Shang L., Peh L.-S., Jha N.K., "Dynamic voltage scaling with links for power optimization of interconnection networks", in: *9th International Symposium on High-Performance Computer Architecture (HPCA'03)*, February 2003.

[54] Shin D., Kim J., "Intra-task voltage scheduling for low-energy hard real-time applications", *IEEE Design & Test of Computers* **18** (2) (March 2001) 20–30.

[55] "SimOS", http://simos.stanford.edu/.

[56] "Simulink", http://www.mathworks.com/products/simulink/, The MathWorks.

[57] Simunic T., Benini L., Glynn P., De Micheli G., "Dynamic power management for portable systems", in: *Proceedings of the International Conference on Mobile Computing and Networking*, 2000, pp. 11–19.

[58] Son D., Yu C., Kim H., "Dynamic voltage scaling on MPEG decoding", in: *International Conference on Parallel and Distributed Systems (ICPADS)*, 2001.

[59] "SPEC JVM98", http://www.specbench.org/osg/jvm98/.

[60] Srivastava M.B., Chandrakasan A.P., Brodersen R.W., "Predictive system shutdown and other architecture techniques for energy efficient programmable computation", *IEEE Transaction on VLSI Systems* **4** (1) (March 1996) 42–55.

[61] Stan M., Burleson W., "Bus-invert coding for low-power I/O", *IEEE Transaction on VLSI* **3** (1) (1995) 49–58.

[62] "The Standard Performance Evaluation Corporation (SPEC)", http://www.spec.org/.

[63] Su C.L., Tsui C.Y., Despain A.M., "Low power architecture design and compilation techniques for high-performance processors", in: *COMPCON'94*, 1994.

[64] Suzuki K., et al., "A 300 MIPS/W RISC core processor with variable supply-voltage scheme in variable threshold-voltage CMOS", in: *IEEE Custom Integrated Circuits Conference*, 1997, pp. 587–590.

[65] Tiwari V., Malik S., Wolfe A., Lee M.T., "Instruction level power analysis and optimization of software", *Journal of VLSI Signal Processing* **13** (2–3) (August 1996).

[66] Tomiyama H., Ishihara T., Inoue A., Yasuura H., "Instruction scheduling for power reduction in processor-based system design", in: *Proceedings of Design Automation and Test in Europe (DATE98)*, 1998, pp. 855–860.

[67] Wang H.-S., Zhu X.-P., Peh L.-S., Malik S., "Orion: a power-performance simulator for interconnection networks", in: *Proceedings of MICRO 35*, November 2002.

[68] Weiser M., Welch B., Demers A., Shenker S., "Scheduling for reduced CPU energy", in: *USENIX Symposium on Operating Systems Design and Implementation, (OSDI)*, November 1994, pp. 13–23.

[69] Wen V., Whitney M., Patel Y., Kubiatowicz J.D., "Exploiting prediction to reduce power on buses", in: *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, 2004.

[70] Ye W., Vijaykrishnan N., Kandemir M., Irwin M.J., "The design and use of SimplePower: a cycle-accurate energy estimation tool", in: *Proceedings of the Design Automation Conference*, June 2000.

[71] Ye T.T., De Micheli G., Benini L., "Analysis of power consumption on switch fabrics in network routers", in: *Proceedings of the 39th Conference on Design Automation*, June 2002.

Flexible and Adaptive Services in Pervasive Computing

BYUNG Y. SUNG, MOHAN KUMAR, AND BEHROOZ SHIRAZI

Department of Computer Science and Engineering
University of Texas at Arlington
Arlington, TX 76019-0019
USA
sung@cse.uta.edu
kumar@cse.uta.edu
shirazi@cse.uta.edu

Abstract
Availability of services to users and applications is a critical issue in mobile and pervasive computing environments replete with hundreds and thousands of smart devices. In such environments, devices such as sensors, embedded processors, personal servers, cell phones and PDAs have limited resources in terms of CPU, memory, communication bandwidth, and battery energy. Therefore, it is extremely important as well as challenging to conserve device resources while providing software services in a timely fashion for a variety of application scenarios. The solution lies in the deployment of software services in pervasive computing environments that can be used by mobile and wireless clients on resource-constrained devices. In this chapter we discuss issues in providing services in pervasive environments and give a summary of related work. Furthermore, we propose a service provisioning framework that is not only *flexible* in defining and generating services, but also *adaptive* to the environment. The service provisioning mechanism based on the community computing framework captures the essence of devices and services around users, and facilitates the creation and composition of flexible and adaptive services. We develop an elegant Petri net based modeling of the framework and present several case studies to demonstrate and validate the model.

ISSN: 0065-2458/DOI 10.1016/S0065-2458(04)63005-1

1. Introduction

The pervasive computing paradigm is leading us to the next generation of computing environment in which hundreds and thousands of smart devices exist and are expected to provide a variety of services. In order to utilize these services efficiently and effectively, it is imperative for users and applications to be aware of the types of services. Due to the vast number of devices, an effective service discovery mechanism is essential to aid users/applications to become aware of services that are available [29]. Service provisioning framework in pervasive computing environment aids users in locating and utilizing available services, and composing new services using existing ones.

Many research projects aim at improving users' experience in pervasive computing environment by designing and implementing a new breed of services [23,32]. These research efforts will expand the spectrum of applications and services available to users in pervasive environments. Research has progressed in the areas of resource discovery [1,9,11], middleware support [24], and user interface [23]. Currently, to the best of our knowledge, existing mechanisms are not equipped to create and provide composite services to users on a need-basis and at the same time enhance service availability.

The proposed service provisioning mechanism is built on top of the PICO (Pervasive Information Community Organization) framework [16,17] and [28]. We use the following abstractions to model services that are provided by devices in order to better manage services and devices available to users:

- High-level abstraction of services around users: Users are surrounded by pervasive computing devices that possess implicit/explicit services. Any service provisioning framework that utilizes existing services should provide a methodology to abstract services, such that applications or even human users in the environment can recognize and utilize services effectively.
- High-level abstraction of relationship between services: A single service alone may be inadequate to fulfill a user's (or application's) needs, leading to the consolidation of a number of necessary services to meet the requirements. In this case, the abstraction of services should provide a glimpse of how new services can be composed. The abstraction should include basic compatibility information to assess a service composed of two services—determine whether the consolidated service is syntactically correct and semantically compatible.

The above abstractions are crucial elements in our service provisioning framework to capture all the necessary aspects. The reference software platform also utilizes these abstractions to create software entities that represent services on behalf of users.

Consolidated services from existing simple ones on devices can be composed in various ways as described in Section 6. There are basically two levels of service compositions in our framework using the abstractions described earlier—compose services using the functional units and create new services using the services distributed throughout multiple devices. A functional unit of service has a specific purpose and exists on a single device. Each level of service provisioning poses a unique set of challenges and issues. These issues are addressed and an attempt is made to resolve them in Section 6. Currently, our service provisioning scheme is static, but we look at this as a step towards creating dynamic just-in-time services.

Like other pervasive computing software systems, our reference software implementation of the service provisioning framework should resolve (address) issues of systems and network heterogeneity, and resource management. We adopt the middleware approach as it is proven to be a better and reliable way to address the above issues. In addition, routing event messages are critical to our system since all the inter-operations between services involve exchange of events. We address the event routing issue by incorporating the two abstractions to the event handler described in Section 6.4. Even though we partially deal with issues of profiling and context awareness in our model, they will be addressed in future work.

The remainder of the chapter is organized as follows. Section 2 gives an overview of pervasive computing and discusses the challenges. Section 3 investigates role of services in pervasive computing and discusses existing service provisioning schemes. Related work carried out by researchers in the area of services is discussed in Section 4. Section 5 presents a reference software implementation for the framework. Section 6 proposes the services provisioning model we are investigating. Section 7 presents Petri net based modeling of services and simulation results. Section 8 extends the model by using Colored Petri Nets to capture environment changes. Section 9 concludes the chapter and discusses possible future directions.

2. Overview of Pervasive Computing

Recent advances in hardware, wireless and sensor networks, software agents, and middleware technologies have been mainly responsible for the emergence of pervasive computing as an exciting area with applications in many areas. It is imperative to acknowledge that pervasive computing is not just an extension of wireless networking and mobile computing. Pervasive computing encompasses networks, mobile and distributed computing, and more—agent technologies, middleware, situation-aware computing etc.

Pervasive computing is about providing ‘*where you want, when you want, what you want and how you want*’ services to users, applications and devices. Pervasive computing incorporates research findings from many areas of computer science to meet the challenges posed by a myriad of applications. Currently, we do have the necessary hardware and software infrastructure to foster the growth of pervasive computing. What is necessary is the ‘glue’ to put together existing entities in order to provide meaningful and unobtrusive services to applications. There have been many outstanding papers in recent years, highlighting the challenges of pervasive computing [26,2]. These issues and challenges can be listed as—invisibility, interoperability, heterogeneity, proactivity, mobility, intelligence and security.

The technological advances in the types of devices and their functionality, size, shape and appearance is the driving force for pervasive computing. Heterogeneous devices and hardware can be employed to meet the challenging demands of the applications. Software services at the middleware level can provide proactive, transparent, and secure environments for seamless interaction.

Technological and research advances in several areas are mainly responsible for the advances in pervasive computing. We identify the following areas: Devices and network technologies; Enabling technologies such as software agents and active networks; wireless and mobile computing; and smart environments.

Several new embedded devices and sensors are being developed in industry and research laboratories. The design and architecture of the Berkeley sensor motes and the TinyOS operating system are very good examples of the type of devices that can be used in applications. TinyOS is an event based operating environment designed for use with embedded networked sensors. The challenge here is to design devices that are tiny (disappear into the environment), consume little or no power (perhaps powered by ambient pressure, light, or temperature); communicate seamlessly with other devices, humans, and services through a simple all purpose communication protocol.

Understanding of device and network technologies is important to create a seemingly uniform computing space in a heterogeneous environment. The challenge is to overcome Internet's end-to-end architecture and at the same time *network* devices, services and users.

The computing world is replete with numerous types of devices, operating systems and networks. Cooperation and collaboration among various devices and software entities is necessary for pervasive computing. At the same time, the overheads introduced by the adaptation software should be minimal and scalable. While it is almost unthinkable (in today's environment) to have homogeneous devices and software, it is however, possible to build software bridges across various entities to ensure interoperability. Several questions arise: how many such bridges should we create? what about the additional costs introduced by the overheads? The Oxygen project, envisions the use of uniform hardware devices and network devices to enable smooth interoperability [2]. The limitations of low resource hardware can be overcome by exploiting the concepts of agents and services. The challenge is to develop effective and flexible middleware tools that mask the uneven conditions and, create portable and lightweight application software.

Traditionally, agents have been employed to work on behalf of users, devices and applications [6]. In addition, agents can be effectively used to provide transparent interfaces between disparate entities in the environment and thus enhance invisibility. Agent interaction and collaboration is an integral part of pervasive computing as agents can overcome the limitations of hundreds and thousands of resource limited devices.

Service discovery is described as the process of discovering software processes/ agents, hardware devices and services. Discovering services in mobile environments has been addressed by many researchers, but service discovery in pervasive computing is still in its infancy. The role of service discovery in pervasive computing is to provide environment-awareness to devices and device-awareness to the environment. Service provisioning, advertisement and service discovery are the important components of this module. Existing service discovery mechanisms include Jini, Salutation and the International Naming System (INS) [10].

The development of computing tools hitherto has been in general, ‘reactive’ or ‘interactive’. On the other hand, ‘human in the loop’ has its limits, as the number of networked computers will surpass the number of users within a few years. Users of pervasive computing applications would want to receive ‘what I want’ kind of information and services in a transparent fashion. Tennenhouse, in his thought provoking chapter [29] envisions that a majority of computing devices in future will be proactive. Proactivity can be provided by the effective use of overlay networks. Active networks can be used to enhance network infrastructure for pervasive computing, ensure network management on a just-in-time basis and provide privacy and trust. The challenges include:

(i) how to leverage the research work in the areas of situation-aware computing, device-, user-, application profiling, and software agents to enhance proactivity in existing computing devices?, and
(ii) How to exploit active network technology to overcome the end-to-end to limitations of the Internet to provide just-in-time services?

Mobile computing devices have limited resources, are likely to be disconnected, and are required to react (transparently) to frequent changes in the environment. Mobile users desire *anytime anywhere* access to information while on the move. Typically, a wireless interface is required for communication among the mobile devices. The wireless interface can be a wireless LAN, a cellular network, a satellite network or a combination. Techniques developed for routing, multicasting, caching, and data access in mobile environments should be extended to pervasive environments. A pervasive environment comprises numerous invisible devices, anonymous users, and pervasive services. Development of effective middleware tools to mask the heterogeneous wireless networks and mobility effects is a challenge.

Provisioning uniform services regardless of location is a vital component of mobile and pervasive computing paradigms The challenge is to provide location-aware services in an adaptive fashion, in a form that is most appropriate to the location as well as the situation.

To create smart spaces we need an intelligent, situation-aware system augmented by middleware support. The challenge is to incorporate middleware tools into smart environments. In a crisis management situation, the remote sensing system should have abilities to infer the user’s intention and current situation to inform of the possible danger prior to the occurrence of the crisis. For example, the Sentient computing environment [12], is localized and very limited in terms of functionality. With the middleware services, the same environment can be used for pervasive health care as envisioned in [27]. Smart home projects are being carried out in many universities and research laboratories. Smart homes, allow the occupants to interact with home equipment and appliances by exploiting knowledge gained through user behavior. Current smart homes, for example are not advanced enough to allow the user to take the (virtual) home with her. Several methods for knowledge representation in intelligent spaces have been investigated. Provisioning proactive, context-aware or situation oriented response from the infrastructure to the users' needs is a challenge. Several promising smart environments have been developed [23].

Current notion is that security is the price we pay for pervasiveness. It is true, that security is compromised significantly due to the needs of proactive system performance. Security is a major challenge that needs to be addressed in order to build confidence among users. But, security and privacy are also dependent on the situations. For example, in a pervasive health care system, if the patient is in an emergency condition, swift response is critical, perhaps at the expense of privacy. The challenge is to strike a balance: between proactivity and privacy; between security and efficiency/cost; between security/privacy and quality. Investigations in mobile and ad hoc network security have progressed significantly in recent years. Further more, inevitably pervasive environments comprise software agents and services. Ensuring secure transmission and execution of mobile code on remote devices as well as protection of devices and software from mobile code is a challenge.

3. Services in Pervasive Computing

Consider the following scenario. Dr. John Smith is a medical surgeon and an experimental biologist. On his walk to the cafeteria, Dr. Smith makes notes on his handheld device about a patient he just visited. While in the cafeteria, he also receives messages and vital information from other doctors, patients, students and nurses. He frequently provides his identity and other authorization codes to a security service prior to receiving/transmitting data. On some days, he consults, remotely with his patients—listens to sounds, examines images and data provided by remote consultation machines, patients and nurses. He also queries the patient records systems (EMR) for latest patient histories, analyzes images, sounds and textual data, and updates records accordingly. Dr. Smith receives results of experimental work from his research laboratory and performs statistical analysis. There are occasions when he is required to do a quick review of his finances. He does all this on his handheld terminal such as a cell phone equipped with personal digital assistant (PDA) capabilities as shown in Figure 1. But oft-times he is forced to wait and wait for an eternity as his resource-poor PDA/terminal seems to take forever to run simple applications. It is not difficult to imagine a similar scenario involving a realtor selling properties, an engineer on the manufacturing floor of an Industry, or a shopper at the mall.

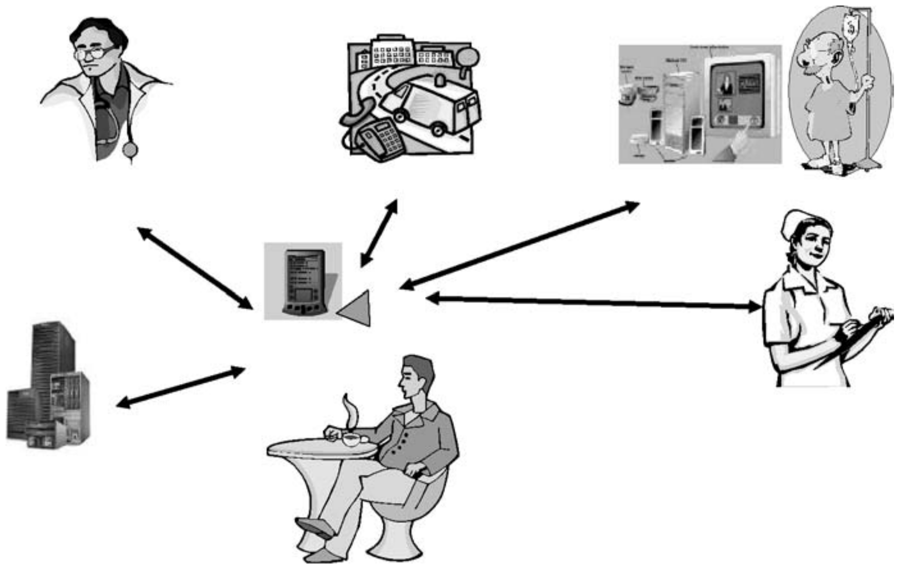


FIG. 1. Services needed in a typical application scenario.

In the above scenario, the PDA is required to support the following functionalities so that the Doctor can carry out his important tasks:

(a) Situation-aware information acquisition and processing—pick and choose from a large number of data sources, filter information, adapt incoming information to the Doctor's device;
(b) Information creation—transform the Doctor's prescriptions, advice, etc. to a form acceptable to the tablet PC carried by a Nurse on the move;
(c) Data compression and decompression;
(d) Database transactions—with the EMR or other hospital data repositories;
(e) Virus detection and removal;
(f) Security and privacy—fire walling and authentication;
(g) situation-aware capability—for example, tell the doctor that there is too much of (signal) interference inside the cafeteria.

Now, regardless of the CPU power and memory capacity of the PDA, it is almost impossible to equip the PDA with all the above software services and perhaps more.

Existing research is mainly focused on how information/content is delivered or received at the mobile users' terminal. But mobile users will need to process information, make transactions, choose between multiple sources of information, ensure secure access and avail various utility services via their handheld devices. In addition, mobile users will be required to perform other supporting tasks that need additional resources in order to execute vital tasks such as those mentioned above. For certain application tasks, communications, collaborations and interactions between different devices and execution of resource-intensive jobs may be required. Some tasks may require high performance computing resources. Mobile users may desire the flexibility and convenience of carrying lightweight low battery consuming handheld devices, and yet having the option to access resource-consuming *state of the art* computing and communication services. The solution lies in the design and development of effective *middleware services* that execute on resource-rich platforms such as the wired network infrastructure and clusters of computers or Grids, thus effectively masking the limitations of resource-poor mobile wireless devices.

The gap between computing and communication devices is decreasing rapidly—in the near future every device will be required to perform both types of services. Network processors, active networks and ad hoc networks subscribe to this trend. Proliferation of wireless communications would require all devices to serve as communication routers to switch packets among communicating and computing devices. In other words, a computing device or a network's resources can be utilized for various purposes. On the other hand, due to the tremendous progress in such areas as context-aware computing, proactive computing and intuitive computing, there is a steady progress towards integration of physical and cyber spaces. A large proportion of our daily lives are spent in analyzing various kinds of information (often excessive and useless) on the Web, pervasive computing devices and sensors, P2P applications, and different kinds of smart environments. There is a growing need to acquire *only* 'what is necessary' kind of information and services at the user's device. Future devices will be geared to meet the individual needs of users while at the same time allowing them to access a variety of services in a transparent manner, regardless of what devices, technology or interfaces they use.

4. Related Work

Service Composition is the process of combining a number of existing services to provide a richer, grander service, assembled to meet the needs of the situation. The concept of service composition is not new. It is an evolved form from object oriented programming and component based systems, such as OMG's CORBA [22], and Microsoft's COM/DCOM [20]. These systems put emphasis mainly in software reuse. There exist several service composition schemes for the web. The purpose of these schemes is to provide a way to integrate services that exist on various web sites. The component based systems are static in that service compositions are carried out at the design phase, and interfaces of services have well-defined syntax and semantics. Some of web-based service compositions are carried out at runtime, but most of them are centralized approaches and the base services are not mobile. The emphasis of service composition in pervasive computing is to create new and more suitable services from base services that are available in user's environment. Further the availability of such base services is temporal, as the network topology in typical pervasive environments changes dynamically. Thus the traditional approaches are inappropriate.

In any service composition framework, there are three components:

(1) service description language,
(2) service discovery mechanism, and
(3) service composition framework.

Services need to be described in such a way that others that want to utilize the service can interpret what the services are about. This has two aspects—syntax, and semantics. Syntax can be unified using some standard, but semantics is much more involved. Several service description languages have been developed. Web Service Development Language (WSDL) [30], Web Service Flow Language (WSFL) [18] and DARPA Agent Markup Language for service (DAML-S) [5] are examples of the description languages. The WSDL by W3C is an XML format for describing network service as a set of endpoints operating on document-oriented or procedure-oriented messages. The DAML project by DARPA and the W3C focuses on standardizing web ontology language (OWL) [5] as the language to use to describe information available on any data source, in order that the information may be understood and used by any class of computers, without human intervention.

Service Discovery Protocols (SDPs) are carrying out the tasks of matching necessary service and finding service in an environment. Service discovery protocols provide a mechanism for dynamically discovering available services in a network and providing the necessary information to:

(1) search and browse for services;
(2) choose the right services (with desired characteristics), and
(3) utilize the services.

A variety of service discovery protocols are currently under development. The most well known so far are:

(1) Service Location Protocol (SLP) [9], developed by the IETF;
(2) Jini [1], which is Sun's Java-based approach to service discovery;
(3) Salutation from the Salutation consortium;
(4) Microsoft's UPnP [10] and the

(5) Bluetooth Service Discovery Protocol (SDP) [10].

The protocols mentioned above, however, lack in providing a framework of modeling services. Most of them are designed to support Internet scale applications, not considering network limitations of pervasive computing such as localized scalability issues and network heterogeneity [26]. Detailed discussion of these protocols can be found in the following section.

Existing service composition systems maintain service process schemata that define a planned execution order of a composed service. Process schemata are mostly defined offline. The problem of splitting a task into sub-tasks is complex and goes into the domain of planning in AI [8]. There is an effort to build a service composition tool that uses natural language interface that allows users to compose new services on a need basis. A composition engine, either centralized or distributed, enacts service processes defined in the schemata with the help of service discovery protocols. Existing service composition systems broadly address the problems associated with composing various services that are available over the fixed network infrastructure. They primarily rely on a centralized composition engine to carry out discovery, integration and composition of web-based e-services. Anamika project [4] addresses issues related service composition in ad hoc network environments. It relies on multiple decentralized composition engine.

In the following sections, we first look at services designed for pervasive computing. Existing service discovery protocols are discussed. Lastly, various service composition mechanisms are introduced, compared and contrasted.

Intel's personal server [32] takes advantage of increasing capacity of portable storage devices and technologies for localized communication. A device that Intel is prototyping is Portable Media Player, aimed at enabling users accessing dynamic media anytime, anywhere. IBM's Steerable interface intends to create a new class of interactive interface. The interface follows users' location changes and projects a GUI interface on the surface of ordinary objects that exist anywhere in pervasive computing environment. The interface has the potential to offer a radically new and powerful style of interaction in intelligent pervasive computing environment. These efforts to create pervasive services will expand the spectrum of applications and services available to users in pervasive computing environment.

4.1 Service Discovery Protocols

There are several research projects and commercially available service discovery protocols (SDPs) intended to be used in wired networks. Examples of service discovery protocols are SLP, Jini, UPnp, and Salutation. For wireless networks, Bluetooth's service discovery protocol is an example. Not many service discovery protocols exist for pervasive computing. The SDP protocols from IBM Zurich [11] is targeted for pervasive computing environment.

Service Location Protocol (*SLP*) is a language-independent protocol for resource discovery on IP networks. The base of the SLP discovery mechanism lies on pre-defined service attributes, which can be applicable to both software and hardware services. The architecture consists of the three types of agents: User Agent (UA), Service Agent (SA) and Discovery Agent (DA). The UAs are responsible for the discovery of available DAs, and acquiring service handles on behalf of end-user applications that request for services. The SAs are responsible for advertising the service handles to DAs. Lastly, the DAs are responsible for collecting service handles and maintaining the directory of advertised services. SLP also works without the presence of DAs. In the latter case, the UA and the SA follow a multicast message-passing mechanism to exchange service related information.

Jini is a Java based distributed service discovery architecture developed by Sun Microsystems. Jini builds on top of Java, object serialization, and RMI to enable objects to move around the network from virtual machine to virtual machine and extend the benefits of object-oriented programming to the network. Instead of requiring device vendors to agree on the network protocols through which their devices can interact, Jini enables devices to talk to each other through interfaces to objects. The central component of Jini is the Jini Lookup Service (JLS). JLS maintain the dynamic information about the available services in a group of devices called the Jini federation. Service provisioning entities require to discover one or more JLSs by using either a known address or by using a Jini multicast discovery mechanism. After discovering a JLS, the service registers itself by uploading its service proxy, and it periodically refreshes its registration at the JLS. Similarly, each client that utilizes services in Jini is required to discover a JLS before it queries a JLS. Once a client locates a JLS, it can download the matching service and invoke various methods by contacting the original service. A client should know the precise name of the Java class representing the service a priori. This gives Jini inflexibility in term of matching a service.

UPnP extends the original Microsoft Plug and Play peripheral model to support service discovery provided by network devices from numerous vendors. UPnP works and defines standards primarily at the lower-layer network protocol suites. Thus devices can implement these standards independent of language and platforms. UPnP uses the Simple Service Discovery Protocol (SSDP) for discovery of services over IP networks, which can operate with or without a lookup service in the network. In addition, the SSDP operates on the top of the existing open standard protocols utilizing HTTP over both unicast (HTTPU) and multicast UDP (HTTPMU). When a new service wants to join the network, it transmits an announcement to indicate its presence. If a lookup service is present, it can record such advertisement to be subsequently used to satisfy client's service discovery requests. Additionally, each service on the network may also observe these advertisements. Lastly, when a client wants to discover a service, it can either contact the service directly through the URL that is stored within the service advertisement, or it can send out a multicast query message, which can be answered by either the directory service or directly by the service.

Salutation is an open standard service discovery and session management protocol that is independent of communication types, operating systems, and platforms. The goal of Salutation is to solve the problem of service discovery and utilization among a broad set of appliances and equipments in a wide area or mobile environment. The architecture provides applications, services and defines a standard method for describing a advertising their capabilities, as well as locating and determining other services and their capabilities. In addition, the Salutation architecture defines an entity called the Salutation Lookup Manager (SLM) that functions as a service broker for services in the network. The SLM can classify services based on their meaningful functionalities, called Functional Units (FU), and the SLM can be discovered by both a unicast and a broadcast method. The services are discovered by the SLM based on a comparison of the required service types with the service types stored in the SLM directory. Finally, the service discovery process can be performed across multiple Salutation managers, where one SLM is a client to another SLM.

The *Salutation-lite* standard prototypes the required architectural changes required by the Salutation technology to support multiple operating system and small form-factor devices. It provides a means to determine the operating system, processor type, device class, amount of free memory, display capabilities and other characteristics of a hand-held device. The message exchange protocols are tailored to reduce the quantity of data exchanged. By limiting the functions of the Service Discovery, the footprint of the implementation has been reduced significantly.

Bluetooth has a specification for a very simple service discovery mechanism in peer-to-peer type networks. The architecture does not have a centralized directory for storing information about services. The information about a service is stored in a local Service Discovery Server. The services are advertised using unique 128 bit identifiers. The attributes of a service also have unique identifiers to distinguish themselves from attributes of another service. The client who wants to find out the existence of a particular service has to know the unique identifier corresponding to that service class. The Bluetooth Service Discovery server does not specify a means to access the service. The server only replies whether a service is available on the queried platform.

Our service provisioning framework provides two types of operations. One is to serve user with the raw service provided by servicing entity, and the other is to provide collaborative services that are composed from existing services based on users' high-level requirements from existing services. It is possible to employ service discovery protocols (SDPs) to find and utilize services, since SDPs provide mechanisms to store service descriptions and to search services based on description user provided. SDPs do not intend to provide a framework to model services and most such protocols assume that services are defined a priori. SDPs do not specify compatibility among services; consequently they are incapable of composing services from already specified services. SDPs are not designed solely for pervasive computing, while pervasive computing environment exposes different characteristics.

4.2 Service Composition Frameworks

In this section, we look at existing service composition frameworks. The eFlow [3] is a work flow based service composition framework and it is targeted for use in eBusiness. Task-Driven Computing from CMU lets users interact with their computing environments in terms of high level tasks and to free them from low level configuration activities. Anamika [4] is a project at UMBC to create a reactive service composition architecture that is distributed, de-centralized and fault-tolerant for reactive service composition in pervasive environments.

The *eFlow* presented in [3] aims at creating the opportunity for providing value-added, integrated services, which are delivered by composing existing e-services. The eFlow is a system that supports the specification, enactment, and management of composite e-services. Composite services are modeled as business processes, enacted by a service process engine. The eFlow provides a number of features that support service process specification and management, including a powerful yet simple service composition language, events and exception handling, ACID service-level transactions, security management, and monitoring tools. Ideally, service process should be able to transparently adapt to changes in the environment and to the needs of different customers with minimal or no user intervention. The eFlow supports the definition and enactment of adaptive and dynamic service process. A composite service is described as a process schema that composes other basic or composite services. A service process instance is an enactment of a process schema.

Task-Driven Computing [31] at CMU is based on the insight that with appropriate system support it is possible to let users interact with their computing environments in terms of high level tasks and free them from low level configuration activities. Computing devices and networks are becoming ubiquitous. In this new world, computing will no longer be tethered to desktops: users will become increasingly mobile. As users move across environments, they will have access to a dynamic range of computing devices and software services. They would want to use the resources to perform computing tasks. Today's computing infrastructure does not support this model of computing very well because computers interact with users in terms of low level abstractions: application and individual devices. As people move from one environment to another, they should be able to quickly migrate their computing context. Computing context includes the state of computing tasks, such as currently open documents and editing positions, open email messages application options such as POP settings and personal data. The illusion of “continuity” across environments is important when you have long-lasting tasks and highly mobile users. The key idea behind task-driven computing approach is that the system should take over many low-level management activities so that users can interact with a computing system directly in terms of high-level tasks they wish to accomplish. Their approach centers on following elements:

(1) system-maintained, globally-accessible computing tasks and task states;
(2) new type of service (or wrappers on legacy services) that supports automatic configuration;
(3) automatic selection and configuration of service coalitions to accomplish computing tasks and reestablish computing contexts;
(4) proactive guidance to help a user accomplish a task and compensate for resource limitations in an environment; and
(5) automatic management of dynamic services and resource conditions. The prototype contains limited set of the five elements, and their approach requires centralized management.

Further, the management of the state will affect performance of the system as the state information is heavy.

Anamika [4] is a reactive service composition architecture that is distributed, de-centralized and fault-tolerant for reactive service composition in pervasive environments. The development of customized services by integrating and executing existing ones has received a lot of attention in the last few years with respect to wired, infrastructure based web-services. However, service discovery and composition in web-based environments is performed in a centralized manner with the help of a fixed entity. Moreover, wired infrastructures do not take into consideration factors arising from the possible mobility of the service providers. Service composition system for pervasive computing environments need a different design approach than those developed for wired services. This is because many of the assumptions of standard composition architectures of wired services are no longer valid in such dynamic environments. Service composition architectures in wired infrastructure assume the existence of a centralized composition entity that carries out the discovery, integration and execution of services distributed over the web. They also need tighter coupling with the underlying network layers. The system comprises of five layers; network layer, service discovery layer, service composition layer, service execution and application layer. The *network layer* forms the lowest layer and encapsulates networking protocols that provide wireless/ad hoc connectivity to peer devices in the vicinity. The *service discovery layer* encompasses the protocol used to discover the different services that are available in the vicinity of a mobile device. Each device has a service manager where the local services register their information. Service manager advertises services to neighboring nodes and these advertisements are cached. Services are described using a semantically rich language DAML + OIL, which is used in service matching also. On a cache miss, the service request is forwarded to neighboring nodes. The *service composition layer* is responsible for carrying out the process of managing the discovery and integration of services to yield a composite service. The process model of the composite service is supplied as input to this layer (possible weakness or shortcomings). DAML-S is used to describe a process model. The service execution layer is responsible for carrying out the execution of different services. Prior to this, the service composition layer provides a feasible order in which the services can be composed and also provides location and invocation information of the services. The application layer embodies any software layer that utilizes the service composition platform. In the dynamic broker selection technique, a mobile device is selected as the broker of the composition user has requested, and the device is informed of its responsibility. The mobile device acting as *the broker is responsible for the whole composition process for a certain request*. The selection of the broker platform may be dependent on several parameters. Each request may be assigned a separate broker. However this selection process can swamp the system. To avoid this problem, the authors suggest that the requesting source could act as the broker of that request. The key idea of distributed brokering technique is to distribute the brokering of a particular request to different entities in the system by determining their "suitability" to execute a part of the composite request. In this approach, a single broker is not responsible for performing the whole composition.

5. Modeling Components of Pervasive Computing

In this section we present the model of the service provisioning framework for pervasive computing environments. First, we describe Pervasive Information Community Organization (PICO), a middleware framework that enhances existing services [17]. PICO deals with the creation of mission-oriented dynamic computing communities that perform tasks on behalf of users and devices autonomously. PICO consists of software entities called delegents (intelligent delegates) and hardware devices, called camileuns (connected, adaptive, mobile, intelligent, learned, efficient, ubiquitous nodes). PICO's objective is to provide *what we want, when we want, where we want and how we want* type of services autonomously and continually. The concept of PICO extends the current notion of pervasive computing; namely, that computers are everywhere [32]. The novelty of the PICO initiative lies in creating communities of delegents that collaborate proactively to handle dynamic information, provide selective content delivery, and facilitate application interface. In addition, delegents representing low-resource devices have the ability to carry out tasks remotely.

In this chapter, we propose a unique service provisioning framework that is not only *flexible* in defining and generating services, but also *adaptive* to the environment where the service is actually provided. Further, we develop an elegant model of the service provisioning framework, demonstrate the effectiveness of the proposed framework, and validate the model through case studies and simulation.

In this section, we present the following:

- Creation of a model that captures the essence of devices and services around users.
- Creation of a flexible and adaptive service composition scheme based on the model. The resulting new services, created by composing existing single services enhance service availability and user experience.
- Implementation of a reference software platform to carry out case studies based on the service composition scheme.

5.1 Modeling Pervasive Computing Devices

We use three entities—camileuns, delegents and communities to describe services and interactions among them in an abstract way. Camileuns capture the capabilities of physical devices that exist in pervasive computing environments. Delegents are used to model delegated or composed services from the native functionalities of a camileun. Interactions among delegents that are executing on a number of individual camileuns may lead to the formation of communities. In the following sections, we will describe these three modeling entities in detail.

Camileuns are logical entities used to depict the functionalities of physical devices. We have extended the notion of camileuns in PICO to provide a foundation for our service provisioning scheme. Camileuns are represented as a tuple $C = \langle C_{\text{id}}, F, H \rangle$ where C_{id} is the camileun identifier, F is the set of functional units, and H is the set of system characteristics. A functional unit is a hardware and/or software component that is capable of accomplishing a specific task. Each physical device may contain a number of functional units. To simplify our model for the proposed framework, we classify functional units of a camileun into four categories: sensors, actuators, processors and communicators.

- Sensor: A sensor monitors the physical features of the environment or receives inputs from and generates events. Events are to be consumed by other functional units, such as actuators, processors, communicators. The examples of sensors are physical sensors, keyboards, mice, etc.

- Actuator: An actuator listens to the stream of events and performs action(s) to pervasive computing environments. The resulting effects of an actuator's action is only limited to the local configuration of the camileun associated with the actuator. The examples of actuators are displays, disks, mechanical actuators, etc.
- Processor: A processor is a functional unit that translates/manipulates/filters incoming events and generates value added events. A processor can be a hardware or software component. The hardware components of this category include micro-processor units in a device, and the software components include driver modules and software applications installed on devices.
- Communicator: A communicator is similar to an actuator, except that it is intended to affect the environment external to a camileun. Typical communicators are hardware and/or software components that transmit or receive information to/from other devices. This category includes physical network device such as wired/wireless LAN routers/gateways and their accompanying software.

The four categories described above are sufficient to represent the nature of any functional units of camileuns. Each device can possess one or more of above three functions—input, process, and output. Seven combinations of these three functions are possible: (input); (process); (output); (input, process); (input, output); (process, output); (input, process, output). Out of these seven combinations, only three are practical within a camileun: {(input, process), (process, output), (input, process, output)}. The combination (process, output), and (input, process, output) are represented by sensor, and processor, respectively. The combination (input, process) is represented by actuator and communicator. Note that actuators affect device's internal environment, but communicators affect environments outside of devices.

This classification helps us to examine compatibility between functional units. Based on this classification, we define *directional compatibility*. We say two functional units, f_a and f_b, are *directionally compatible* if and only if f_a can be *followed by* f_b, otherwise they are not compatible.

Table I shows *followed-by* relationship of the four classifications. Sensors produce information consumed by processors, actuators, and communicators; we say sensors may be followed by all the other types of functions, and processors may be followed by all the others except sensors. The actuator and communicator are terminal functional units that are not followed by any other units.

The set H represents system characteristics and describes system aspects of a functional unit, such as type of input/output, and characteristics of internal operation. The definition of F, allows us to determine directional compatibility between

TABLE I
FOLLOWED-BY RELATIONSHIP OF THE FOUR FUNCTIONAL CATEGORIES

Successor / Predecessor	S	P	A	C
Sensor (S)	X	O	O	O
Processor (P)	X	O	O	O
Actuator (A)	X	X	X	X
Communicator (C)	X	X	X	X

functional units of devices, but we need more information to examine whether a combination of two functional units is semantically correct. Suppose, we have two devices, an image scanner and a printer, the scanner scans images and produces JPEG formatted files only. On the other hand, the printer can only print post-script formatted files. They are directionally compatible since the scanner's F set has a "sensor" and the printer's F set has an "actuator". They, however, are semantically incompatible. We say two functional entities are *semantically compatible* if they are directionally compatible and match input and output types. The set H in the camileuns is used to determine semantic compatibility. Therefore, two functional units, f_a and f_b, are *compatible* if and only if f_a can be *followed by* f_b, and they are *semantically compatible*. The predecessor and successor of a new composed service should be compatible.

5.2 Modeling Services

Camileuns capture services that devices can provide to users. However, camileuns are passive modeling entities and no operation could be performed on them without delegents. Delegents expose functional units of camileuns such that camileun capabilities can be exploited by service provisioning schemes. In this section, we first describe modeling of delegents in detail, and then go on to discuss briefly the mapping of functional units (services) of a camileun to delegent(s), in the following section.

A delegent is described by $D = \langle d_{\mathrm{id}}, M, R, S, Q\rangle$ where d_{id} is the identifier, M is the set of modules, R is the set of rules, S is the set of services provided by the delegent, and Q is a set of states.

Suppose a camileun c possesses a number of functional units f_i, and c^F represent the set of functional units of the camileun that is $c^F = \{f_i \mid i \geqslant 0\}$. There is a set of delegents that are executing on, or associated with the camileun c, and these delegents are represented by D_c. Each delegent in the set D_c utilizes one or more functional unit(s) of c^F. The functional units that are associated with a delegent d_k,

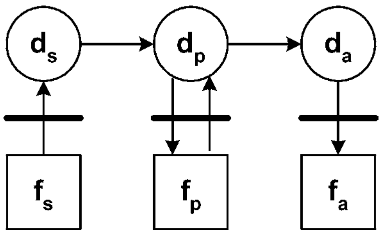


FIG. 2. Simple mapping between functional units of a camileun and delegents.

where $d_k \in D_c$, is selected on the basis of whether any functional unit $f_i \in c^F$ is required to carry out a service s, where $s \in d_k^s$, and d_k^s is the set of services provided by d_k.

The set M is formally defined as $M = \{m_i \mid m_i = \mu(f_i),\ f_i \in F\}$ where m_i is a module, μ represents the implementation of f_i, and F is the set of functionalities of the camileun associated with the delegent. Figure 2 depicts the relationship between a functional unit f_i of a camileun and a module m_i of a delegent. The set M consists of required modules to carry out S and the modules are derived from the set F of the camileun that the delegent is associated with. For example, M of a delegent delegating “print” functional units of printer can be modeled as follows: The delegents use the printer’s function, $f_1 = \{\text{actuator} = \{\text{print}\}\}$, the set of modules of the delegents is given by $M = \{m_1 = \mu(f_1 = \{\text{actuator} = \{\text{print}\}\})\}$. The details of mapping between M and F will be presented in Section 6. The set R defines how a delegent responds to events when it is in a certain state q_i where $q_i \in Q$. The operational rules also include community engagement, communication, and migration of delegents. R is defined by $R = \langle e, c, a\rangle$ where $e \in E_i$, and $a \in A_i$. E_i is a set of events defined in delegent d_i, and A_i is a set of actions that can be performed by delegent d_i running on a camileun c_i. The rules are maintained in a knowledge store, and retrieved by a simple search engine within the delegent. Note that currently rule matching is done by a simple search process. A conflict may arise when multiple rules are applicable in a context, and it is resolved by employing a simple tie-breaking rule that selects the rule that comes first in textual order in the current implementation.

The set Q is defined by $Q = \{q_0, \dots, q_n, \bar{q}\}$ where q_0 is the initial state and $\bar{q}$ is the final (goal) state. Q, a set of states of a delegent, includes the operational states, namely created, disposed, dormant, active, and mobile, and delegent specific states that inherited from functional units that are utilized by M. Each of the states related to general operations of delegents is associated with an event such as *instantiate, disposed, activate, roam, return, and wait/fail.* The FSM is translated into the rule specification of the delegents. Note that the “active” is a multi-state that contains

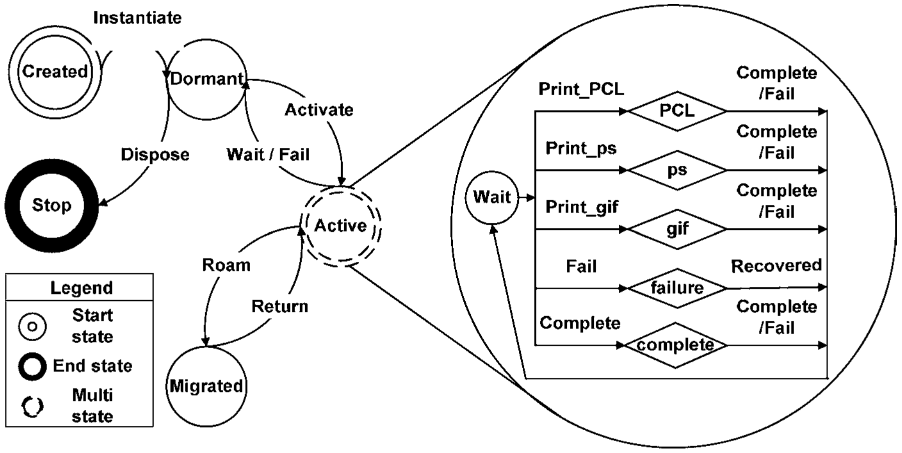


FIG. 3. Finite state machine of a printing delegent.

another self-contained FSM that describes how a job is to be carried out. The goal state of the delegent is defined as $\bar{q}$. $\nabla\bar{q}$ implies that delegent always attempts to be in the state of $\bar{q}$, and $\Diamond\bar{q}$ indicates that a delegent has finished its job after reaching the state $\bar{q}$. Figure 3 shows the finite state machine (FSM) of a printing delegent.

The S is a set of services provided by the delegent, and we define $S_i = \{s \mid s \text{ is a service provided by the delegent } d_i\}$. The set S should be descriptive enough in order for other delegents to know what services a delegent can provide.

5.3 Modeling Collaborative Services

Communities represent collaborative services among delegents. A community is described by $P = \langle p_{\mathrm{id}}, U, G, E\rangle$. Where, p_{id} is the identifier, U is the set of delegents that are members of the community, G is the set of community goals, and E represents community characteristics. E includes information of the leader of community and any other community management and operation related factors such as the rate of periodic vital message exchange, servicing model etc. Before defining G formally, the configuration of a community is defined. The configuration of a community is a collection of state information of member delegents, defined by $X_{p_i} = \prod_{i=0}^{n} Q_i$. We define the goal of community p_i as $G_{p_i} = \{g_i \mid g_i \in X_{p_i},\ i \geqslant 0\}$. The member selection process of a community P is carried out as follows. Let $\Delta = \bigcup_{i=0}^{n} D_{c_i}$ where $c_i \in X$ and $P = \{p_l \mid l \geqslant 0\}$, there exists a function $G_l : \Delta \to P_l^D$ where G_l is a goal of community, and P_l^D is the set of member delegents.

6. Service Provisioning Model

As described earlier, delegents are the first level service provisioning entities created by composing new services from functional units of devices. In single delegent scenarios, at any given time service provisioning by a delegent is limited to a single camileun. Communities complement this constraint by provisioning services over delegents that exist on a number of camileuns. We will illustrate the service provisioning process at both levels using the heart patient scenario described in Section 6.1.

6.1 An Example Scenario

The heart patient scenario is concerned with a person with chronic heart problems. S/he has an implanted heart monitor or is carrying the monitor. The heart monitor has the capability to monitor the heart activity and raise flags when there are some irregularities in the heart activity. The raised flag acts as the initiator for the creation of a logical community. There are other service provisioning delegents across the network that respond to flags from the heart monitor delegent and act accordingly.

In the HM scenario, we assume existence of two kinds of devices, PDA and cell phone. Camileun descriptions for the scenario are as follows.

- C_p^H = {(CPU, StrongArm), (CPU:Clock, 202 MHz), (Memory, 64 MB), (Network, IRDA), (IRDA:Bandwidth, 4 Mbps), (Network, 802.11b), (802.11b:Bandwidth, 11 Mbps), (Software, PIM), (PIM:Vendor, Microsoft), (PIM:Name, Outlook), …};
- $C_p^F = \{S = \{\text{HM:monitor};\ P = \{\text{HM:upload}\},\ A = \{\text{HM:alert}\}, \ldots\}$.

C_p^H and C_p^F represent the H and F of a PDA, respectively. Here we only show the elements required to describe our service provision framework, but there are more entries in each of the sets. The C_p^H of the PDA indicates that it has the StrongArm processor running at 202 MHz; 64 MB of memory, IRDA communication port whose bandwidth is 4 Mbps; and so on. The C_p^F of the PDA represents the functional units of the PDA, but, for illustration we present only three functional units such as monitor, upload and alert.

6.2 Delegent Level Service Provisioning

The issues of delegent level service provisioning are two fold:

(1) how to utilize F of camileun to create delegents that satisfy users' desire to exploit available services? and

(2) how to construct the rule set of a delegent?

In the HM scenario, we assume that in addition to the heart monitor carried by a patient, a PDA and a cell phone are also in the networking vicinity.

6.2.1 Simple Delegents

Each function of a camileun may be mapped to a single delegent. We call this type of delegent a simple delegent. The M of a simple delegent has only one module, m, and one interface to the functional units, f of a camileun. The advantages of such a mapping include simplicity, and easy implementation of a delegent's migration to another camileun.

Four kinds of simple delegents are created at the camileuns in the HM scenario,

- $D_{\text{HM}} \equiv \langle\{$Listen, parse_flag, alert_patient, upload_data, call_emergency$\}$, $\{$Rule$\}$, $\{$alert_patient, upload_patient_data, call_emergency$\}\rangle$;
- $D_{\text{CELL-PHONE}} \equiv \langle\{$listen, parse_request, initiate_call, place_call$\}$, $\{$Rule$\}$, $\{$make_call$\}\rangle$;
- $D_{\text{UPLOAD}} \equiv \langle\{$listen, validate & initiate, create_upload_channel, upload$\}$, $\{$Rules$\}$, $\{$upload_data_http, upload_data_ftp$\}\rangle$;
- $D_{\text{ALERT}} \equiv \langle\{$listen, extract_payload, alert$\}$, $\{$Rules$\}$, $\{$alert user$\}\rangle$;
- $D_{\text{CALL_EMERGENCY}} \equiv \langle\{$listen, extract_emergency_info, initiate_call, place_call$\}$, $\{$place call$\}\rangle$.

The disadvantage of this kind of service provisioning is that the number of delegents is proportional to the number of functions a camileun can carry out. The number of camileun functions depends on how each function of the camileun is defined and becomes unnecessarily large if the granularity of functional unit is too fine. This will result in a large number of delegents that in turn cause unnecessary duplication and hence increased overhead on the camileun. On the other hand, the number of camileun functions is unfairly small if the function is defined in a coarse way. The coarse functional definition of a camileun causes low utilization of a camileun's resources because of unnecessary sharing between two or more delegents. If a camileun function includes one or more sub-functions, the delegent is forced to include all of the sub-functions of a camileun function regardless of the usefulness of the sub-functions. Some of these issues are problems of concern in system decomposition.

6.2.2 Complex Delegents

Unlike simple delegents, complex delegents use two or more functional units of a single camileun. The main purpose of having this mapping is to provide delegent level service composition. Complex delegents compose services to carry out the compound tasks whereas simple delegents are limited to functionalities they are associated with. Sets of functional units utilized by delegents can be either overlapped or disjointed. The main reason for a complex delegent to utilize two or more functional units from different camileuns is that it causes difficulties in managing camileun resources.

There are many ways to compose complex delegents. After careful examinations, we have chosen two practical mechanisms: custom-built and event-driven service compositions. In custom-built service composition, users or programmers determine M, R, S, and Q of delegents and develop specification for delegents. Even though M is pre-defined, it does not mean that each module $m \in M$ is bound to a specific $f \in F$ of camileun. This implies adaptability of delegents—custom-built delegents can be run on any camileun that can fulfill M of the delegent. The R of a custom-built delegent acts as controller, just like the main function of conventional program.

Custom-built service composition is done *a priori*, but event-driven composition in dynamic situations is a real challenge. For example, in event-driven service composition, sensors may initiate the process of service composition. The events generated by sensors, or processors are routed to other functional units that can handle the event. The goal of the delegent or community is to handle the event, and the rule specifies the route of the packet.

6.3 Community Level Service Provisioning

Communities in PICO are formed either statically or dynamically. The former is called *service provider community* and the latter is called *dynamic community*. Service provider communities are created to provide services to various applications. The mission or goal of a service provider community and the rules for creation of such communities are pre-defined. Dynamic communities are created in response to exceptions. The goal of such a community is achieved through the occurrence of a series of events.

6.4 A Reference Software Implementation of the Proposed Service Provisioning Framework

In this section, we describe the implementation of the proposed service provisioning. We design the reference software implementation such that the following challenges are met:

(1) provide interoperability among devices to mask device heterogeneity;

(2) employ a message transport protocol that is reliable and efficient, and supports user mobility;
(3) provision service composition and discovery in the way users desire; and
(4) maintain integrity of the collaborative services throughout the life time of the services.

We choose a middleware platform to meet the above requirements. There are three reasons why we choose to proceed with middleware approach. First, a middleware can be designed to be independent of the operating system, programming language and actual location of servicing entities. Second, the application can make use of advanced features such as an event service or a transaction service provided by the middleware. Lastly, middleware approach enforces the object-oriented and modular design paradigms. The following section describes our middleware implementation in detail.

6.4.1 Layered Architecture

In our implementation there are three layers: delegent, delegent support, and adaptation as shown in Figure 4. The adaptation layer masks device and network heterogeneity. The delegent support layer is composed of the camileun's delegent manager and other managers that support execution of delegents. Delegents execute on top of the delegent layer.

The *camileun's delegent manager* typically resides on a host camileun. The camileun's delegent manager performs management tasks. Specific tasks that must be performed by the camileun's delegent manager include process, memory, and mobility management; and creation and deployment of delegents. It is also involved in event packet delivery process. The event handler passes event packets destined to delegents running on the local camileun to the camileun's delegent manager, then CDM relays the packets to destination delegents. The camileun's delegent manager keeps execution state of delegents, namely *created, disposed, dormant, active,* and *mobile*, but it does not keep the state information specific to delegent's modules. It will also accept and process the events associated with general operations of delegents, such as *instantiate, dispose, activate, roam, return*, and *wait/fail*.

The *adaptation layer* comprises PICO-compliance software so that delegents and other components running above this layer can adapt to existing hardware devices in the environment. PICO-compliance software consists of dynamic libraries that are necessary to mask underlying systems and network heterogeneity of devices. Recently, a plethora of wireless devices have been deployed in a wide variety of situations: pocket PCs, cell phones, 2-way pagers, Bluetooth devices, 3G multimedia terminals, and many others. Most of these devices have processors and their focus

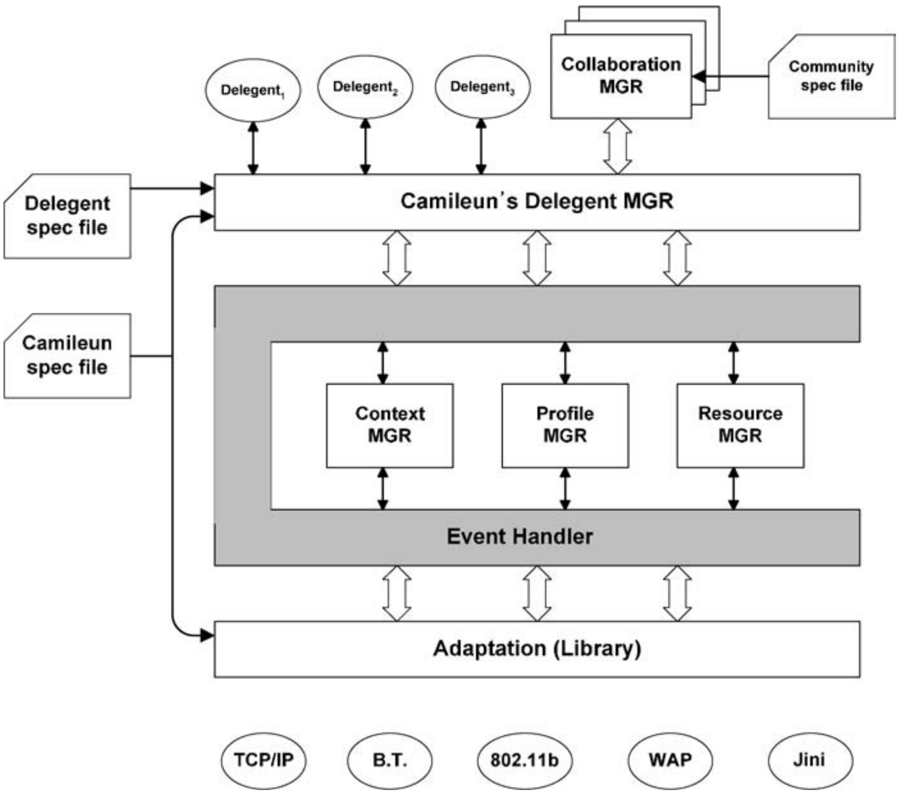


FIG. 4. The reference framework.

is on providing processing support to perform their own native functions. These device capabilities are captured by the set F of camileuns, and H is used to populate necessary dynamic library components in this layer.

The main task of the *event handler* is to deliver messages from the adaptation layer or the camileun's delegent manager to the appropriate recipient delegent(s). An event packet consists of source and destination delegent addresses, and event identification, and associated payload. Event packets destined to a delegent that is located on the local camileun are handed at the camileun's delegent manager for delivery of messages. If events are addressed to remote delegent, the event handler will carry out actual delivery by using one of the underlying communication mechanisms—such as 802.11b, Bluetooth, and IrDA, with help of the adaptation layer.

From time to time, the destination of an event packet can be unknown and its sink is needed to be determined during actual transmission of the packet. We call this kind of an event packet a *late-bound event packet*. To handle this type of packet, the event handler will consult with the community (collaboration) manager and other middleware components that maintain information about the environment the device is in, such as context manager, resource manager, and profile manager to determine the recipients of the events. Once the information of recipients of late bound events are resolved, they follow the same event delivery mechanism as that of normal event packets.

Current implementation of the *event handler* solely relies on the event identification of a packet to determine appropriate recipient delegent(s). However, this will create a problem of assigning unique event identification to every event in the environment. The requirement of unique identification of events hinders expansion of our service provision framework to wider environments. We are in the process of developing an event description language to replace the event id, so that the requirement of global uniqueness of event identification is resolved.

The *context manager* maintains contextual information for the user of the device. It is important for any pervasive computing device to exploit contextual information such that the amount of users' involvement is reduced. Contextual information is gathered by sensors or derived by software entities by fusing information available to them. The sensed/gathered contextual information is processed and filtered by the infrastructure to make useful to the components in the environment. The *context manager* helps to determine the recipient of late bound event packets using the current context of user or device, and it also generates event packets when there are context changes and a delegent's behavior needs to be adapted according to the changing context.

Many devices in pervasive computing environment are resource limited. The *resource manager* of a camileun is responsible for managing precious resource efficiently. The resource manager will generate an event whenever there is an exception when any resource, such as memory limit is reached or battery running low or overloaded CPU is nearing its limits. The exception might force delegents to react to the events by, for example, turning off the screen, reducing the power usage, etc.

6.5 Implementation of Delegents

We have implemented delegents that have the ability to provide encapsulation, interface, delegation, adaptation and manageability for the camileun with which they are associated.

- Encapsulation: A delegent encapsulates one or more functional units of a camileun. A simple delegent encapsulates only one functional unit, whereas a complex delegent encapsulates two or more functional units. Further, the concept of community provides a way for such encapsulations among multiple camileuns.

- Interface: A delegent provides a common interface to communicate or collaborate with other delegents. Community is a collection of one or more delegents working together towards a common goal. Communication and collaboration are essential to operations of a community.
- Delegation: Delegents delegate one or more functionalities of their associated camileuns to one or more communities. A delegent may represent services provided by a camileun to a community that requires camileun function(s) for its operation.
- Adaptation: Delegents make camileuns not only adaptive to their surrounding environments, but also condition them to overcome uneven conditions of various collaborating camileuns. When a camileun is in the environment with poor network resources, a delegent of the camileun will initiate collaboration with a another delegent on a different camileun to cache message packets.

Figure 5 depicts current delegent architecture designed to meet the above requirements. The *rule-driven scheduler* maintains the R and Q sets of delegents, accepts events from camileun's delegent manager (CDM) and schedules tasks according to rules and current state. The module database maintains elements of the set M of

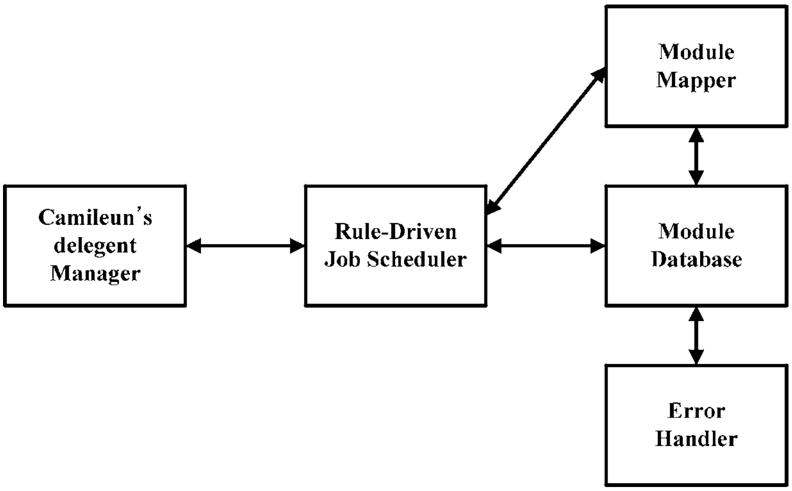


FIG. 5. Delegent architecture.

delegents. The module mapper links modules in the module database to actual f of camileun. Error handler deals with errors that might occur during the execution of delegents.

6.6 Implementation of Communities

Communities in PICO are defined as entities consisting of one or more delegents working towards achieving a common goal. Communities are the framework for collaboration and coordination among delegents. The community manager carries out the tasks of creation, merger and disintegration of communities. It also maintains the current states of the community, schedules activities among the member delegents according to the community specification, generates and processes events related to communities' operation. Since a community can have delegents executing on different devices, a community manager facilitates collaboration.

7. Petri Net Modeling and Simulation Study

We have carried out simulation studies to evaluate the proposed service provisioning framework in terms of its flexibility and effectiveness by employing Petri net modeling. Through the Petri net model, we prove such vital properties of the framework as *liveness* and *reachability*. We also gathered various performance results, such as average delay and scalability, to evaluate the framework quantitatively.

A model of existing or planned system captures the properties and behaviors of the system in an abstract way. We use a model to analyze and validate the properties of the system, and to carry out simulation studies to measure the performance of the system. Further, a model is utilized as a means of communication among various entities. Therefore, a modeling approach provides a guideline for representing behaviors and properties of a system as well as methodologies for analysis, validation, and simulation.

There exist many different modeling techniques for software systems. First, mathematical approaches such as first order predicate and temporal logic based ones. These approaches provide a concise way of representing a system, and have mathematically rigorous analysis and validation methods. However, this approach experiences difficulty as a means of communication, and it often requires translation of the model to other representations in order to carry out simulations using generic methods.

The second approach is an entity-relationship approach, and UML is one example. This approach provides an easier way of representing system, and facilitates communication among related entities of the model by employing visual representations. However this approach lacks mathematical rigor and means for analysis and validation. For this reason, this approach is often employed to design and develop a system only.

Petri-net based approach is another alternative for modeling a system. Petri-nets provide a mathematically based simple method that can be used for rigorous analysis and validation of modeled system. Petri nets (PNs) can also be used for Distributed Event Simulation (DES). They have been applied successfully in a wide variety of models, and have a mathematical basis which can be exploited to determine many different properties of modeled system.

A Petri net is a directed, connected, and bipartite graph in which each node is either a place or a transition. Tokens occupy places. When there is at least one token in every place connected to a transition, we say that the transition is enabled. Any enabled transition may fire by removing one token from every input place and depositing one token in each output place.

There are many different kinds of Petri models, but we can classify them into two main categories: ordinary Petri nets and high-level Petri nets. The ordinary Petri nets have shortcomings in many practical situations. First, they may become too large and inaccessible as we model details of the systems. Second, they are unsuitable for modeling particular activities, such as time. To resolve these issues, a Petri net have been expanded in many ways—color, time, and hierarchical extensions. Petri nets that incorporate the extensions are called high-level Petri nets.

Colored Petri net was developed in the late 1970s by K. Jensen [13]. In the ordinary Petri net, it is impossible to distinguish between two tokens: two tokens in the same place are by definition indistinguishable. Tokens often represent objects (e.g., resources, goods, humans) of the modeled system. To represent attributes of these objects, the Petri net model is extended with colored or typed tokens. Each token has a value often referred to as “color”. Transitions use the values of the consumed tokens to determine the values of produced tokens. A transition describes the relation between the values of the “input tokens” and the values of “output tokens”. It is also possible to specify “preconditions” which take the color of tokens to be consumed into account.

Another extension is the timed Petri net [21]. Given a process modeled as a PN, we often want to be able to make statements about its expected performance. In order to obtain such metrics, it is necessary to include pertinent information about the timing of a process in the model. The ordinary Petri net does not allow the modeling of ‘time’. In the time extension, tokens receive a timestamp as well as a value. This indicates the time from which the token is available. A transition is only enabled at the moment when each of the tokens to be consumed has a timestamp equal or prior to the current time. In other words, the enabling time of a transition is the earliest moment at which its input places contain enough available tokens. Tokens are consumed on a FIFO basis. Timed Petri net is classified into three categories based on distribution of time—deterministic, stochastic, and generalized stochastic. In deterministic time Petri nets, transitions happen in deterministic time. Stochastic net specifies time delays as exponentially distributed random variables. Generalized stochastic Petri net can have both deterministic and exponential distribution of transition time.

7.1 Petri Net Model

Petri net has been proven to be useful in verifying properties and measuring the performance of a system. A Petri net is a directed, connected, and bipartite graph in which each node is either a place or a transition. Tokens occupy places. When there is at least one token in every place connected to a transition, we say that the transition is enabled. Any enabled transition may fire by removing one token from every input place and depositing one token in each output place. Interested readers are referred to [21].

Figure 6 depicts the Petri net model of a delegent that represents a composed service using the proposed framework. The composed services are a partially ordered set of operations. Therefore, it is easy to map them onto a Petri net. Operations are modeled by transitions and the states of the services are modeled by places. The arrows between places and transitions are needed to specify causal relationships. In Figure 6, the places P1 and P10 are the input and output places of a delegent respectively. The place P3 represents the rule engine that matches current event to particular modules that a delegent is associated with. Petri net representation of a module contains one input and an output place. In the figure, the delegent is associated with three modules. Each module carries out its own task(s), and exposes unique stochas-

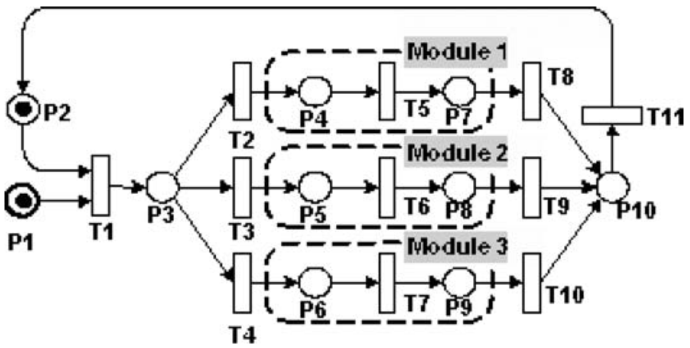


FIG. 6. Petri net model.

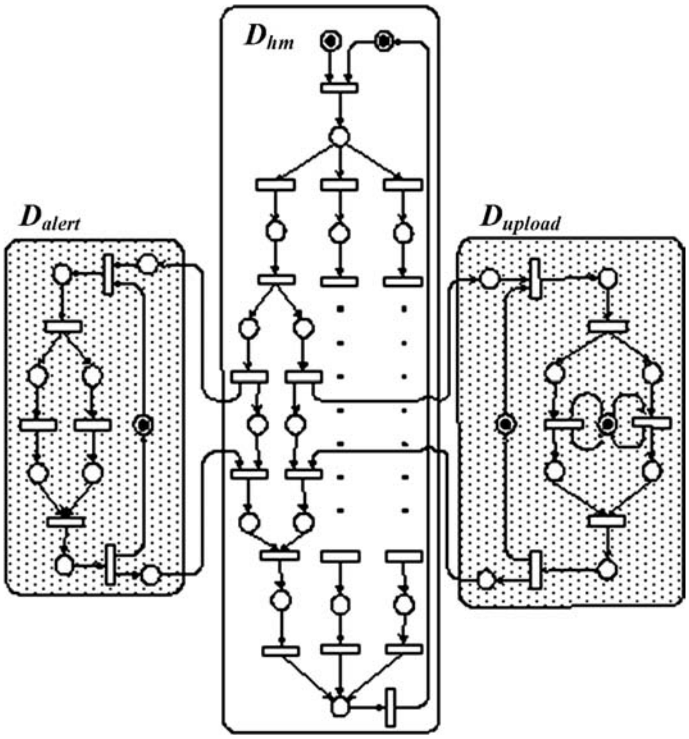


FIG. 7. Petri net model for multiple delegents collaborating.

tic processes to model the time to complete its task(s). For example, the module 1 uses uniform distribution, but modules 2 and 3 use exponential distribution.

For a delegent to complete its task(s), it may be required to collaborate with other delegents. In the heart patient scenario, the heart monitor delegent, (D_{hm}), collaborates with the alert (D_{alert}) and upload (D_{upload}) delegents. A Petri net model for the scenario is presented in Figure 7. As we described earlier, D_{hm} reacts to events differently depending on their severity. Figure 7 is a model that focuses on how D_{hm} reacts to a medium risk condition. From this model, we have conducted liveness and reachability study by employing the Petri net tool box [19]. We draw a conclusion that the model is live and all the places in the model are reachable. According to the definition of liveness of Petri net, if a transition is not deadlocked, then it must be live, i.e., there exists a firing sequence such that the transition will be enabled. Therefore, the Petri net of the composed service created using our Petri net model is deadlock free.

In the actual modeling of our whole system, we use G-net [7] due to the explosive number of places and transitions in the nets. G-net is a high level Petri-net that employs multiple levels of abstractions but it is inadequate for modeling communications between entities, such as delegents. For communication model, we are investigating Colored Petri net (CPN) [14]. CPN is an extension of Petri net capable of modeling communication. Detailed modeling and analysis of using the techniques will be carried out in the future.

7.2 Simulation Results

We have extended the pure Petri net model, shown in Figure 7, to stochastic Petri net to evaluate performance of the net. We have developed two types of workload models for our performance study: Sequential and Markov. In the Markov process, the inter-arrival time of each request is obtained from the Poisson process, while the inter-arrival time of sequential process is uniformly distributed.

In the simulation, average processing delay of events is of particular interest. We are interested in determining the processing delay of events to study the scalability of our proposed framework. A system is scalable whenever it is able to guarantee a service level when the size the system is significantly changed. Therefore a relation-

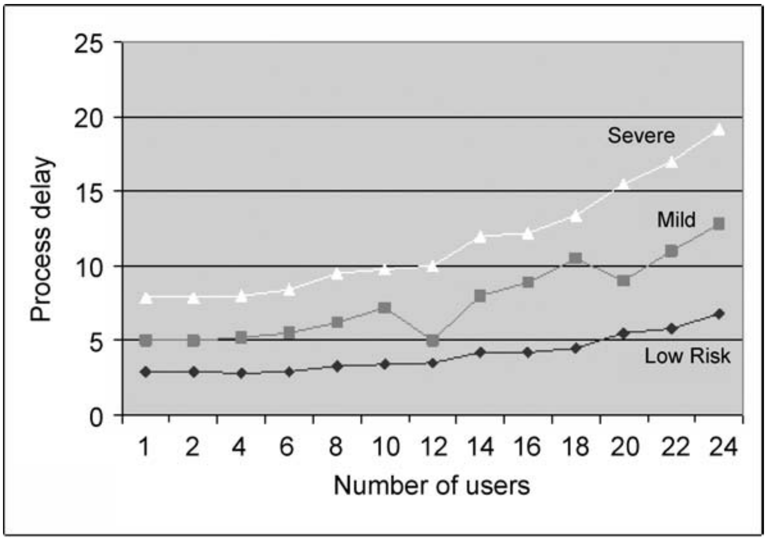


FIG. 8. Processing delay of events.

ship between a number of requests and delay time should be measured to show if a system is scalable.

Figure 8 shows the simulation results. The x-axis of the graph represents the number of concurrent users and the y-axis shows the corresponding average delay. The average delay is the difference between arrival time of the request at the place P1 and departure time at the place P10 of the Petri net shown in Figure 7. As shown in the graph, the system suffers little variance in processing delays as we increase the load. Therefore we conclude that the system is scalable.

The response time of the Petri net, shown in Figure 8, depends much on the performance of modules embedded in the delegent. The functions of delegents, such as rule inference, and mapping between events to modules, are carried out in deterministic time. However the run time of modules depends on tasks they carry out, and are modeled as Markov processes in our simulation. The encapsulation provided by delegents does not introduce much overhead.

8. Modeling Dynamism of Pervasive Computing Using Colored Petri Nets

Pervasive computing systems operate under very dynamic and ever changing environments. Continuous changes in pervasive computing environments require the system to react to dynamic changes in a seamless and unobtrusive manner. One of the critical issues realizing this vision is to make pervasive computing applications aware of the changing environment. Various monitoring mechanisms, such as periodical probing and push-based mechanisms are developed to aid applications in pervasive computing to become aware of environmental changes. These mechanisms have their own strengths and weaknesses and also exhibit different performance characteristics that influence how fast and how well applications in pervasive computing adapt themselves in the changing environment and affect the performance of the entire system. Study of the characteristics and performance evaluation are important to the design and implementation of any pervasive computing system. We will investigate at how these characteristics affect pervasive computing systems.

A typical pervasive computing system is depicted in Figure 9. There are three major components in a pervasive computing system, namely a group of applications, a middleware, and an environment. An environment of pervasive computing applications can be represented as a union of current states of resources and other applications that affect the environment. Most pervasive computing systems employ middleware in order to resolve heterogeneity and more important to capture changes in the environment. Middleware is normally implemented as a group of processes such that each process carries out a specialized function, namely context manager,

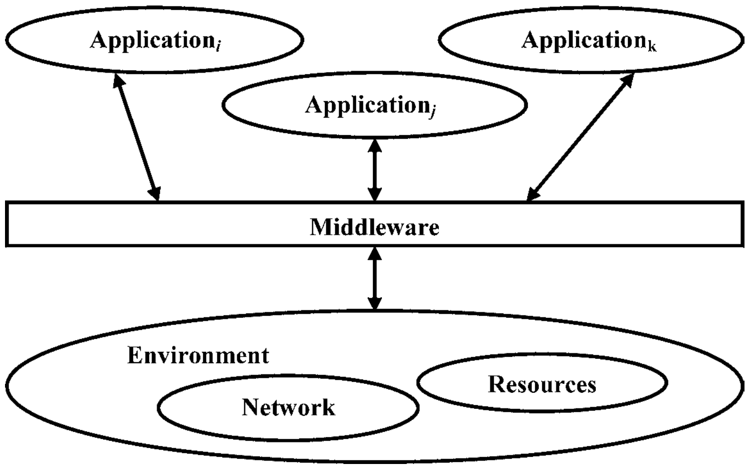


FIG. 9. Typical pervasive computing system.

resource manager, network monitors, etc. Middleware detects changes by monitoring the environment or receiving notification from applications and middleware that run on other systems. Another critical function of middleware is to interact with applications running on the top of them in order to sense environmental changes.

The interaction mechanism between the middleware and a group of applications can be classified into three different categories:

(a) Application initiated method (periodic): In this method, the middleware stores changes in the environment, and applications periodically probe the middleware for any environmental changes of interest. Since this model uses periodical probing, changes in the environment are often not propagated to applications in time and causes inconsistency between states as perceived by the middleware and the applications. Applications should tolerate temporal inconsistency in this model.

(b) Middleware initiated method: When a middleware perceives a change in this environment, it multicasts a message to notify applications of the change that has been made in the environment. The major advantage of this model is that applications receive timely information on changes happening in the environment. However this will result in waste of bandwidth and processing time because of unnecessary notification. All applications will receive notification messages, and this results in performance degradation since those applications that are not interested in a particular change are also interrupted.

(c) Hybrid method: Assume that a middleware is intelligent enough to determine the urgency of changes in the environment. A middleware notifies application changes in the environment using the push model described above. Otherwise it will buffer the changes until the application probes it.

We argue that the periodical probing is the best mechanism for pervasive computing systems due to the following reasons. First, middleware for pervasive computing systems needs to be light weight, since pervasive computing deals with devices of various levels of intelligence. Second, each application has its own interest in the environmental information. According to the end-to-end argument [25], it is better for the application to seek information needed rather than information push from the middleware that may not have full knowledge of applications' interests.

In this section, we validate the above statement by modeling how changes made in the environment to the applications running in the environment using colored Petri net modeling. The model is focused on propagation of changes in the environment to applications. We also carry out the study on performance of each of the above three models.

8.1 Colored Petri Nets (CPNs)

The first proposed Petri net has limitations on representation of some aspects of the modeled system, such as time, types of messages, etc. There are several extensions to the Petri net to accommodate such aspects that could not be modeled with the original Petri nets. Colored Petri net (CPN) is an example of such extensions.

CPN provides more powerful way of describing operations of a system. Unlike ordinary Petri net, CPN uses tokens that can be differentiated from each other. This feature is the primary reason to employ CPNs for modeling. Further, CPNs allow us to model routing of message more precisely than ordinary Petri nets. Since pervasive computing system intensively interacts among software components, it is crucial that message exchanges are modeled correctly.

Another advantage of CPNs is that there are many modeling tools developed and available to aid CPN modeling. Most of the tools allow modeler to include custom-made procedures in the model by using ML or Java programming language. Modeler can reflect the behavior of the system precisely. By utilizing these features, we can model some level of dynamism of pervasive computing environments. The CPN model may also be analyzed for the certain correctness property of the system, namely liveness and safeness. Further it can be easily and directly translated into a prototype of the modeled system. King et al. [15] propose a way of translating a CPN model to a UML so that it can easily be implemented using existing software implementation tools. The above features make CPN a suitable candidate for modeling of dynamism of pervasive computing system.

8.2 Modeling

In this section, we will describe three methods that attempt to resolve inconsistency between applications and the middleware. We also present CPN models for each method and discuss the properties of each of the methods using CPN analysis technique.

8.2.1 Application Initiated Periodical Probing

Application initiated periodical probing requires that an application in a pervasive computing environment periodically probes the middleware, it is running on top of, for possible changes. This can be implemented as simple request–reply protocol depicted in Figures 10 and 11. An application running on top of middleware sends out probe messages periodically. The period of probing is represented by ΔT. In other words, the frequency of updates is $1/\Delta T$. Once a middleware receives the probe message, it will reply to the application with current status information.

The periodical probing method is easier to implement, but suffers from the following disadvantages.

Accuracy problem: Suppose that the period of probing is ΔT. During ΔT time, the application makes all its decisions based on the result of the last probing. If the environment changes quickly and frequently as is often the case in dynamic environments, the accuracy is affected.

Middleware load problem: In order to improve accuracy, the ΔT should be as short as possible, but, on the other hand, too many probing messages should be avoided. Probing messages generate a heavy load on middleware and consume much bandwidth.

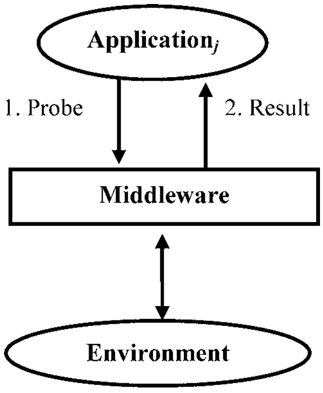


FIG. 10. Application initiated periodical probing.

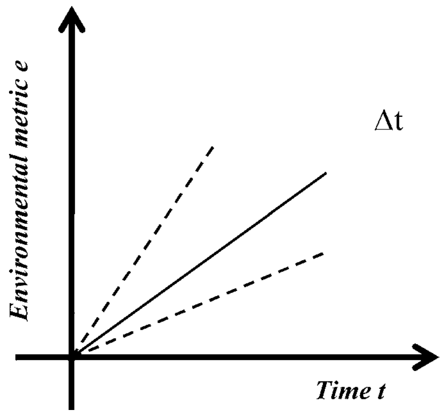


FIG. 11. Relationship between environmental change rate and accuracy.

Solving the above problems is hard because they are contradictory. We need to look for the value of ΔT that can satisfy both issues adequately.

8.2.2 Determining ΔT

As discussed above, the probe period ΔT is the main performance factor of the algorithm. From information theory, we can determine ΔT based on how fast the environment is changing. Let's define that environmental changing rate is δ and accuracy of monitoring is defined by ε.

The worst possible divergence between application and middleware is given by,

$$\varepsilon = 2\delta\Delta T.$$

From this we can get the following,

$$\Delta T = \varepsilon/2\varepsilon.$$

Given the required accuracy ε for the application and system's environmental changing rate δ, ΔT can be determined as above. For an application to accurately monitor the environment, it needs to send a probing message every ΔT. This will be studied through simulation using the CPN model described in the following section.

8.2.3 CPN Model

In this section, we consider a system that is composed of applications and middleware services. We denote a set of applications and middleware services in the system as A and M, respectively. An application $x \in A$ is running on top of a middleware service $y \in M$, and we say $x \times y \in A \times M$. Two middleware services $y, z \in M$ may be connected by a bidirectional channel.

We describe dynamic behavior of the model using CPNs. A Colored Petri net consists of a set of places, which are graphically represented as ellipses, and a set of transitions, which are represented by squares. Places and transitions are related to each other by arcs. A state of the net is represented by tokens on the different places of the net, and is changed by firing transitions. A transition can be fired if there is a matched token in each of input arcs.

A CPN for periodic probing method is depicted in Figure 12. The upper portion of the CPN above represents behavior of application and the bottom one represents the behavior of middleware.

The place *Application* holds an application $x \in A$, and place *Middleware* has $y \in M$. Initially, an application x probes a set of middleware services y by sending a request message. The transition *Send_Req* abstracts the sending of a request message. Since the application sends probe messages to all the middleware it needs

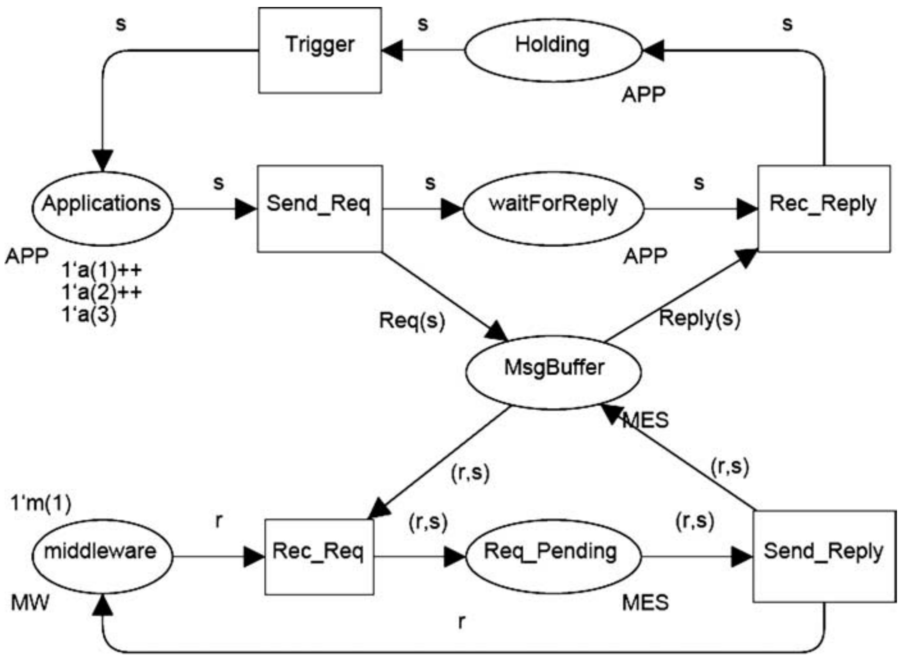


FIG. 12. CPN modeling of periodic probing.

to probe, arc-inscription $Req(s)$ generates a set of messages $(x, y) \in A \times M$. The generated messages are buffered at place MsgBuffer, until they are consumed by transition *Req_Rcv*. When a message is placed at MsgBuffer, the transition *Rec_Req* is enabled if there exists a token $z \in M$ such that messages $(x, y) \in A \times M$ are in MsgBuffer and $y = z$. Once the transition *Rec_Req* is fired, the message (x, y) which is consumed by the transition is placed in Req_Pending. The place Req_Pending represents the processing delay in the middleware for a request message to be processed. The transition Send_Reply creates reply messages indicating whether the request has been processed successfully or not. The reply message is placed in Msgbuffer again, and it makes *Rec_Reply* enabled so that message pending in the place waitForReply can be consumed by *Rec_Reply* transition. Firing *Rec_Reply* represents the completion of probing sequence. Once one probing is done, the application waits for a pre-defined period of time for the next probing to start. The transition Trigger is ready to fire when the token in the place Holding is aged more than the defined timeout period. The application token is reincarnated by the firing of Trigger.

In order for the probing mechanism to be logically correct, two properties must be satisfied—liveness and safety. Liveness property is defined as a desirable state that will be entered from all legal initial states of the system. In our mechanism, we can say the system is live if every application's probe message reaches the place of completion. More formally, each probe message a should reach the state completed. The above statement says every probe message eventually reaches the place of COMPLETED. We have proved the liveness of the mechanism by running CPN tool's state space analysis.

Safety requires that an incorrect state cannot be entered from any initial state of the system. In our mechanism, the application reaches completed state after all the middleware services reply back to the application.

9. Conclusions and Future Work

In this chapter, we introduced a service provisioning framework that is not only flexible in defining and generating services, but also adaptive to the environment where the service is actually provided. The proposed service provisioning framework aids users in locating and utilizing available services, and composing new services using the existing services in the environment. We described a model that captures the essence of devices and services around users in pervasive computing environments. The model provides high-level abstractions of services and relationship among them. The model allows depiction of enhancements service provisioning and availability and user's experience. We describe the implementation of a reference software platform to carry out case studies based on the service composition scheme.

We also presented an analytical model using Petri net and simulation results of the model. The Petri net model shows that services created by our framework are live (deadlock-free) and effective, and the overheads introduced do not affect scalability. A colored Petri net model has been developed to capture the environmental changes and the recognition of such changes by the middleware and the application.

REFERENCES

[1] Arnold K., O'Sullivan B., Scheifler R.W., Waldo J., Wollrath A., *The Jini Specification*, Addison–Wesley, Longman, Reading, MA, 1999.

[2] Banavar G., Beck J., Gluzberg E., Munson J., Sussman J., Zukowski D., "Challenges: an application model for pervasive computing", in: *6th Annual International Conference on Mobile Computing and Networking (MOBICOM 2000), Boston MA, USA*, 2000, pp. 266–274.

[3] Casati F., Ilnicki S., Jin L.-J., Krishnamoorthy V., Shan M.-C., "eFlow: a platform for developing and managing composite e-services", in: *AIWoRC 2000*, 2000, pp. 341–348.

[4] Chakraborty D., Perich F., Joshi A., Finin T., Yesha Y., "A reactive service composition architecture for pervasive computing environments", in: *7th Personal Wireless Communications Conference, Singapore*, 2002.

[5] The DAML Service Coalition, "DAML-S: semantic markup for Web services", http://www.daml.org/services/daml-s/0.9/daml-s.pdf, 2003.

[6] Crosbie M., Spafford E.H., "Defending a computer system using autonomous agents: making security real", in: *Proceedings of the 18th National Information Systems Security Conference, Baltimore, MD*, 1995, pp. 549–558.

[7] Deng Y., Chang S.K., Figueiredo J.C.A.D., Perkusich A., "Integrating software engineering methods and Petri nets for the specification and prototyping of complex information systems", in: *Application and Theory of Petri Nets 1993, Proceedings 14th International Conference, Chicago, Illinois, USA*, in: *Lecture Notes in Computer Science*, 1993, pp. 206–223.

[8] Erol K., Hendler J., Nau D.S., "HTN planning: complexity and expressivity", in: *12th National Conference on Artificial Intelligence (AAAI-94), Seattle*, 1994, pp. 1123–1128.

[9] Guttman E., "Service location protocol: automatic discovery of IP network services", *IEEE Internet Computing* **3** (1999) 71–80.

[10] Helal S., "Standards for service discovery and delivery", *IEEE Pervasive Computing* (2002) 95–100.

[11] Hermann R., Husemann D., Moser M., Nidd M., Rohner C., Schade A., "DEAPspace—transient ad hoc networking of pervasive devices", *Computer Networks* **35** (2001) 411–428.

[12] Hopper A., "The Royal Society Clifford Paterson Lecture, 1999, Sentient Computing", AT&T Lab Cambridge Technical Report 1999.12, 1999, pp. 1–10.

[13] Jensen K., "Coloured Petri nets and the invariant method", *Theoretical Computer Science* **14** (1981) 317–336.

[14] Jensen K., "An introduction to the practical use of coloured Petri nets", in: *Lectures on Petri Nets II: Applications*, in: *Lecture Notes in Computer Science*, vol. 1492, 1998.

[15] King P., Pooley R., "Derivation of Petri net performance models from UML specifications of communications software", in: *11th International Conference, TOOLS 2000, Schaumburg, IL, USA*, 2000, p. 262.

[16] Kumar M., Das S.K., Shirazi B., Levine D., Marquis J., Welch L., "Pervasive Information Community Organization (PICO) of camileuns and delegents for future global networking", in: *LSN Workshop, Vienna, USA*, 2001.

[17] Kumar M., Shirazi B., Das S.K., Singhal M., Sung B.Y., Levine D., "PICO: a middleware framework for pervasive computing", *IEEE Pervasive Computing* **2** (2003) 72–79.

[18] Leymann F., "Web Services Flow Language (WSFL 1.0)", http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, 2001.

[19] Mahulea C., Matcovschi M.-H., Patravanu O., "Learning about Petri net toolbox for use with MATLAB", http://www.ac.tuiasi.ro/pntool/index.php, 2004.

[20] Microsoft, "COM: delivering on the promises of component technology", http://www.microsoft.com/com/, 2002.

[21] Murata T., "Petri nets: properties, analysis and applications", *Proceedings of the IEEE* **77** (1989) 541–580.

[22] OMG, "CORBA component model, v3.0", http://www.omg.org/technology/documents/formal/components.htm, 2002.

[23] Pingali G., Pinhanez C., Levas A., Kjeldsen R., Podlaseck M., Chen H., Sukaviriya N., "Steerable interfaces for pervasive computing spaces", in: *PerCom, Fort Worth, TX*, 2003.

[24] Roman M., Hess C.K., Cerqueira R., Ranganathan A., Campbell R.H., Nahrstedt K., "Gaia: a middleware infrastructure to enable active spaces", *IEEE Pervasive Computing* (2002) 74–83.

[25] Saltzer J.H., Reed D.P., Clark D.D., "End-to-end arguments in system design", *ACM Trans. Comput. Syst.* **2** (1984) 277–288.

[26] Satyanarayanan M., "Pervasive computing: vision and challenges", *IEEE Personal Communications* **8** (2001) 10–17.

[27] Stanford V., "Pervasive health care application face tough security challenges", *IEEE Pervasive Computing* **1** (2002) 8–12.

[28] Sung B.Y., Shirazi B., Kumar M., "Community computing framework for enhancing Internet service", in: *Eurasian Conference on Advances in Information and Communication Technology, Iran*, 2002.

[29] Tennenhouse D., "Proactive computing", *Communication of the ACM* **43** (2000) 43–50.

[30] W3C, "Web Services Description Language (WSDL) 1.1", http://www.w3.org/TR/wsdl, 2001.

[31] Wang Z., Garlan D., "Task-driven computing", Technical Report, CMU-CS-00-154, 2000.

[32] Want R., "New horizons for mobile computing", in: *PerCom, Fort Worth, TX, USA*, 2003.

Search and Retrieval of Compressed Text

AMAR MUKHERJEE, NAN ZHANG, TAO TAO,
RAVI VIJAYA SATYA, AND WEIFENG SUN

School of Computer Science
University of Central Florida
Orlando, FL 32816
USA

Abstract
In recent times, we have witnessed an unprecedented growth of textual information via the Internet, digital libraries and archival text in many applications. To be able to store, manage, organize and transport the data efficiently, text compression is necessary. We also need efficient search engines to speedily find the information from this huge mass of data, especially when it is compressed. In this chapter, we present a review of text compression algorithms, with particular emphasis on the LZ family algorithms, and present our current research on the family of Star compression algorithms. We discuss ways to search the information from its compressed format and introduce some recent work on compressed domain pattern matching, with a focus on a new two-pass compression algorithm based on LZW algorithm. We present the architecture of a compressed domain search and retrieval system for archival information and indicate its suitability for implementation in a parallel and distributed environment using random access property of the two-pass LZW algorithm.

ADVANCES IN COMPUTERS, VOL. 63
ISSN: 0065-2458/DOI 10.1016/S0065-2458(04)63006-3

1. Introduction

In recent times, we have witnessed an unprecedented growth of textual information via the Internet, digital libraries and archival text in many applications. A recent estimate [42] puts the amount of new information generated in 2002 to be 5 exabytes (1 exabyte $= 10^{18}$ bytes which is approximately equal to all words spoken by human beings) and 92% of this information is in hard disk. While a good fraction of this information is of transient interest, useful information of archival value will continue to accumulate. The TREC [62] database holds around 800 million static pages having 6 trillion bytes of plain text equal to the size of a million books. The Google system routinely accumulates millions of pages of new text information every week. We need ways to manage, organize and transport this data from one point to the other on data communications links with limited bandwidth. We must also have means to speedily find the information we need from this huge mass of data. Search engines have become the most popular and powerful tools for hunting relevant documents on the Internet. Although many well known search engines claim to be able to search multimedia information such as image and video using

Abbreviations

AC:	Acho–Corasick Automaton	**PPM:**	Prediction by Partial Matching
BPC:	Bits Per Character	**RLE:**	Run Length Encoding
BWT:	Burrows–Wheeler Transform	**RLPT:**	Reversed Length Preserving Transform
DC:	Distance Coding	**SCLPT:**	Shortened Context Length Preserving Transform
DTD:	Document Type Definition	**TREC:**	Text REtrieval Conference
LIPT:	Length Index Preserving Transform	**UD:**	Uniquely Decodable
LPT:	Length Preserving Transform	**VLC:**	Variable Length Coding
LZW:	Lempel–Ziv–Welch Algorithm	**XML:**	Extended Markup Language
LZ:	Lempel–Ziv Algorithm		
MTF:	Move To Front		

sample images or text description, text documents are still the most frequent targets. Sometimes, a single site may also contain large collections of data such as a library database, thereby requiring an efficient search mechanism even to search within the local data. To facilitate the information retrieval, an emerging ad hoc standard for uncompressed text is XML which preprocesses the text by putting additional user defined metadata such as DTD or hyperlinks to enable searching with better efficiency and effectiveness. This increases the file size considerably, underscoring the importance of applying text compression. On account of efficiency (in terms of both space and time), there is a need to keep the data in compressed form for as much as possible.

Text compression is concerned with techniques for representing the digital text data in alternate representations that take less space. Not only does it help conserve the storage space for archival and online data, it also helps system performance by requiring less number of secondary storage (disk or CD Rom) accesses and improves the network bandwidth utilization by reducing the transmission time. Unlike static images or video, there is no international standard for text compression, although compressed formats like *.zip*, *.gz*, *.Z* files are increasingly being used. In general, data compression methods are classified as lossless or lossy. Lossless compression allows the original data to be recovered exactly. Although used primarily for text data, lossless compression algorithms are useful in special classes of images such as medical imaging, finger print data, astronomical images and data bases containing mostly vital numerical data, tables and text information. In contrast, lossy compression schemes allow some deterioration and are generally used for video, audio and still image applications. The deterioration of the quality of lossy images are usually not detectable by human perceptual system, and the compression systems exploit this by a process called '*quantization*' to achieve compression by a factor of 10 to a couple of hundreds. Many lossy algorithms use lossless methods at the final stage of the encoding stage underscoring the importance of lossless methods for both lossy and lossless compression applications.

In order to be able to utilize effectively the full potential of compression techniques for the future retrieval systems, we need to meet two challenges: First, we need efficient information retrieval in the compressed domain. This means that techniques must be developed to search the compressed text without decompression or only with partial decompression independent of whether the search is done on the text or on some inversion table corresponding to a set of key words for the text. Second, different parts of the compressed text should be made available for parallel search in a distributed or networked environment. This is important particularly when the system has to deal with massive amount of data even when compressed. Distributed search capability is a key feature for search engines like Google but none of the search engines available today make explicit use of compressed domain search. In order to be able to do that the compression algorithms must be capable of or should be modified to have search awareness via random access property. Being able to randomly access and partially decode the compressed data is highly desirable for efficient retrieval and is required in many applications. The flexibility for access to different levels of details of the context is also desirable for various requirements. We will deal with this subject again later in the chapter.

Compression algorithms are inherently sequential. Text compression can be done on-the-fly by using universal adaptive algorithms such as LZ family of algorithms or by typically two-pass algorithms such as Bzip2. The two pass algorithms cannot be used for on-line applications but in many applications such as archival library information or Internet based browsing, their importance is undeniable. Furthermore, use of distributed and network based searching are only possible if the compression algorithms are applied in the preprocessing stage. This point will be further clarified later in the chapter.

The plan of the chapter is as follows. Section 2 presents the basic concepts of information theory applicable in the context of lossless text compression. Section 3 will then briefly describe the well-known compression algorithms discussed in detail in several excellent recent books [52,64,33,53,47]. In Section 4 we present our research on the Star family of compression algorithms. Section 5 will review and present some of our own research in compressed domain search algorithms. Section 6 will discuss search and retrieval systems using compressed text data bases. Section 7 will provide some searching techniques with the help of useful data structures. Section 8 will present our ideas on parallel search and retrieval approaches.

2. Information Theory Background[1]

The general approach to text compression is to find a representation of the text requiring less number of binary digits. The standard ASCII code uses 8-bits[2] to encode each character in the alphabet. Such a representation is not very efficient because it treats frequent and less frequent characters equally. If we encode frequent characters with a smaller (less than 8) number of bits and less frequent characters with larger number of bits (possibly more than 8 bits), it should reduce the *average* number of *bits per character* (BPC). This observation is the basis of the invention of the so-called Morse code and the famous Huffman code developed in the early 50s. Huffman code typically reduces the size of the text file by about 50–60% or provides compression rate of 4–5 BPC [64] based on statistics of frequency of characters. In the late 1940s, Claude E. Shannon laid down the foundation of the information theory and modelled the text as the output of a source that generates a sequence of symbols from a finite alphabet A according to certain probabilities. Such a process is known as a *stochastic process* and in the special case when the probability of occurrence of the next symbol in the text depends on the previous symbols or its context it is called a *Markov process*. Furthermore, if the probability distribution of a typical sample represents the distribution of the text it is called an *ergodic process* [54,56]. The information content of the text source can then be quantified by the entity called *entropy* H given by

$$H = -\sum p_i \log p_i \tag{1}$$

where p_i denotes the probability of occurrence of the ith symbol in the text, sum of all symbol probabilities is unity and the logarithm is with respect base 2 and $-\log p_i$ is the amount of *information* in bits for the event (occurrence of the ith symbol). The expression of H is simply the sum of the number of bits required to represent the symbols multiplied by their respective probabilities. Thus the entropy H can be looked upon as defining the average number of BPC required to represent or encode the symbols of the alphabet. Depending on how the probabilities are computed or modeled, the value of entropy may vary. If the probability of a symbol is computed as the ratio of the number of times it appears in the text to the total number of symbols in the text, the so-called static probability, it is called an Order (0) model. Under this model, it is also possible to compute the dynamic probabilities which can be roughly described as follows. At the beginning when no text symbol has emerged

[1] Sections 2, 3, and part of Section 4 are directly taken from [49].

[2] Most text files do not use more than 128 symbols which include the alphanumerics, punctuation marks and some special symbols. Thus, a 7-bit ASCII code should be enough.

out of the source, assume that every symbol is equiprobable.[3] As new symbols of the text emerge out of the source, revise the probability values according to the actual frequency distribution of symbols at that time. In general, an Order(k) model can be defined where the probabilities are computed based on the probability of distribution of the $(k+1)$-grams of symbols or equivalently, by taking into account the context of the preceding k symbols. A value of $k = -1$ is allowed and is reserved for the situation when all symbols are considered equiprobable, that is, $p_i = 1/|A|$, where $|A|$ is the size of the alphabet A. When $k = 1$ the probabilities are based on bigram statistics or equivalently on the context of just one preceding symbol and similarly for higher values of k. For each value of k, there are two possibilities, the static and dynamic model as explained above. For practical reasons, a static model is usually built by collecting statistics over a test *corpus*, which is a collection of text samples representing a particular domain of application (viz. English literature, physical sciences, life sciences, etc.). If one is interested in a more precise static model for a given text, a *semi-static* model is developed in a two-pass process; in the first pass the text is read to collect statistics to compute the model and in the second pass an encoding scheme is developed. Another variation of the model is to use a specific text to prime or seed the model at the beginning and then build the model on top of it as new text files come in.

Independent of the model, there is entropy associated with each file under that model. Shannon's fundamental noiseless source coding theorem says that entropy defines a lower limit of the average number of bits needed to encode the source symbols [56]. The "worst" model from information theoretic point of view is the order (-1) model, the equiprobable model, giving the maximum value H_m of the entropy. Thus, for the 8-bit ASCII code, the value of this entropy is 8 bits. The redundancy R is defined to be the difference[4] between the maximum entropy H_m and the actual entropy H. As we build better and better models by going to higher order k, lower will be the value of entropy yielding a higher value of redundancy. The crux of lossless compression research boils down to developing compression algorithms that can find an encoding of the source using a model with minimum possible entropy and exploiting maximum amount of redundancy. But incorporating a higher order model is computationally expensive and the designer must be aware of other performance metrics such as decoding or decompression complexity (the

[3] This situation gives rise to what is called the zero-frequency problem. One cannot assume the probabilities to be zero because that will imply an infinite number of bits to encode the first few symbols since $-\log 0$ is infinity. There are many different methods of handling this problem but the equiprobability assumption is a fair and practical one.

[4] Shannon's original definition is R/H_m which is the fraction of the structure of the text message determined by the inherent property of the language that governs the generation of specific sequence or words in the text [56].

process of decoding is the reverse of the encoding process in which the redundancy is restored so that the text is again human readable), speed of execution of compression and decompression algorithms and use of additional memory.

Good compression means less storage space to store or archive the data, and it also means less bandwidth requirement to transmit data from source to destination. This is achieved with the use of a *channel* that may be a simple point-to-point connection or a complex entity like the Internet. For the purpose of discussion, assume that the channel is noiseless, that is, it does not introduce error during transmission and it has a *channel capacity* C that is the maximum number of bits that can be transmitted per second. Since entropy H denotes the average number of bits required to encode a symbol, C/H denotes the average number of symbols that can be transmitted over the channel per second [56]. A second fundamental theorem of Shannon says that however clever you may get developing a compression scheme, you will never be able to transmit on average more than C/H symbols per second [56]. In other words, to use the available bandwidth effectively, H should be as low as possible, which means employing a compression scheme that yields minimum BPC.

3. Classification of Lossless Compression Algorithms

The lossless algorithms can be classified into three broad categories: *statistical methods*, *dictionary methods* and *transform based methods*. We will give a very brief review of these methods in this section.

3.1 Statistical Methods

The classical method of statistical coding is Huffman coding [34]. It formalizes the intuitive notion of assigning shorter codes to more frequent symbols and longer codes to infrequent symbols. It is built bottom-up as a binary tree as follows: given the model or the probability distribution of the list of symbols, the probability values are sorted in ascending order. The symbols are then assigned to the leaf nodes of the tree. Two symbols having the two lowest probability values are then combined to form a parent node representing a composite symbol that replaces the two child symbols in the list and whose probability equals the sum of the probabilities of the child symbols. The parent node is then connected to the child nodes by two edges with labels ‘0’ and ‘1’ in any arbitrary order. The process is now repeated with the new list (in which the composite node has replaced the child nodes) until the composite node is the only node remaining in the list. This node is called the root of the tree. The unique sequence of 0’s and 1’s in the path from the root to the leaf node is the Huffman code for the symbol represented by the leaf node. At the decoding end the same binary tree has to be used to decode the symbols from the compressed code. In effect, the tree behaves like a dictionary that has to be transmitted once from the sender to receiver and this constitute an initial overhead of the algorithm. *This overhead is usually ignored in publishing the BPC results for Huffman code in literature*. The Huffman codes for all the symbols have what is called the *prefix property* which is that no code of a symbol is the prefix of the code for another symbol, which makes the code *uniquely decipherable* (*UD*). This allows forming a code for a sequence of symbols by just concatenating the codes of the individual symbols and the decoding process can retrieve the original sequence of symbols without ambiguity. Note that a prefix code is not necessarily a Huffman code nor may obey the Morse's principle and a uniquely decipherable code does not have to be a prefix code, but the beauty of Huffman code is that it is UD, prefix and is also optimum within one bit of the entropy H. Huffman code is indeed optimum if the probabilities are $1/2^k$ where k is a positive integer. There are also Huffman codes called canonical Huffman codes which uses a look up table or dictionary rather than a binary tree for fast encoding and decoding [52,64].

Note in the construction of the Huffman code, we started with a model. Efficiency of the code will depend on how good this model is. If we use higher order models, the entropy will be smaller resulting in shorter average code length. As an example, a word-based Huffman code is constructed by collecting the statistics of words in the text and building a Huffman tree based on the distribution of probabilities of words rather than the letters of the alphabet. It gives very good results but the overhead to store and transmit the tree is considerable. Since the leaf nodes contain all the distinct words in the text, the storage overhead is equal to having an English words dictionary shared between the sender and the receiver. We will return to this point later when we discuss our transforms. Adaptive Huffman codes take longer time for both encoding and decoding because the Huffman tree has to be modified at each step of the process. Finally, Huffman code is sometimes referred to as a variable length code (VLC) because a message of a fixed length may have variable length representations depending on what letters of the alphabet are in the message.

In contrast, the *arithmetic code* encodes a variable size message into a fixed length binary sequence [51]. Arithmetic code is inherently adaptive, does not use any lookup table or dictionary and in theory can be optimal for a machine with unlimited precision of arithmetic computation. The basic idea can be explained as follows: at the beginning the semi-closed interval [0, 1) is partitioned into $|A|$ equal sized semi-closed intervals under the equiprobability assumption and each symbol is assigned one of these intervals. The first symbol, say a_1 of the message can be represented by a point in the real number interval assigned to it. To encode the next symbol a_2

in the message, the new probabilities of all symbols are calculated recognizing that the first symbol has occurred one extra time and then the interval assigned to a_1 is partitioned (as if it were the entire interval) into $|A|$ sub-intervals in accordance with the new probability distribution. The sequence a_1a_2 can now be represented without ambiguity by any real number in the new sub-interval for a_2. The process can be continued for succeeding symbols in the message as long as the intervals are within the specified arithmetic precision of the computer. The number generated at the final iteration is then a code for the message received so far. The machine returns to its initial state and the process is repeated for the next block of symbol. A simpler version of this algorithm could use the same static distribution of probability at each iteration avoiding re-computation of probabilities. The literature on arithmetic coding is vast and the reader is referred to the texts cited above [52,53,64] for further study.

The Huffman and arithmetic coders are sometimes referred to as the *entropy coders*. These methods normally use an order (0) model. If a good model with low entropy can be built external to the algorithms, these algorithms can generate the binary codes very efficiently. One of the most well known modelers is "*prediction by partial match*" (PPM) [20,45]. PPM uses a finite context Order (k) model where k is the maximum context that is specified ahead of execution of the algorithm. The program maintains all the previous occurrences of context at each level of k in a trie-like data structure with associated probability values for each context. If a context at a lower level is a suffix of a context at a higher level, this context is excluded at the lower level. At each level (except the level with $k = -1$), an *escape character* is defined whose frequency of occurrence is assumed to be equal to the number of distinct context encountered at that context level for the purpose of calculating its probability. During the encoding process, the algorithm estimates the probability of the occurrence of the *next character* in the text stream as follows: the algorithm tries to find the current context of maximum length k in the context table or trie. If the context is not found, it passes the probability of the escape character at this level and goes down one level to $k-1$ context table to find the current context of length $k-1$. If it continues to fail to find the context, it may go down ultimately to $k = -1$ level corresponding to equiprobable level for which the probability of any next character is $1/|A|$. If a context of length q, $0 \leqslant q \leqslant k$, is found, then the probability of the next character is estimated to be the product of probabilities of escape characters at levels $k, k-1, \dots, q+1$ multiplied by the probability of the context found at the qth level. This probability value is then passed to the backend entropy coder (arithmetic coder) to obtain the encoding. Note, at the beginning there is no context available so the algorithm assumes a model with $k = -1$. The context lengths are shorter at the early stage of the encoding when only a few contexts have been seen. As the encoding proceeds, longer and longer contexts become available. In one version of PPM, called PPM*, an arbitrary length context is allowed which should give the optimal minimum entropy. In practice a model with $k = 5$ behaves as good as PPM* [19]. Although the algorithm performs very well in terms of high compression ratio or low BPC, it is very computation intensive and slow due to the enormous amount of computation that is needed as each character is processed for maintaining the context information and updating their probabilities.

Dynamic Markov Compression (DMC) is another modeling scheme that is equivalent to finite context model but uses finite state machine to estimate the probabilities of the input symbols which are bits rather than bytes as in PPM [21]. The model starts with a single state machine with only one count of '0' and '1' transitions into itself (the zero frequency state) and then the machine adapts to future inputs by accumulating the transitions with 0's and 1's with revised estimates of probabilities. If a state is used heavily for input transitions (caused either by 1 or 0 input), it is cloned into two states by introducing a new state in which some of the transitions are directed and duplicating the output transitions from the original states for the cloned state in the same ratio of 0 and 1 transitions as the original state. The bit-wise encoding takes longer time and therefore DMC is very slow but the implementation is much simpler than PPM and it has been shown that the PPM and DMC models are equivalent [10].

3.2 Dictionary Methods

The dictionary methods, as the name implies, maintain a *dictionary or codebook* of words or text strings previously encountered in the text input and data compression is achieved by replacing strings in the text by a reference to the string in the dictionary. The dictionary is *dynamic or adaptive* in the sense that it is constructed by adding new strings being read and it allows deletion of less frequently used strings if the size of the dictionary exceeds some limit. It is also possible to use a *static* dictionary like the word dictionary to compress the text. The most widely used compression algorithms (Gzip and Gif) are based on Ziv–Lempel or LZ77 coding [67] in which the text prior to the current symbol constitute the dictionary and a greedy search is initiated to determine whether the characters following the current character have already been encountered in the text before, and if yes, they are replaced by a reference giving its relative starting position in the text. Because of the pattern matching operation the encoding takes longer time but the process has been fine tuned with the use of hashing techniques and special data structures. The decoding process is straightforward and fast because it involves a random access of an array to retrieve the character string. A variation of the LZ77 theme, called the LZ78 coding, includes one extra character to a previously coded string in the encoding scheme. A more popular variant of LZ78 family is the so-called LZW algorithm which leads to widely used *Compress* utility. This method uses a suffix tree to store the strings previously encountered and the text is encoded as a sequence of node numbers in this tree. To encode a string the algorithm will traverse the existing tree as far as possible and a new node is created when the last character in the string fails to traverse a path any more. At this point the last encountered node number is used to compress the string up to that node and a new node is created appending the character that did not lead to a valid path to traverse. In other words, at every step of the process the length of the recognizable strings in the dictionary gets incrementally stretched and is made available to future steps. Many other variants of LZ77 and LZ78 compression family have been reported in the literature (see [52] and [53] for further references).

3.3 Transform Based Methods: The Burrows–Wheeler Transform (BWT)

The word 'transform' has been used to describe this method because the text undergoes a transformation, which performs a permutation of the characters in the text so that characters having similar lexical context will cluster together in the output. Given the text input, the forward Burrows–Wheeler transform [17] forms all cyclic rotations of the characters in the text in the form of a matrix M whose rows are lexicographically sorted (with a specified ordering of the symbols in the alphabet). The last column L of this sorted matrix and an index r of the row where the original text appears in this matrix is the output of the transform. The text could be divided into blocks or the entire text could be considered as one block. The transformation is applied to individual blocks separately, and for this reason the method is referred to as *block sorting* transform [29]. The repetition of the same character in the block might slow down the sorting process; to avoid this, a run-length encoding (RLE) step could be preceded before the transform step. The Bzip2 compression algorithm based on BWT transform uses this step and other steps as follows: the output of the BWT transform stage then undergoes a final transformation using either move-to-front (MTF) [12] encoding or distance coding (DC) [7] which exploits the clustering of characters in the BWT output to generate a sequence of numbers dominated by small values (viz. 0, 1 or 2) out of possible maximum value of $|A|$. This sequence of numbers is then sent to an entropy coder (Huffman or Arithmetic) to obtain the final compressed form. The inverse operation of recovering the original text from the compressed output proceeds by decoding the inverse of the entropy decoder, then inverse of MTF or DC and then an inverse of BWT. The inverse of BWT obtains the original text given (L, r). This is done easily by noting that the first column of M, denoted as F, is simply a sorted version of L. Define an index vector Tr of size $|L|$ such that $Tr[j] = i$ if and only if both $L[j]$ and $F[i]$ denote the kth occurrence of a symbol from A. Since the rows of M are cyclic rotations of the text, the elements of L precede the respective elements of F in the text. Thus $F[Tr[j]]$ cyclically precedes $L[j]$ in the text which leads to a simple algorithm to reconstruct the original text.

3.4 Comparison of Performance of Compression Algorithms

An excellent discussion of performance comparison of the important compression algorithms can be found in [64]. In general, the performance of compression methods depends on the type of data being compressed and there is a tradeoff between compression performance, speed and the use of additional memory resources. The authors report the following results with respect to the Canterbury corpus: In order of increasing compression performance (decreasing BPC), the algorithms can be listed as order zero arithmetic, order zero Huffman giving over 4 BPC; the LZ family of algorithms come next whose performance range from 4 BPC to around 2.5 BPC (Gzip) depending on whether the algorithm is tuned for compression or speed. Order zero word based Huffman (2.95 BPC) is a good contender for this group in terms of compression performance but it is two to three times slower in speed and needs a word dictionary to be shared between the compressor and decompressor. The best performing compression algorithms are: Bzip2 (based on BWT), DMC, and PPM all giving BPC ranging from 2.1 to 2.4. PPM is theoretically the best but is extremely slow as is DMC, bzip2 strikes a middle ground, it gives better compression than Gzip but is not an on-line algorithm because it needs the entire text or blocks of text in memory to perform the BWT transform. LZ77 methods (Gzip) are fastest for decompression, then LZ78 technique, then Huffman coders, and the methods using arithmetic coding are the slowest. Huffman coding is better for static applications whereas arithmetic coding is preferable in adaptive and online coding. Bzip2 decodes faster than most of other methods and it achieves good compression as well. A lot of new research on Bzip2 (see Section 5) has been carried on recently to push the performance envelope of Bzip2 both in terms of compression ratio and speed. As a result Bzip2 has become a strong contender to replace the popular Gzip and Compress.

New research is going on to improve the compression performance of many of the algorithms. However, these efforts seem to have come to a point of saturation with regard to lowering the compression ratio. To get a significant further improvement in compression, other means like transforming the text before actual compression and use of grammatical and semantic information to improve prediction models should be looked into. Shannon made some experiments with native speakers of English language and estimated that the English language has entropy of around 1.3 BPC [55]. Thus, it seems that lossless text compression research is now confronted with the challenge of bridging a gap of about 0.8 BPC in terms of compression ratio.

Of course, combining compression performance with other performance metric like speed, memory overhead and on-line capabilities seem to pose even a bigger challenge.

4. Transform Based Methods: Star (*) Transform

In this section we present our research on new transformation techniques that can be used as preprocessing steps for the compression algorithms described in the previous section. The basic idea is to transform the text into some intermediate form, which can be compressed with better efficiency. The transformation is designed to exploit the natural redundancy of the language. We have developed a class of such transformations, each giving better compression performance over the previous ones and most of them giving better compression over current and classical compression algorithms discussed in the previous section. We first present a brief description of the first transform called Star Transform (also denoted by *-encoding). We then present four new transforms called LPT, SCLPT, RLPT and LIPT. We conclude the section by introducing yet another new transform (StarNT) which forms a new class of compression algorithms called Starzip. The algorithms use a fixed amount of storage overhead in the form of a word dictionary for the particular corpus of interest and must be shared by the sender and receiver of the compressed files. In comparison, word based Huffman method also make use of a static word dictionary but there are important differences as we will explain later. Because of this similarity, we specifically compare the performance of our preprocessing techniques with that of the word-based Huffman. Typical size of dictionary for the English language is about 0.5 MB and can be downloaded along with application programs. If the compression algorithms are going to be used over and over again, which is true in all practical applications, the amortized storage overhead for the dictionary is negligibly small. We will present experimental results measuring the performance (compression ratio, compression times, and decompression times) of our proposed preprocessing techniques using three corpuses: Calgary, Canterbury [23,22] and Gutenberg corpus [24].

4.1 Star (*) Transform

The basic idea underlying the star transformations is to define a unique signature of a word by replacing letters in the word by a special placeholder character (*) and keeping a minimum number of characters to identify the word uniquely [30]. For an English language dictionary D of size 60,000 words, we observed that we needed at most two characters of the original words to keep their identity intact. In fact, it is not necessary to keep any letters of the original word as long as a unique representation can be defined. The dictionary is divided into sub-dictionaries D_s containing words of length s, $1 \leqslant s \leqslant 22$, because the maximum length of a word in English dictionary is 22 and there are two words of length 1 viz. ‘a’ and ‘I’.

The following encoding scheme is used for the words in D_s: The first word is represented as sequence of s stars. The next 52 words are represented by a sequence of $s - 1$ stars followed by a single letter from the alphabet $\Sigma = (a, b \ldots z, A, B \ldots Z)$. The next 52 words have a similar encoding except that the single letter appears in the last but one position. This will continue until all the letters occupy the first position in the sequence. The following group of words have $s - 2$ *’s and the remaining two positions are taken by unique pairs of letters from the alphabet. This process can be continued to obtain a total of 53 unique encodings which is more than sufficient for English words. A large fraction of these combinations are never used; for example for $s = 2$, there are only 17 words and for $s = 8$, there are about 9000 words in English dictionary. Given such an encoding, the original word can be retrieved from the dictionary that contains a one-to-one mapping between encoded words and original words. The encoding produces an abundance of * characters in the transformed text making it the most frequently occurring character. If the word in the input text is not in the English dictionary (viz. a new word in the lexicon) it will be passed to the transformed text unaltered. The transformed text must also be able to handle special characters, punctuation marks and capitalization. The space character is used as word separator. The character ‘∼’ at the end of an encoded word denotes that the first letter of the input text word is capitalized. The character ‘ ’ denotes that all the characters in the input word are capitalized. A capitalization mask, preceded by the character ‘ˆ’, is placed at the end of encoded word to denote capitalization of characters other than the first letter and all capital letters. The character ‘\’ is used as escape character for encoding the occurrences of ‘*’, ‘∼’, ‘ ’, ‘ˆ’, and ‘\’ in the input text. The transformed text can now be the input to any available lossless text compression algorithm, including Bzip2 where the text undergoes two transformation, first the *-transform and then a BWT transform.

4.2 Class of Length Preserving Transforms (LPT and RLPT)

The Length-Preserving Transform (LPT) [41] was invented to handle the problem that arises due to the use of run-length encoding after BWT transform is applied to the *-transformed output. It is defined as follows: words of length more than four are encoded starting with ‘*’, this allows Bzip2 to strongly predict the space character preceding a ‘*’ character. The last three characters form an encoding of dictionary offset of the corresponding word in this manner: entry $D_i[0]$ is encoded as “zaA”. For entries $D_i[j]$ with $j > 0$, the last character cycles through [A–Z], the second-to-last character cycles through [a–z], the third-to-last character cycles through [z–a].

For words of more than four characters, the characters between the initial '*' and the final three-character-sequence in the word encoding is filled up with a suffix of the string "*...nopqrstuvw*". The string may look arbitrary but note that its order is the same as that of the orders of the letters in the alphabet and the suffix length is exactly 4 minus the length of the word. For instance, the first word of length 10 would be encoded as "*rstuvwzaA*". This method provides a strong local context within each word encoding and its delimiters. In this scheme each character sequence contains a marker ('*') at the beginning, an index at the end, and a fixed sequence of characters in the middle. The fixed character sequence provides BWT with a strong prediction for each character in the string.

In the further study of BWT and PPM, we found that a skewed distribution of context should have better result because PPM and its alternatives should have fewer entries in the frequency table. This leads to higher probabilities and we have a shorter code length. The Reverse Length-Preserving Transform (RLPT), a modification of LPT, exploits this information. The padding part is simply reversed for RLPT. For example, the first word of length 10 would be encoded as "*wvustrzaA*". The test results show that the RLPT plus PPMD, outperforms *-transform and LPT. RLPT combined with compression algorithms of Huffman, Arithmetic compress, Gzip, Bzip2 also performs better than if RLPT is not used in combination with these algorithms.

4.3 Class of Index Preserving Transforms SCLPT and LIPT

We observed that it is not necessary to keep the word length in encoding and decoding as long as the one-to-one mappings are held between word pairs in original English dictionary and transform dictionary. The major objectives of compression algorithms such as PPM are to be able to predict the next character in the text sequence efficiently by using the deterministic context information. We noted that in LPT, The padding sequence to maintain length information can be uniquely determined by its first character. For example, a padding sequence "*rstuvw*" is determined by '*r*' and it is possible to replace the entire sequence used in LPT by the sequence "*rzAa*" and vice versa. We call this transform SCLPT (Shortened Context LPT).We now have to use a shortened-word dictionary. If we apply LPT along with the PPM algorithm, there should be context entries of the forms "*rstu*" '*v*', '*stu*' '*v*', '*tu*' '*v*', '*u*' '*v*' in the context table and the algorithm will be able to predict '*v*' at length order 5 deterministically. Normally PPMD goes up to order 5 context, so the long sequence of "*rstuvw*" may be broken into shorter contexts in the context trie. In SCLPT, such entries will all be removed and the context trie will be used to reveal the context information for the shortened sequence such as "*rzAa*". The result shows that this method competes with the RLPT plus PPMD combination.It beats RLPT using PPMD in 50% of the files and has a lower average BPC over the test bed. The SCLPT dictionary is only 60% of the size of the other transform dictionary, thus there is about 60% less memory use in conversion and less CPU time consumed. In general, it outperforms the other schemes in the star-encoded family. LPT has an average improvement of 4.4% on Bzip2 and 1.5% over PPMD; RLPT has an average improvement of 4.9% on Bzip2 and 3.4% over PPMD+ [61] using file *paper6* in Calgary corpus [23] as training set. We have similar improvement with PPMD in which no training set is used. The SCLPT has an average improvement of 7.1% on Bzip2 and 3.8% over PPMD+. For Bzip2, SCLPT has the best compression ratio in all the test files. Our results show that SCLPT has the best compression ratio in half of the test files and ranked second in the rest of the files.

A different twist to our transformation comes from the observation that the frequency of occurrence of words in the corpus as well as the predominance of certain lengths of words in English language might play an important role in revealing additional redundancy to be exploited by the backend algorithm. The frequency of occurrence of symbols, k-grams and words in the form of probability models, of course, forms the corner stone of all compression algorithms but none of these algorithms considered the distribution of the length of words directly in the models. We were motivated to consider length of words as an important factor in English text as we gathered word frequency data according to lengths for the Calgary, Canterbury [23,22], and Gutenberg Corpus [24]. A plot showing the total word frequency versus the word length results for all the text files in our test corpus (combined) is shown in Figure 1.

It can be seen that most words lie in the range of length 1 to 10. The maximum number words have length 2 to 4. The word length and word frequency results provided a basis to build context in the transformed text. We call this Length Index Preserving Transform (LIPT). LIPT can be used as an additional component in the

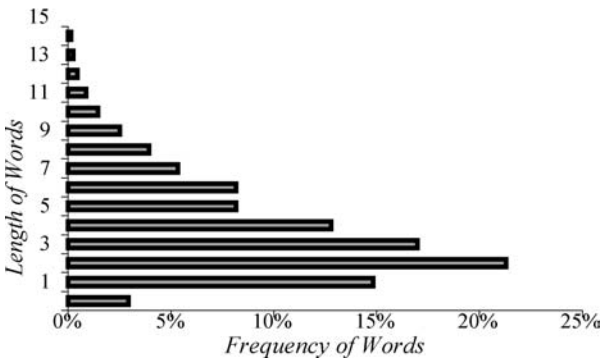


FIG. 1. Frequency of English words versus length of words in the test corpus.

Bzip2 before run length encoding or simply replace it. Compared to the *-transform, we also made a couple of modifications to improve the timing performance of LIPT. For *-transform, searching for a transformed word for a given word in the dictionary during compression and doing the reverse during decompression takes time which degrades the execution times. The situation can be improved by pre-sorting the words lexicographically and doing a binary search on the sorted dictionary both during compression and decompression stages. The other new idea that we introduce is to be able to access the words during decompression phase in a random access manner so as to obtain fast decoding. This is achieved by generating the addresses of the words in the dictionary by using, not numbers, but the letters of the alphabet. We need a maximum of three letters to denote an address and these letters introduce artificial but useful context for the backend algorithms to further exploit the redundancy in the intermediate transformed form of the text. LIPT encoding scheme makes use of recurrence of same length of words in the English language to create context in the transformed text that the entropy coders can exploit.

LIPT uses a static English language dictionary of 59,951 words having a size of around 0.5 MB. LIPT uses transform dictionary of around 0.3 MB. The transformation process requires two files namely English dictionary, which consist of most frequently used words, and a transform dictionary, which contains corresponding transforms for the words in English dictionary. There is one-to-one mapping of word from English to transform dictionary. The words not found in the dictionary are passed as they are. To generate the LIPT dictionary (which is done offline), we need the source English dictionary to be sorted on blocks of lengths and words in each block should be sorted according to frequency of their use.

A dictionary D of words in the corpus is partitioned into disjoint dictionaries D_i, each containing words of length i, where $i = 1, 2, \ldots, n$. Each dictionary D_i is partially sorted according to the frequency of words in the corpus. Then a mapping is used to generate the encoding for all words in each dictionary D_i. $D_i[j]$ denotes the jth word in the dictionary D_i. In LIPT, the word $D_i[j]$, in the dictionary D is transformed as $*c_{\text{len}}[c][c][c]$ (the square brackets denote the optional occurrence of a letter of the alphabet enclosed and are not part of the transformed representation) in the transform dictionary D_{LIPT} where c_{len} stands for a letter in the alphabet [a–z, A–Z] each denoting a corresponding length [1–26, 27–52] and each c is in [a–z, A–Z]. If $j = 0$ then the encoding is $*c_{\text{len}}$. For $j > 0$, the encoding is $*c_{\text{len}}[c][c]$. Thus, for $1 \leqslant j \leqslant 52$ the encoding is $*c_{\text{len}}c$; for $53 \leqslant j \leqslant 2756$ it is $*c_{\text{len}}cc$, and for $2757 \leqslant j \leqslant 140{,}608$ it is $*c_{\text{len}}ccc$. Thus, the 0th word of length 10 in the dictionary D will be encoded as "$*j$" in D_{LIPT}, $D_{10}[1]$ as "$*ja$", $D_{10}[27]$ as "$*jA$", $D_{10}[53]$ as "$*jaa$", $D_{10}[79]$ as "$*jaA$", $D_{10}[105]$ as "$*jba$", $D_{10}[2757]$ as "$*jaaa$", $D_{10}[2809]$ as "$*jaba$", and so on.

4.4 StarNT

There are three considerations that lead us to this transform algorithm.

First, we gathered data of word frequency and length of words information from our collected corpora (all these corpora are publicly available), as depicted in Figure 1. It is clear that more than 82% of the words in English text have lengths greater than three. If we can recode each English word with a representation of no more than three symbols, then we can achieve a certain kind of "*pre-compression*". This consideration can be implemented with a fine-tuned transform encoding algorithm, as is described later.

The second consideration is that the transformed output should be compressible to the backend compression algorithm. In other words, the transformed immediate output should maintain some of the original context information as well as provide some kind of "artificial" but strong context. The reason behind this is that we choose BWT and PPM algorithms as our backend compression tools. Both of them predict symbols based on context information.

Finally, the transformed codewords can be treated as the offset of words in the transform dictionary. Thus, in the transform decoding phase we can use a hash function to achieve O(1) time complexity for searching a word in the dictionary. Based on this consideration, we use a continuously addressed dictionary in our algorithm. In contrast, the dictionary is split into 22 sub-blocks in LIPT [49]. Results show that the new transform is better than LIPT not only in time complexity but also in compression performance.

The performance of search operation in the dictionary is the key for fast transform encoding. We have used a special data structure, *ternary search tree* to achieve this objective. we will discuss the ternary tree technique and its applications in Section 7 as well as the possible parallel processing using ternary suffix tree in Section 8.2.

4.4.1 Dictionary Mapping

The dictionary used in this experiment is prepared in advance, and shared by both the transform encoding module and the transform decoding module. In view of the three considerations mentioned in Section 3.2, words in the dictionary D are sorted using the following rules:

- Most frequently used words are listed at the beginning of the dictionary. There are 312 words in this group.
- The remaining words are stored in D according to their lengths. Words with longer lengths are stored after words with shorter lengths. Words with same length are sorted according to their frequency of occurrence.

- To achieve better compression performance for the backend data compression algorithm, only letters $[a..zA..Z]$ are used to represent the codeword.

With the ordering specified above, each word in D is assigned a corresponding codeword. The first 26 words in D are assigned 'a', 'b', ..., 'z' as their codewords. The next 26 words are assigned 'A', 'B', ..., 'Z'. The 53rd word is assigned "aa", 54th "ab". Following this order, "ZZ" is assigned to the 2756th word in D. The 2757th word in D is assigned "aaa", the following 2758th word is assigned "aab", and so on. Hence, the most frequently occurred words are assigned codewords form 'a' to "eZ". Using this mapping mechanism, there are totally $52 + 52 * 52 + 52 * 52 * 52 = 143{,}364$ words in D.

4.4.2 The StarNT Transform

In the transform encoding module, the shared static dictionary is read into main memory and the corresponding ternary search tree is constructed. A *replacer* is initiated to read in the input text character by character, which performs the replace operation when it recognizes that the string of a certain length of input symbols (or the lower case form of this string sequence) exists in the dictionary. Then it outputs the corresponding codeword (appended with a special symbol if needed) and continues. If the input symbol sequence does not exist in the dictionary, it will be output with a prefix escape character '*'. Currently only single words are stored in the dictionary. For future implementation, the transform module may contain phrases or sequence of words with all kinds of symbols (such as capital letters, lower case letters, blank symbols, punctuations, etc.) in the dictionary.

The proposed new transform differs from our earlier Star-family transforms with respect to the meaning of the character '*'. Originally it was used to indicate the beginning of a codeword. In this transform, it denotes that the following word does not exist in the dictionary D. The main reason for this change is to minimize the size of the transformed intermediate text file, because smaller size can expedite the backend encoding/decoding.

Currently only lower-case words are stored in the dictionary D. Special operations were designed to handle the first-letter capitalized words and all-letter capitalized words. The character '~' appended at the end of an encoded word denotes that the first letter of the input text word is capitalized. The appended character ' ' ' denotes that all letters of the word are capitalized. The character '\' is used as escape character for encoding the occurrences of '*', '~', ' ' ', and '\' in the input text.

The transform decoding module performs the inverse operation of the transform-encoding module. The escape character and special symbols ('*', '~', ' ' ', and '\') are recognized and processed, and the transformed words are replaced with their original forms, which are stored continuously in a pre-allocated memory to minimize the memory usage. And their addresses are stored in an array sequentially. There is a very important property of the dictionary mapping mechanism that can be used to achieve O(1) time complexity to search a word in the dictionary. Because codewords in the dictionary are assigned sequentially, and only letters $[a..zA..Z]$ are used, these codewords can be interpreted as the address in the dictionary. In our implementation, we use a very simple one-to-one mapping to calculate the index of the corresponding original words in the array that stores the address of the dictionary words.

4.4.3 Performance Evaluation

Our experiments were carried out on a 360 MHz Ultra Sparc-IIi Sun Microsystems machine housing SunOS 5.7 Generic_106541-04. We choose Bzip2 (−9), PPMD (order 5) and Gzip (−9) as the backend compression tool. Facilitated with our proposed transform algorithm, Bzip2 −9, Gzip −9 and PPMD all achieve a better compression performance in comparison to most of the recent efforts based on PPM and BWT. Figure 2 shows that, for Calgary corpus, Canterbury corpus and Gutenberg corpus, StarNT achieves an average improvement in compression ratio of 11.2% over Bzip2 −9, 16.4% over Gzip −9, and 10.2% over PPMD. *Ternary Search Tree* is used in the encoding module. With a finely tuned dictionary mapping mechanism, we can find a word in the dictionary at time complexity O(1) in the transform decoding module. Results shows that for all corpora, the average compression time using the transform algorithm with Bzip2 −9, Gzip −9 and PPMD is 28.1% slower, 50.4% slower and 21.2% faster compared to the original Bzip2 −9, Gzip −9 and PPMD respectively. The average decompression time using the new transform algo-

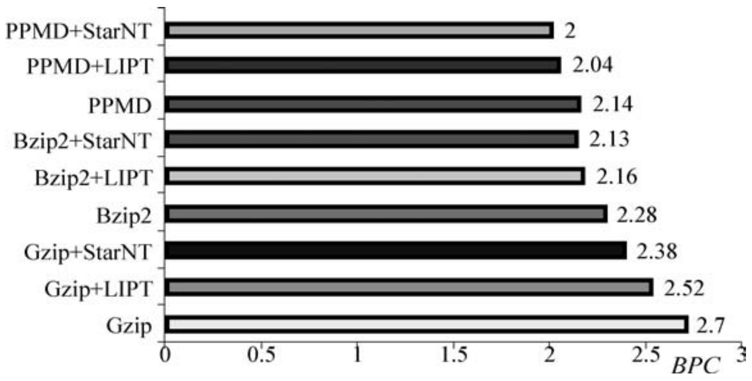


FIG. 2. Compression ratio with/without transform.

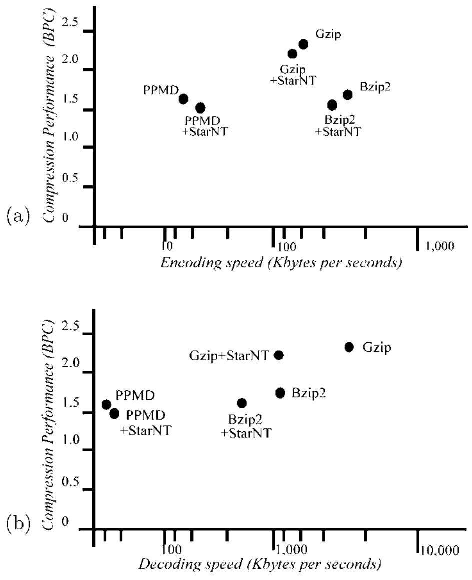


FIG. 3. Compression effectiveness versus (a) Compression speed, (b) Decompression speed.

rithm with Bzip2 −9, Gzip −9 and PPMD is 100% slower, 600% slower and 18.6% faster compared to the original Bzip2 −9, Gzip −9 and PPMD respectively. Figure 3 illustrates the compression ratio vs. compression/decompression speed for the different algorithms. However, since the decoding process is fairly fast, this increase is negligible.We draw a significant conclusion that Bzip2 in conjunction with StarNT is better than both Gzip and PPMD both in time complexity and compression performance.

Based on this transform, we developed StarZip, a domain-specific lossless text compression utility for archival storage and retrieval. StarZip uses specific dictionaries for specific domains. In our experiment, we created five corpora from publicly available website, and derived five domain-specific dictionaries. Results show that the average BPC improved 13% over bzip2 −9, 19% over Gzip −9, and 10% over PPMD for these five corpora.

5. Compressed Domain Pattern Matching

The pattern matching problem is to find the occurrences of a given pattern in a given text string. The *exact string matching problem* is to check for the existence of a substring of the text that is an exact replica of the pattern string. Various algorithms have been proposed for exact pattern matching [39,14]. With the increasing amount of text data available, most of these data are now typically stored in a compressed format. Thus, efforts have been made to address the *compressed pattern matching* (CPM) problem. Given a text string T, a search pattern P, and Z the compressed representation of T, the problem is to locate the occurrences of P in T with minimal (or no) decompression of Z. Different methods have been proposed [4,44,28,27,32, 50]. The motivation includes the potential reduction of the delay in response time introduced by the initial decompression of the data, and the possible elimination of the time required for later compression. Another motivation is the fact that, with the compact representation of data in compressed form, manipulating such smaller amounts of data directly will lead to some speedup in certain types of processing on the data.

Initial attempts at compressed pattern matching were directed towards compression schemes based on the Lempel–Ziv (LZ, for short) family of algorithms [67, 68] where algorithms have been proposed that can search for a pattern in an LZ77-compressed text string in $\mathrm{O}(n\log^2(u/n)+m)$ time, where $m=|P|$, $u=|T|$, and $n=|Z|$ [28]. The focus on LZ might be attributed to the wide availability of LZ-based compression schemes on major computing platforms. For example, Gzip and Compress (UNIX), Pkzip (MSDOS) and Winzip (MS WINDOWS) are all based on the LZ algorithm. An off-line search index—the *q-gram index*, has been reported for the LZ78 algorithm [35]. Navarro and Raffinot [50] proposed a hybrid compression between LZ77 and LZ78 that can be searched in $\mathrm{O}(\min(u, n\log m)+r)$ average time, where r is the total number of matches. Amir [4] proposed an algorithm which runs in $\mathrm{O}(n\log m+m)$—"almost optimal" or in $\mathrm{O}(n+m^2)$, depending on how much *extra space* is being used, to search the first occurrence of a pattern in the LZW encoded files. Barcaccia [9] extended Amir's work to an LZ compression method that uses the so-called "ID heuristic". Among the above LZ-based CPM algorithms, Amir's algorithm has been well-recognized, not only because of its "almost-optimal" or "near optimal" performance, but also because it works directly with the LZW compression without having to modify it—this is a great advantage because keeping the popular implementations of the LZW and avoiding the re-compression of the LZW-compressed files are highly desirable. In [37,36], Amir's algorithm has been extended by Kida for multiple-pattern matching by using Aho–Corasick algorithm and it takes $\mathrm{O}(n+m^2+t+r)$ time and $\mathrm{O}(m^2+t)$ extra space for the algorithm to report the pattern occurrences, where t is the LZW trie size.

Methods for pattern matching directly on data compressed with non-LZ methods have also been proposed. Bunke and Csirik [15,16] proposed methods that can search for patterns in run-length encoded files in $\mathrm{O}(um_c)$ or $\mathrm{O}(nu+mm_c)$, when m_c is the length of the pattern P when it is compressed. Other methods for performing pattern matching directly on run-length encoding (RLE) files have been studied in [43] and [6]. In [69,48], $O(n + m\sqrt{u})$ algorithms were proposed for searching Huffman-encoded files, while [58] proposed a method to search directly on files compressed with the *antidictionaries* proposed in [25]. Some authors have also proposed special compression schemes that will facilitate later pattern matching directly on the compressed file [44,57]. The BWT provides a lexicographic ordering of the input text as part of its inverse transformation process (see Section 3.3). It forms a middle ground between the superior compression ability of the PPM*, and the fast compression time of the LZ-family. This makes the BWT an important approach to data compression, especially where there is need for significant compression ratios with fast compression or decompression. Algorithms have been proposed that perform exact pattern matching based on a fast q-gram intersection of segments from the pattern P and the text T in $O(u + m\log(u/|\Sigma|))$ time on average [2,1,11]. The k-mismatch problem has been solved in $O(uk\log(u/|\Sigma|))$ time, and the k-approximate matching problem in $O(|\Sigma|\log|\Sigma| + m^2/k + m\log(u/|\Sigma|) + \alpha k)$ time on average ($\alpha \leqslant u$) [65], where $u = |T|$ is the size of the text, $m = |P|$ is the size of the pattern, and Σ is the symbol alphabet. Each algorithm requires $O(u)$ auxiliary arrays, which are constructed in $O(u)$ time and space.

In general, the LZ-family are relatively fast, although they do not produce the best results in terms of compression ratio. Improvements are also pursued to produce better compression while keeping the efficiency. In the rest of the section, we will describe our recent work on pattern matching algorithms based on LZW compressed text. We will also propose a modified LZW algorithm to facilitate good compression, fast searching, random access, partial decoding in the next section.

5.1 Compressed Pattern Matching with LZW

We first introduce Amir's compressed pattern matching algorithm. We then give two multiple pattern matching algorithms, one of which is by Kida and it is derived from Amir's algorithm; another one is our research.

5.1.1 LZW Compression

Let $S = c_1c_2c_3\ldots c_u$ be the uncompressed text of length u over alphabet $\Sigma = \{a_1, a_2, a_3, \ldots, a_q\}$, where q is the size of the alphabet. We denote the LZW compressed format of S as $S.Z$ and each code in $S.Z$ as $S.Z[i]$, where $1 \leqslant i \leqslant n$.

We also denote the pattern as $P = p_1 p_2 p_3 \ldots p_m$, where m is the length of pattern P.

The LZW compression algorithm uses a tree-like data structure called a "trie" to store the dictionary generated during the compression processes. Each node on the trie contains:

- A node number: a unique ID in the range $[0, n + q]$; thus, "a node with node number N" and "node N" are sometimes used interchangeably.
- A label: a symbol from the alphabet.
- A chunk: the string that the node represents. It is simply the string consisting of the labels on the path from the root to this node.

For example, in Figure 4, the leftmost leaf node's node number is 8; its label is 'b'; and its chunk is "aab".

At the beginning of the trie construction, the trie has $q + 1$ nodes, including a root node with node number 0 and a NULL label and q child nodes each labeled with a unique symbol from the alphabet. During compression, LZW algorithm scans the text and finds the longest sub-string that appears in the trie as the chunk of some node N and outputs N to $S.Z$. The trie then grows by adding a new node under N and the new node's label is the next un-encoded symbol in the text. Obviously, the new node's chunk is node N's chunk appended by the new node's label. At the end of the compression, there are $n + q$ nodes in the trie.

An example of the trie structure is illustrated in Figure 4. The decoder constructs the same trie and uses it to decode $S.Z$. Both the compression and decompression (and thus the trie construction) can be done in time $O(u)$.

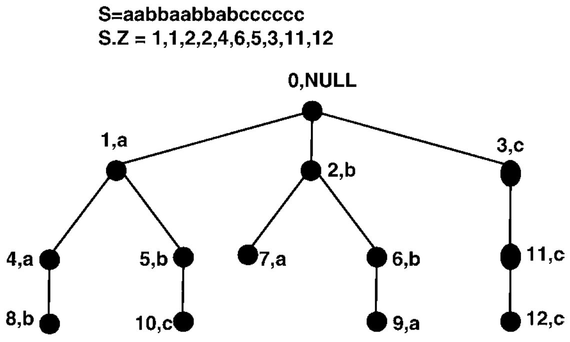


FIG. 4. LZW trie structure.

5.1.2 Amir's Algorithm

Amir's algorithm performs the pattern matching directly on the trie, which can be constructed from the compressed form $S.Z$ in time $O(n)$ without explicitly decoding $S.Z$.

Observation: When the decoder receives code $S.Z[i]$, assuming $S.Z[i-1]$ has already been received in the previous step ($2 \leqslant i \leqslant n$), a new node is created and added as a child of node $S.Z[i-1]$. The node number of the new node is $i-1+q$ and the label of the new node is the first symbol of node $S.Z[i]$'s chunk. For $S.Z[1]$, no new node is created.

In Amir's algorithm, the following terms of a node in the trie are defined with respect to the pattern:

- A chunk is a *prefix chunk* if it ends with a non-empty pattern prefix; the representing prefix of a prefix chunk is the longest pattern prefix it ends with.
- A chunk is a *suffix chunk* if it begins with a non-empty pattern suffix; the representing suffix of a suffix chunk is the longest pattern suffix it begins with.
- A chunk is an *internal chunk* if it is an internal sub-string of the pattern, i.e., the chunk is $p_i \dots p_j$ for $i > 1$ and $j \leqslant m$. If $j = m$, the internal chunk also becomes a suffix chunk.

If a node's chunk is prefix chunk, suffix chunk or internal chunk, the node is called a prefix node, suffix node or internal node, respectively. To represent a node's representing prefix, a prefix number is defined for the node to indicate the length of the representing prefix; Similarly, to represent a node's representing suffix, a suffix number is defined for the node to indicate the length of the representing suffix; To represent a node's internal chunk status, an internal range $[i, j]$ is defined to indicate that the node's chunk is an internal chunk $p_i \dots p_j$. The prefix number, suffix number and internal range are computed for a node when the node is being added to the trie:

(a) The new node's internal range is computed as function $Q_3(I_P, a)$, where I_P is the internal range of its parent and a is its label.
(b) If the result from step (a) tells that the new node is not only an internal node, but also a suffix node (i.e., $j = m$), set its suffix number as $m-i+1$. Otherwise the new node's suffix number is set as its parent's suffix number.
(c) The new node's prefix number is computed as function $Q_1(P_P, a)$, where P_P is the prefix number of its parent and a is its label.

When the new node's label is not in the pattern, the above computations can be done easily; when the new node's label is in the pattern, the operands of function Q_1 and Q_3 are all sub-strings of the pattern. Since the number of the sub-strings of a given pattern is finite, we can pre-compute the results for all possible combinations of the operands. In [4], this pre-processing of the pattern is done by Knuth–Morris–Pratt automaton [39] and the suffix-trie. The pre-processing takes time $O(m^2)$. Once the preprocessing is done, (a)–(c) can be answered in constant time.

The pattern matching is performed simultaneously as the trie grows, as described in the following algorithm:

Pre-process the pattern.

Initialize trie and set global variable $\mathit{Prefix} = \mathit{NULL}$.

For $i = 2$ to n, perform the followings after receiving code $S.Z[i]$ (we will refer it as the current node).

Step 1. Add a new node in the trie and compute the new node's prefix number, suffix number and internal range.

Step 2. Pattern matching:

(a) If $\mathit{Prefix} = \mathit{NULL}$, set variable Prefix as current node's prefix number.
(b) If $\mathit{Prefix} \neq \mathit{NULL}$ and the current node is a suffix node, check the pattern occurrence from *Prefix* and the current node's representing suffix S_P; this checking is defined as function $Q_2(\mathit{Prefix}, S_P)$.
(c) If $\mathit{Prefix} \neq \mathit{NULL}$ and the current node's chunk is an internal chunk, compute Prefix as $Q_1(\mathit{Prefix}, I_P)$ where I_P is the current node's internal range.
(d) If $\mathit{Prefix} \neq \mathit{NULL}$ and the current node's chunk is not an internal chunk, set *Prefix* as the current node's prefix number.

Note that function Q_2 can also be pre-processed by the KMP automata and the suffix trie of the pattern because both its two operands are sub-strings of the pattern. The algorithm has a total of $O(n + m^2)$ time and space complexity. A tradeoff between the time and space alternatively gives a $O(n \log m + m)$ time and $O(n + m)$ space algorithm.

Amir's algorithm cannot be directly used in a practical application because it reports only the first occurrence of the pattern. Besides, multiple-pattern matching is not addressed in the original algorithm. The algorithm has been extended to report all occurrences of the pattern and multiple patterns. Next, we will give two of such extensions. One is from Kida et al. and the other one is our work. The comparison of these two algorithms is also given.

5.1.3 Kida's Multiple Pattern Matching Algorithm

Aho–Corasick algorithm [3] is a classic solution for multiple-pattern matching. Pattern matching is performed using a trie structure, which is also called an Aho–Corasick automaton. The AC automaton for patterns *he, she, his, hers* is shown in Figure 5.

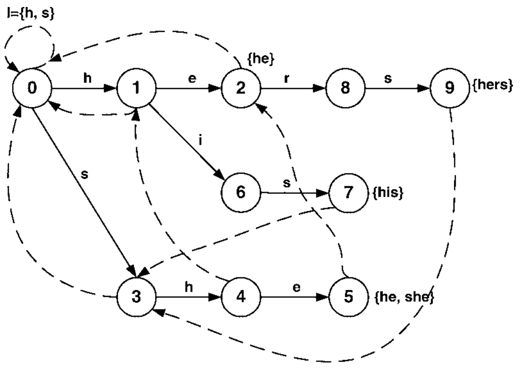


FIG. 5. An example of the Aho–Corasick automaton.

It can be seen that in the automaton, each edge is labeled with a symbol and edges coming out from a node (we will refer the nodes as *states* in the remaining of this paper) have different labels. If we define $R(v)$ of a state v as the concatenation of labels along the path from the root to state v, the following is true: for each pattern P in the pattern set, there is a state v such that $R(v) = P$, and this state is called a final state; each state represents a prefix of some pattern; for each leaf state v, there is some pattern P in the pattern set so that $R(v) = P$. For instance, pattern "he" is represented by state 2 and leaf state 9 represents pattern "hers". Both states 2 and states 9 are final states.

For each state, a $goto(v, a)$ function is defined which gives the state to enter from state v by matching symbol a. For instance, $goto(1, e) = 2$ means that the state to enter is 2 if we match symbol e from state 1. A failure link $f(v)$ (indicated as dotted line in Figure 5) is also defined for each state and it gives the state to enter when mismatch happens. The failure link $f(v)$ points to a state that represents the longest proper suffix of $R(v)$. Thus, when mismatch happens, by following the failure link, we will be able to continue the matching process since the state to enter also corresponds to a prefix of some pattern. Finally, a $out(v)$ function is defined for state v that gives the patterns recognized when entering that state.

When searching the patterns, the AC automaton starts from the root of the automaton and processes one symbol from the input text S at a time. Through the *goto* functions and the failure links, the automaton changes its current state from one to another. The automaton reports the pattern occurrence if a final state is entered. The construction of the automaton takes time and space $O(m)$ where m is the total length of the patterns. The search takes time $O(u)$ where u is the size of the input text. Thus, the overall computational time of Aho–Corasick algorithm is $O(u + m)$.

The basic idea of Kida's algorithm is to have a AC automaton that is able to process the compressed symbols, specifically, the LZW trie node numbers. The algorithm first constructs the GST (General Suffix Trie) and the AC automaton of the patterns. The algorithm then relies on two main functions: *Next*(q, s) and *Output*(q, s) that computes the next AC state from state q by taking string s and outputs all patterns that ends in $q.s$, respectively. Note that string s is limited to only those patterns that are represented as LZW trie nodes.

Function *Next*(q, s) is computed as:

$$N(q, s) = \begin{cases} N_1(q, s) & \text{if } s \text{ is substring of any pattern;} \\ \delta(\varepsilon, s) & \text{otherwise} \end{cases}$$

where N_1 is a two-dimensional table that gives the result in constant time. Since the number of AC states is $O(m)$ and the number of sub-strings of the patterns is $O(m^2)$, the table can be constructed in time and space $O(m^3)$ using the AC automaton and the GST. The time and space can be further improved to $O(m^2)$. $\delta(\varepsilon, s)$ computes the state from the initial state by taking string s and it can be incrementally computed in constant time when a new node is added to the LZW trie. Thus, the overall time and space for δ is $O(t)$ where t is the LZW trie size.

Function *Output*(q, s) is broken into two parts:

$$\textit{Output}(q, s) = \textit{Output}(q, \check{s}) \cup A(s)$$

where $\check{s}$ is the longest prefix of s such that $\check{s}$ is a suffix of any pattern. It can be computed incrementally when constructing the trie. The first part can be answered by combing all outputs of the states traversed from q by taking $\check{s}$ and it takes time proportional to the number of pattern occurrences. A two-dimensional table N_2 is used to answer the states traversed. Since $\check{s}$ is a suffix of a pattern and the total number of suffixes is $O(m)$, the table can be pre-computed in time and space $O(m^2)$. The second part $A(s)$ is basically the set of patterns that also begin in $q.s$ and it can be represented by $\check{s}$ and $\tilde{s}$, which is the longest proper prefix of s whose suffix is a suffix of any pattern. Like $\check{s}$, $\tilde{s}$ can be computed incrementally when the trie is constructed. Therefore, it takes $O(t)$ time and space for $A(s)$ and it takes $O(m^2 + t)$ time and space for the construction of the output function. As we mentioned above, the output function enumerates the patterns in time proportional to the number of pattern occurrences, i.e., $O(r)$.

Overall, Kida's algorithm takes time $O(n + m^2 + t + r)$ and it needs extra space of $O(m^2 + t)$.

5.1.4 A Novel Compressed Domain Multiple Pattern Matching Algorithm

In this section, we present a multiple pattern matching algorithm with LZW. This is achieved by constructing a LZW trie with a *state-transition list* constructed

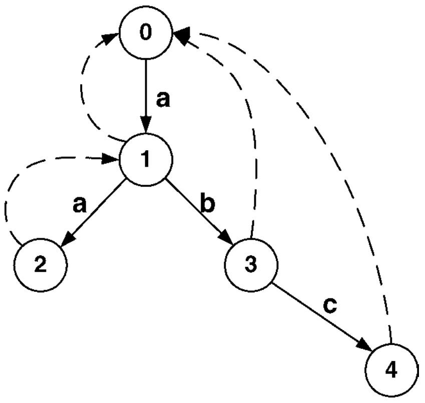


FIG. 6. The AC automaton for patterns *aa*, *ab* and *abc*.

for each node. Consider the same example shown in Figure 4, where the text is: "*aabbaabbabcccccc*" and the compressed data is 1, 1, 2, 2, 4, 6, 5, 3, 11, 12 and we assume the patterns are: "*aa*", "*ab*", "*abc*". In the proposed method, the AC automaton for the patterns is first constructed, as shown in Figure 6.

After the AC automaton has been constructed, we then use it to create the state-transition list for each node in the initial LZW trie. Each entry in the state-transition list of a node is in the form $v_1 - v_2$, which indicates that state v_1 will be changed to state v_2 if the chunk of the current node is fed to the AC automaton. For the initial LZW trie, since each node's chunk is simply its label, we may immediately create the state-transition list for a node by feeding its label to the AC automaton. Figure 7 shows the initial LZW trie with the state-transition list. For example, the state-transition list for node 1 is 0-1, 1-2, 2-2, 3-1, 4-1. It means that when node 1 is

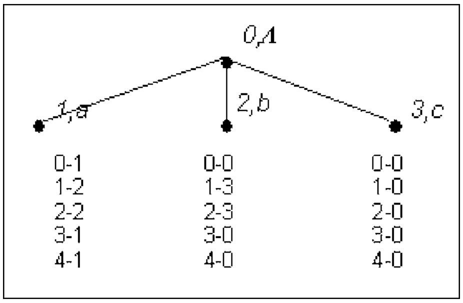


FIG. 7. The initial LZW trie with state-transition lists.

received, if the current state is 0, the next state will be 1; if the current state is 1, the next state will be 2; and so on. Since the *from* state v_1 exhausts all possible states and they are ordered from the lowest to the highest numbered state, the state-transition list can be written as 1, 2, 2, 1, 1 for node 1.

The algorithm then linearly scans the compressed data. Each time a code is received, a new node is added to the LZW trie, until there is no more space for new node. As have been discussed above, the LZW trie can be reconstructed without explicitly decoding the data. The new node's state-transition list, instead of being obtained by feeding its chunk to the AC automaton, can be computed directly from its label and its parent. For example, when node 4 is added under node 1, we do not have to construct the state-transition list by applying its chunk aa to the AC automaton. Instead, we could simply apply its label a to the AC automaton starting from the corresponding v_2 state in its parent's state-transition list. Thus, we obtain the new state-transition list: 2; 2; 2; 2; 2 because the states 1, 2, 2, 1 and 1, when receiving symbol a, will be changed to states 2, 2, 2, 2 and 2, respectively.

Finally, the complete trie (we ignored the state-transition information for some nodes because those nodes are not referenced during the compression) is shown in Figure 8.

In the figures, the state-transition list is stored directly under a LZW trie node to better describe our idea. In actual implementation, instead of having a list for each node, a single table is constructed for the whole trie and we call this table the *state-transition table*. Each row of the state-transition table corresponds to one state in the AC automaton; each column of the state-transition table corresponds to one node in the LZW trie.

We now show how to perform pattern matching for the same example using the state-transition table. At the beginning of the search, the current state is set as 0.

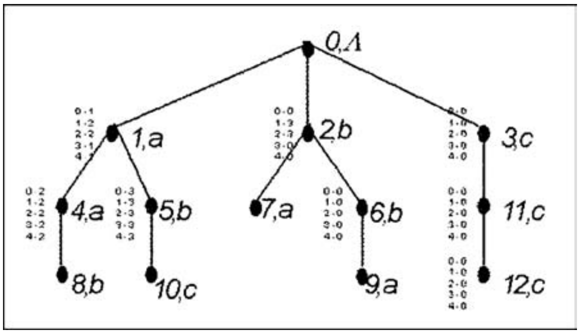


FIG. 8. The LZW trie with state-transition lists.

When node 1 is received, using the state-transition table, the current state is changed to 1; when the second node 1 is received, the station-transition table indicates that the current state is changed to 2. Thus, we are able to report pattern occurrences whenever a final state is entered.

However, if we follow the above operation, an occurrence of ab is missing. The reason is that, when we compute the third entry (v_1 is 2) of the state-transition list of node 6, by matching the label b from the parent node's corresponding v_2 state, state 3, the computed state is 0, which is not a final state. However, the intermediate state, state 3, is a final state. Thus, a final state is "skipped" during the transitions and this is why the second occurrence ab is not reported. A final-state skipping happens only if the corresponding entry in a node's ancestor node's transition list is a final state. Thus, the problem can be fixed by adding a flag for each entry in the state-transition list of a node and it is set as true if the corresponding entry of the node's ancestor is a final state. This flag is inheritable by a node's offspring. During the search, if the flag is on, the current node's chunk needs to be processed symbol by symbol by the Aho–Corasick automaton so we will not miss any pattern occurrence.

Analysis and Comparison. The state-transition table construction takes time and space $O(mt)$ where t is the LZW trie size and m is the total length of the patterns. The search time depends on how many final states are skipped during the search process and is proportional to r, the number of occurrences of the pattern. Thus, the search takes $O(n + r)$ time and the total processing time of our algorithm is $O(n + mt + r)$. The extra space used in our algorithm is solely the cost on the state-transition table, i.e., $O(mt)$, which is independent of the file size.

In a practical implementation, the size of the dictionary is constant. Therefore, the time and space complexity of our algorithm will be $O(n + m + r)$ and $O(m)$, respectively, contrasting to Kida's $O(n + m^2 + r)$ time and $O(m^2 + t)$ space algorithm.

6. Text Information Retrieval on Compressed Text

6.1 Introduction

It has been pointed out that compression is a key for next-generation text retrieval systems [69]. At first glance, it appears that compression is not important since the cost of the storage media is getting lower and lower, and the capacity and performance are getting better. However, a good compression method will facilitate efficient search and retrieval in comparison to some operations performed on raw texts. Besides, to facilitate information retrieval, texts are sometime preprocessed by adding meta-data into the raw text. Therefore the size of file is much larger than its initial size. An example is the XML, which has become more and more popular for multimedia data due to its modular, structural, and dynamic property. This illustrates an additional reason for data compression besides the absolute amount of new data and the bandwidth requirement for transmission. However, for large compressed database, efficient information retrieval is an important problem. In this section we will discuss the information retrieval system for the compressed database.

Given a query using keywords, the most obvious approach to search compressed database is to decompress-then-search, which is not very efficient in terms of search time. Compressed domain pattern matching is an efficient method but it tends to solve the problem on per-file basis. Most practical information retrieval systems build an index or inversion table [8,40,46,64], combined with document frequency using the key words. The documents are ranked using some standard to achieve good precision and recall. Relevant feedback may also help to refine the query to have more accurate results. Such systems using inverted file will need the use of larger size index table with increase of file size. It is possible to adopt an approach to information retrieval which uses compressed domain pattern search, not on the original text, but on the compressed inverted file yielding pointers to documents that are also in compressed form.

The amount of storage used and the efficiency of indexing and searching are major design considerations for an information retrieval system. Usually, there is a tradeoff between the compression performance and the retrieval efficiency. Some simple compression schemes such as run-length encoding provide easy indexing and fast search but a low compression ratio [16,15]. Methods have been proposed for compressed domain search and retrieval using Burrows–Wheeler transforms (BWT), Prediction by Partial Matching (PPM), and other techniques (see Section 2). Although PPM and BWT provide the best compression ratio, the retrieval is hard to perform directly or through indexing on compressed files. Word Huffman based coding schemes [18,69, 46,64] provide a better balance of compression ratio and performance of indexing and searching.

Depending on the application, the target for the retrieval operation may vary. Usually, only a small portion of the collection that is relevant to the query need to be retrieved. For example, the user may ask to retrieve a single record, or a paragraph, or a whole document. It is unnecessary to decompress the whole database and then locate the portion that is retrieved. Using a single level document partitioning system may not be the best answer. We propose to add context boundary tags into the document. Different tags indicate different granularity. Decoding will be performed within the boundaries.

6.1.1 Components of a Compressed Domain Retrieval System

The major concerns of compression method for the retrieval purpose, ranked roughly by their importance are: (a) random access and fast (partial) decompression; (b) fast and space-efficient indexing; and (c) good compression ratio. The compression time is not a major concern since the retrieval system usually performs off-line preprocessing to build the index files for the whole corpus. Besides the searching algorithm, random and fast access to the compressed data is critical to the response time for the user query in a text retrieval system. A typical text retrieval system is constructed as follows. First, the keywords are collected from the text database off-line and an inverted index file is built. Each entry in the index points to all the documents that contain the keyword. A popular document ranking scheme is based on the keyword frequency *tf* and inverted document frequency *idf* [64]. When a query is given, the search engine will match the words in the inverted index file with the query. Then the document ranking information is computed according to a certain logic and/or frequency rules, for example by the similarity measurement to obtain the search results that point to the target documents. Finally, only the selected documents are retrieved. The right half of Figure 9 shows the structure of a traditional text retrieval system. We will first discuss the structure and algorithms for compressed text

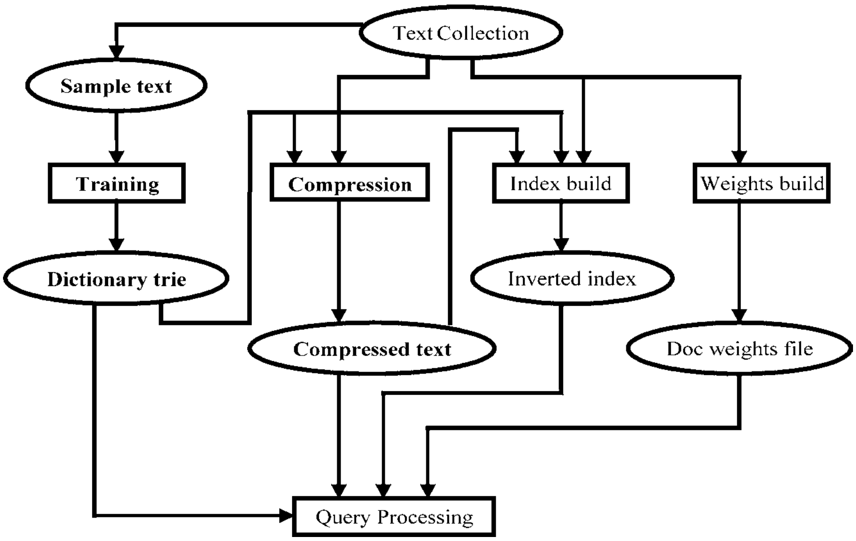


FIG. 9. (Compressed) text retrieval system.

retrieval. Then we will explain the components on the left hand side of Figure 9 that relate to compressed domain text retrieval system.

6.1.2 Flexible Compressed Text Retrieval System Using Modified LZW

We propose a new text retrieval scheme based on the widely used LZW compression algorithm incorporating random access property through inverted index file and partial decoding [66]. The algorithm is based on an off-line preprocessing of the context trie and is specifically designed for searching library databases using key words and phrases. The algorithm uses a public trie or a "dictionary" which is trained for efficiency using a pre-selected standard corpus. The algorithm gives a better compression ratio when compared to the original LZW algorithm. Its byte level coding provides the ability of easy parallel processing for both compression and decompression and made them independent from each other.

6.2 Modified LZW Algorithms

To solve the problem of random access and partial decoding and provide high flexibility for indexing and searching, we need to remove the correlation between the current code and the history information. Dictionary based method may be a possible solution. Huffman coding provides a tree with every file so that each symbol has a unique code. But it has a low compression ratio and is used as an entropy coder at the last stage of a compression system. Word based Huffman compression uses a model (a binary tree); Canonic Huffman uses an implicit dictionary shared between the encoder and the decoder while LZW compression is universal where the model is built dynamically and adaptively [18,46,64]. In this section, we describe a modified LZW approach that aims at partial decoding on compressed files while maintaining a good compression ratio. The dictionary is language independent and is built from the text. A tag system is suggested to output the retrieval results in different scope. Our fixed length coding scheme also helps to build the index and defines the boundary of the text segments in different granularity. Parallel access to the compressed text is also easy to perform.

6.2.1 The LZW Algorithm

The LZW algorithm [63] is one of the many variations of the Ziv–Lempel methods. The LZW algorithm is an adaptive dictionary-based approach that builds a dictionary based on the document that is being compressed. The LZW encoder begins with an initial dictionary consisting of all the symbols in the alphabet, and builds the dictionary by adding new symbols to the dictionary as it encounters new symbols in

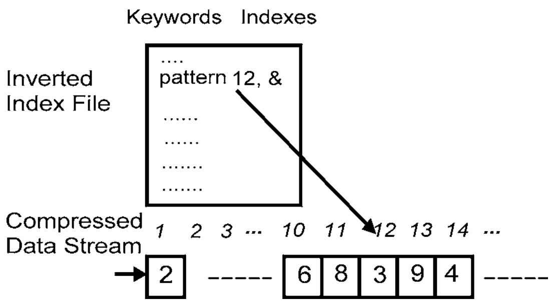


FIG. 10. Illustration of indexing for an LZW compressed file.

the text that is being compressed. The dictionary construction process is completely reversible. The decoder can rebuild the dictionary as it decodes the compressed text. The dictionary is actually represented by a trie. Each edge in the trie is labeled by the symbols, and the path label from the root to any node in the trie gives the substring corresponding to the node. Initially, each node in the trie represents one symbol in the alphabet. Nodes representing patterns are added to the trie as new patterns are encountered. As stated in Section 5, Amir proposed an algorithm to find the first occurrence of a pattern in an LZW-compressed file [5]. The pattern prefix, suffix, or internal substring is detected and checked to determine the occurrence of the pattern. The pattern is searched at the stage of rebuilding the dictionary trie to avoid total decompression. Obviously, this method cannot satisfy the request to find multiple occurrences in the large collections of the text data in the compressed format. Modifications to the algorithm have been proposed in order to find all the occurrences of a given pattern [60].

In the context of indexing in a compressed archival storage system, there are a few disadvantages with the original LZW approach. LZW uses a node number to represent a subsequence, and all the numbers are in sequential order. For example, as shown in Figure 10, given the index of the word “pattern”, located at the 12th position in the compressed file, we can determine that the node 3 in the dictionary contains the beginning of the word. However, we are not able to decode the node and its neighboring text because the trie, which needs to be constructed by sequential scanning and decoding the text, is not available yet. In order to start decoding the file from a given location in the file, the trie up to that location is necessary.

6.2.2 Off-Line Compression with File-Specific Trie

We can change an online LZW algorithm into a two-pass off-line algorithm. Figure 11 shows an example of a trie. A pattern can be either within a path from the

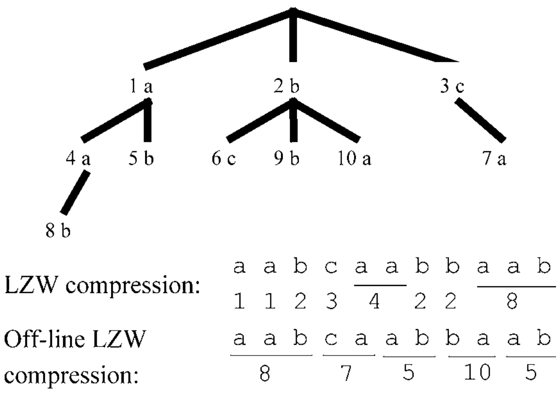


FIG. 11. Example of online and off-line LZW.

root to the node or be contained in the paths of more than one node. Let us consider the text "*aabcaabbaab*" with alphabet $\Sigma = a, b, c$. If we compress the text at the same time when the trie is being built as in current LZW, the output code will be: "11234228". The encoder is getting "smarter" during the encoding process. Take the sub-string "*aab*" as an example; it appears in the text three times. The first time it appears the encoder encodes it as "112"; the second time it is encoded as "42"; the third time it is encoded as "8". If each codeword is 12 bits, the encoder encodes the same substring as 36, 24 and 12 bits at different places. Thus, we may also consider the encoding process as a "training process" of the encoder or, specifically, the trie.

The above example indicates that, if we can "train" the trie before any compression has started, we may get better compression. This is the basic idea of the two-pass compression scheme. In this scheme, the first pass is the training process, which builds the entire trie by scanning the text. The second pass is the actual compression process, which compresses the text from the beginning of the text using the pre-constructed trie. For the above example, the text will be encoded as: "875(10)5". More importantly, since the text is encoded after the trie has been built; it uses the same trie at any point of the encoding process, unlike the original LZW approach, which uses a trie that grows during the encoding. Thus, decoding from any point in the compressed stream is possible. For example, given the compressed data "(10)5" and the above trie, we immediately decode it as "*baab*".

In this approach, a separate static dictionary is used for each file. The dictionary trie is attached to the compressed file to ensure random access property. The size of the trie can be decided by a parameter indicating the number of bits to be used for the trie. The larger the file to be compressed, the less will be the effect on compression ratio by the extra trie overhead. A 12 bit trie occupies a 4 kilobyte space. It indicates that the size of the compressed file plus the dictionary is very close to that of the LZW-compressed file when the file size is large. We can partially decode a portion of a file given the start and end locations of nodes in the compressed node string, or start location of a node and the number of nodes to decode, or a start location and a special flag indicating decoding has to stop. The trie for a given file has to be read and loaded into the memory in order to decode parts of that file. The disadvantage is that when the file size is small, the trie overhead is significant.

6.2.3 Online Compression with Public Trie

Another approach is to use a public static dictionary trie built off-line based on a training data set consisting of a large number of files. This static dictionary trie is then used to compress all the files in the database. In a network environment, both the encoder and decoder keep the same copy of the public trie. For archival text retrieval, the trie is created once and installed in the system and might undergo periodic maintenance if needed. The public trie needs to be transmitted only once. The text is compressed using the trie known to every encoder/decoder. The compressed files are sent separately without the trie. The decoder will refer to the public trie to decode the compressed file. Since the dictionary captures the statistics of a large training set that reflects the source property, the overall compression ratio is expected to improve, although some files may have a worse compression ratio than that obtained by using the original LZW algorithm. Since the trie size is relatively small compared with the overall text size in the text collection, the amortized cost for the whole system is less than the cost for using original LZW and the LZW with individual trie. Another advantage of using a public trie over a file-specific trie is that the words will be indexed based on a single trie. Instead of indexing with a document trie number and the node number inside that trie, we can simply use the node number that is common to all the files in the system.

Figure 12 illustrates the differences among the current LZW, the two-pass off-line LZW, and the public trie LZW. The horizontal bars represent the text file from the first symbol to the end-of-file symbol. Figure 12(a) shows the current implementation of LZW. Usually, the trie is not constructed based on the whole text file. The trie is limited to a certain size. The beginning sector is used for constructing the trie and compressing the text simultaneously. Then we perform compression only for the rest of the text. To retrieve such a compressed text, we need to reconstruct trie and search. Figure 12(b) shows the two pass compression process. The entire text is used to build the trie without any compression. In the implementation, only the beginning portion is used to train the trie due to the limitation of the trie size. Then actual compression is performed on the whole text from the first symbol after the trie is built. Since the algorithm uses a greedy method to do pattern matching in the trie to find a

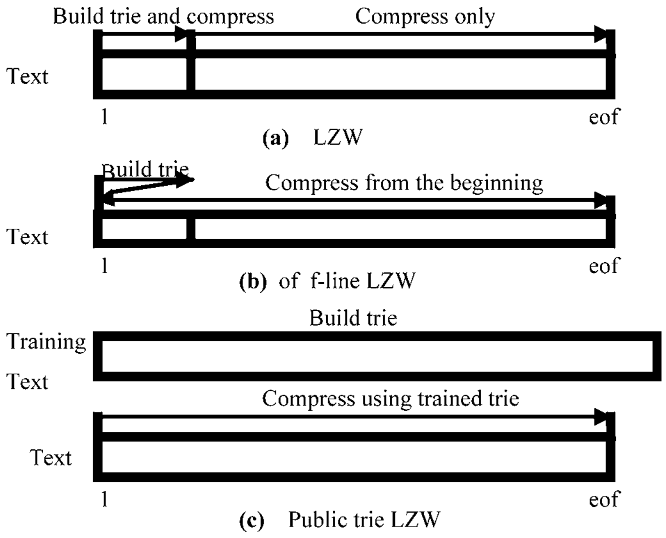


FIG. 12. Illustration of online, off-line, and public trie LZW approach. The shaded part of the text in (a) and (b) is used for training the trie.

representing node number, the beginning portion of the text may have a better compression ratio using a fully built trie. Figure 12(c) shows the public trie method that compresses the text file by first constructing the trie using the training text to capture the statistics of the source. Then compress all the (other) text using an existing trie.

6.2.4 Finding a Public Trie for Good Compression Ratio

Although compression ratio is not the most important issue compared to efficient indexing and searching, it is still worth trying to obtain good compression for the text corpus. An ideal compression scheme for the retrieval purpose should not degrade the compression ratio. Ideally, it should produce some improvement since extra preprocessing has been used. There could be many ways to build a public trie according to the context of the text. For example:

(1) Construct a trie with respect to a randomly chosen text files.
(2) Find a trie that best compressed the test files and use that as the public trie. We will show experimental results in Section 6.3 based on this approach.
(3) Update the trie when the space allocated is full. There could be many criteria to insert/remove the node in the trie. For example, when a new substring occurs, we could prune the node representing the least frequently used sub-

strings. The new substring is added and other updates are performed accordingly. Our aim is to store the most frequently used context in the trie.

(4) Build the trie from the frequency of the q-grams (substring that has a length of q). The PPM algorithm may help to decide which q-grams would be used.

The time to construct the trie for a given file is saved but the pattern matching complexity, i.e., searching the substring in the existing trie, remains the same. The size of the trie is another factor that may affect the compression performance. If the trie is large, we may find longer matching of the substrings. However, we need larger code length for each node. There is a tradeoff between the code length and the size of the trie. We are currently using the commonly used trie size in the popular LZW implementation.

A possible solution to find a good public trie based on LZW dictionary construction method and frequency statistics is as follows. First, we set a relatively large upper bound for the trie size. In addition to the current LZW dictionary construction algorithm that adds the symbol and assigns a new node number, we also add a count to the number of accesses to each node. Note that all the parent nodes have bigger counts than any of the child nodes. In fact, the count of a parent node is the sum of the counts of its child nodes. When the whole trie is built, we start to remove those nodes that have counts less than a threshold value. Obviously, the leaf nodes will be removed first. No parent node will be removed before the child nodes. It is possible that a leaf node has a larger count than a parent node in another branch in the trie. So we will remove the nodes in a branch with a smaller count in a bottom-up order. The pruning will proceed with possibly more than one iteration until a predefined trie size is reached.

6.2.5 Indexing Method and Tag System

When inverted index file is built for the text collection, the keywords are indexed with the document ID and the location where the keywords occurred in the document. In many cases, users are not interested in the exact location of the occurrence, but are interested in the text in the neighborhood of the occurrence. Therefore, we can index the keywords by the block location instead of word location, along with the document ID. For example, if the keyword “computer” is located at 5th paragraph; we simply have a pointer pointing to the starting address of the 5th paragraph in the inverted index file. When the keyword “computer” is searched using any query search method based on the inverted index file, the 5th paragraph will be returned as the result and the whole paragraph will be displayed. If exact location of the keyword needs to be highlighted, a simple pattern matching algorithm such as Boyer–Moore [14] or Knuth–Morris–Pratt [39] algorithms can be used in the very small portion of the text. The definition of “block” is quite flexible. It could be a phrase, a sentence, a line of text, a paragraph, a whole document, a record in a library or database system, or any unit defined by the user. Thus the size of the block determines the granularity of the index system.

To provide different level of granularity in a single index system, we can build a hierarchical system that explores the structure of the text by using the tags inserted at different levels of the file. Although tags have been previously used in systems such as the one in [64], where a block is typically defined as a paragraph and the symbol '`^B`' is used at every block boundary, the tagging is not hierarchical and is not flexible for not being able to provide different granularities of the query. In fixed level partition in [64], compression and index file construction are performed on the block level only. For hierarchical tagging system, XML, which has been a standard for hypertext, is a good example. We can add different user-defined tags to indicate the text boundaries for different levels of granularities. For example, we can use `<r>` and `<\r>` as record boundary, `<P>` and `<\P>` as paragraph boundary, `<doc>` and `<\doc>`as document or file boundary, etc. Some boundary indicators are naturally within the text such as line break and paragraph. However, we need to distinguish between a linguistic word (L-word) in a language dictionary and a word (T-word) in the dictionary trie obtained by training. A T-word stands for a sequence of symbols. As such, the boundaries of the T-words may not coincide with the boundaries of the L-words. A T-word may contain multiple L-words, or parts of one or more L-words. If we are searching for a particular L-word in the text, we need to have some mechanism to assemble the parts of the word into a valid L-word. A node in the trie may also represent more than a single L-word. Although this is an extra pattern assembling stage compared to English word based compressed pattern searching methods, it is language independent.

6.2.6 Compression Ratio vs. Random Access

Besides the equal length byte level code that provides the possibility of dynamic indexing for the context with various resolutions, it also provides the flexibility for the random access. Considering the tradeoff between compression ratio and flexibility of random access for the various compression schemes, the PPM and BWT algorithm give the best compression ratio but can not perform partial decoding at any point. Usually, we have to decode the whole document for further processing. The WordHuff and XRAY provide a better compression ratio than the popular Gzip as well as a better random access at the level of predefined blocks or document. The modified LZW using public trie provides easier random access than the other compression algorithms and a compression ratio competing that of Gzip. In our approach, we can access to the level of the phrases defined in the public dictionary. Raw text is considered the extreme case of no compression and full random access at the symbol level.

To achieve better compression ratio using preprocessing, a filtering processing can be performed before the text is compressed using public trie, as described in Section 4. For example, LIPT can be used to preprocess the raw text. Then, training and compression are based on the transformed text. Our results show that there is a 7% improvement in compression ratio. The disadvantage is that a further processing step is involved and is language dependant as word Huffman based models. The speed of the decoding will also slightly decrease. The storage overhead for the star dictionary is 200 kilobytes.

6.3 Results

6.3.1 Experimental Setup

To evaluate the performance of the modified LZW approach for the partial decoding, we performed experiments on a text database. The database is made up of a total of 840 text files selected from the TREC TIPSTER1993 corpus including Wall Street Journal (1987–1992) and part of AP, DOE and FR files from Disk 1 of the TIPSTER corpus. The file size is 650 MB. The tests were carried out on a PENTIUM-II PC (300 MHz, 512 MB RAM) running REDHAT LINUX 7.0 operating system. In the current LZW implementation, a reference number is used to indicate the code size and the maximum trie size for the dictionary. For example, if we take 12 bits as reference, each node is coded with 12 bit long. The storage of the trie will actually be a prime number greater than 2^{12} bits. The prime number is used for the hashing purposes in order to expedite the access to the trie nodes. The larger the number, the larger is the size of the trie.

Preliminary experiments were performed on different trie sizes and file sizes. In the original LZW, the beginning part of the text is used to build and update the dictionary trie until the trie is full. Then the rest of the text uses the trie to compress without updating the trie. To test our off-line method using file-specific trie, we use the beginning part of the text to build the trie and use the trie to compress the text from the starting point of the text again. In the experiments on the public trie method, we randomly pick the training set from the corpus. Then the trie is used to compress all the files. A pruning algorithm is also implemented and preliminary results are shown.

6.3.2 Performance Comparison

We compared the original LZW implementation *tzip* with our off-line version *trzip* and the public trie version *trdzip*. The trie size is 9604 bytes for a 12-bit trie, 20,836 bytes for a 13-bit trie, and 44,356 bytes for a 14-bit trie. *trzip* uses the beginning part of the text to build the dictionary trie and uses the trie to compress the whole file from the first symbol. *trdzip* loads the dictionary trie from the file that is stored after selection or training. The compression ratio is high when the trie size is selected to be relatively small. We justify our idea with small size trie and then pick the size with best performance in further experiment using larger trie size.

We test the overall compression ratio for the LZW, off-line LZW, and public trie LZW with different trie sizes of 12, 13, and 14 bits respectively. The overall compression is the sum of the compressed file sizes over the sum of the raw text file sizes. The off-line LZW has a slightly smaller compression ratio (0.75%) than the LZW since the beginning part of the text is compressed using the trie trained from itself. The trie size is not included in the individual file because it is fixed no matter how big the file is in the current implementation. The LZW with public trie has a slightly worse compression ratio than LZW (about 7%). However, when we combine the whole corpus together so that we have a universal index over the whole collection instead of breaking the collection into small files, the public trie method has the best compression ratio.

The overall and the average compression ratio are better than that of both LZW and off-line LZW algorithms. The overall compression ratio is given using the trie from the sample file in the corpus that gives the best overall compression. The ratio is computed using the sum of the compressed file size divided by the total corpus size. The average compression ratio is given using the trie from the sample file in the corpus that gives the best average compression. The ratio is computed as the average of the compression ratio of the individual files.

Our results illustrates that the encoding time is linearly increasing with the text file size regardless of the trie size. Our algorithm simply makes another pass that linearly scans through the file after we use partial text to build the trie. If using a public trie, we do not need to build a trie from the beginning. Our algorithm will still be an online one with a predefined dictionary. Although the pattern matching time is not tested here using a large public trie, the time complexity is not expected to significantly improve since the hashing is used to refer to the node.

Experiments are performed by choosing different code/trie size to find an optimal value. The results indicate that the 16 bit code has the best compression ratio in our corpus. Such a byte level coding also brings the advantage of easy random access and parallel processing during retrieval and decoding. The size of the trie is 64 kilobytes. This is much smaller than an English dictionary. We randomly pick the files from the corpus for the training purpose. The training file size is usually around 4 MB. The average compression ratio for our corpus is 0.346 compared with 0.297 from the Word Huffman based compression described in Managing Gigabytes (MG system) [20]. We also tested on the preprocessing of text using LIPT. Using the same training set that is transformed, the compression ratio is improved to 0.32. The compression ratio by Gzip is 0.335 for our corpus. It is worth to mention that in MG system, the compression is based on the statistics of the whole corpus while ours is based upon a small portion of the text.

During decoding stage, our method requires only the storage of the public trie and the space for the text in the stream that requires minimal space. We need 64 KB space for the 16 bit trie. The XRAY system claims to have a 30 MB space requirement. We implement the retrieval system based on the MG system. The dictionary is replaced by the public trie and the pointers. We use the same document ranking function and query evaluation with MG system. The decoding speed is around 5 MB per second for a single CPU that is similar to the MG system. That is, browsing a paragraph or a small file takes insignificant amount of time. We expect to have better performance for the multiple processor system.

Even though we selected trie without fine-tuned training, the compression performance is comparable to Gzip, Word Huffman and XRAY. Most importantly, we obtained the flexibility of indexing on various details of the texts and the ability of random access to any part of the text and decode the small portion comparing with the whole document. Parallel processing can be performed on our byte level code without inherited difficulty. More research is currently underway on optimizing the trie.

7. Search Techniques for Text Retrieval

We have described the text compression using star transform family in Section 4. The performance of search operation in the dictionary mapping is the key for fast transform encoding. We have used a special data structure, *ternary search tree* to achieve this objective.

We have also described the direct search and indexed search schemes on LZW compressed text. As stated before, index file itself may occupy considerable space compared to the original text size. Thus index file may need to be compressed or well organized to facilitate the allocation of the keywords. In Section 5, we have described the Aho–Corasick and Kida’s searching algorithm on compressed text. There are also other searching applications in which ternary trees can be very helpful. In this section, we will explain the dictionary search using ternary tree and some of the applications of ternary suffix trees. We will also look at some efficient construction techniques, some enhancements to enable faster search, and some space-saving optimizations.

7.1 Ternary Search Tree for Dictionary Search

Ternary search trees are similar to digital search tries in that strings are split in the trees with each character stored in a single node as *split char*. Besides, three pointers are included in each node: left, middle and right. All elements less than the split character are stored in the left child, those greater than the split character are stored in the right child, while the middle child contains all elements with the same character.

Search operations in ternary search trees are quite straightforward: current character in the search string is compared with the split char at the node. If the search character is less than the split char, then go to the left child; if the search character is greater than the split char, go to the right child; otherwise, if the search character is equal to the split char, just go to the middle child, and proceed to the next character in the search string. Searching for a string of length k in a ternary search tree with n strings will require at most $O(\log n + k)$ comparisons. The construction time for the ternary tree takes $O(n \log n)$ time [13].

Furthermore, ternary search trees are quite space-efficient. In Figure 13, seven strings are stored in this ternary search tree. Only nine nodes are needed. If multiple strings have same prefix, then the corresponding nodes to these prefixes can be reused, thus memory requirements is reduced in scenarios with large amounts of data.

In the transform encoding module, words in the dictionary are stored in the ternary search trees with the address of corresponding codewords. The ternary search tree is split into 26 distinct ternary search trees. An array is used to store the addresses of these ternary search trees corresponding to the letters $[a..z]$ of the alphabet in the main root node. Words having the same starting character are stored in same subtree, viz. all words starting with ‘a’ in the dictionary exist in the first sub-tree, while all words start with ‘b’ in second sub-tree, and so on.

In each leaf node of the ternary search tree, there is a pointer which points to the corresponding codeword. All codewords are stored in a global memory that is prepared in advance. Using this technique we can avoid storing the codeword in the node, which enables a lot of flexibility as well as space-efficiency. To expedite

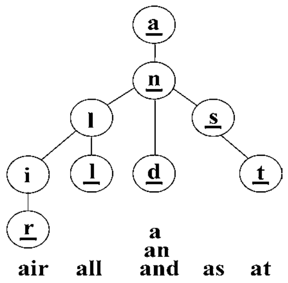


FIG. 13. A ternary search tree.

the tree-build operation, we allocate a big pool of nodes to avoid overhead time for allocating storage for nodes in sequence.

Ternary search tree is sensitive to insertion order: if we insert nodes in a good order (for example, middle element first), we will end up with a balanced tree for which the construction time is small; if we insert nodes in the order of the frequency of words in the dictionary, then the result would be a skinny tree that is very costly to build but efficient to search. In our experiment, we confirmed that insertion order has a lot of performance impact in the transform encoding phase. Our approach is just to follow the *natural* order of words in the dictionary. Result shows that this approach works very well (see Section 4.4.3).

7.2 Ternary Suffix Tree

There are many applications where a large, static text database is searched again and again. A library database is one such application. The user can search for books based on keywords, author names, title or subject. A library database handles thousands of requests per day. Modifications to the data are relatively rare. In such applications, an efficient way of searching the database is necessary. Linear pattern matching techniques, though very efficient, are not applicable in such cases, as they take time proportional to the size of the database. The database can be very big, and handling thousands of requests becomes a huge problem.

For such applications, we need to store the data in some sort of a pre-processed form, in order to be able to handle the search requests efficiently. Building index files and inverted indices is one solution. But this might need a lot of ‘manual’ effort for maintenance. Some one has to decide what words to include in the index. Besides, words that are not in the index can not be searched.

There are some data structures to handle these situations appropriately. Suffix trees and binary search trees are the most popular ones. Suffix trees have very efficient search performance—search takes $O(n)$ time, where n is the length of the pattern being searched. But suffix trees require huge amounts of storage space—they typically require around $24m$ to $28m$ space, where m is the size of the text(the dictionary or the database, in this case). Another alternative is to use the suffix array. The suffix array takes much lesser space than the suffix tree. The search procedure involves a binary search for the pattern within the suffix array. The search performance is much slower than that of a suffix tree. Therefore, we need a data structure that requires lesser space than the suffix tree, but gives better search performance than the binary tree. *Ternary suffix tree* is such a data structure.

A ternary suffix tree is nothing but a ternary search tree built for all the suffixes of an input string/file. In the ternary suffix tree, no branch is extended beyond what is necessary to distinguish between two suffixes. Therefore, if the ternary tree is constructed for a dictionary, as each term in the dictionary is unique, the ternary suffix tree will effectively be a ternary tree for the words in the dictionary.

7.3 Structure of Ternary Suffix Trees

Each node in a ternary tree has three children—the lesser child, the equal child, and the greater child. The search path takes the lesser, equal or greater child depending on whether the current search key is lesser, equal to or greater than the *split char* at the current node. The node stores pointers to all the three children.

In order to be able to retrieve all the occurrences of a key word, we need two more pointers at each node—*begin* and *end*. *Begin* and *end* correspond to the beginning and ending indices of the range of suffixes that correspond to the node. Therefore, *begin* is equal to *end* for all leaf nodes, as the leaf node corresponds to a unique suffix.

The *begin* and *end* pointers serve two purposes—firstly, they eliminate the necessity to store the rest of the suffix at every leaf in the tree. These pointers can be used to go to the exact location in the text corresponding to the current suffix and compare directly with the text if the search for a string reaches a leaf node. Therefore, we need to store the text in memory only once.

Secondly, the *begin* and *end* pointers help in finding multiple occurrences of a string. If the search for a string ends at an intermediate node, all the suffixes in the suffix array from *begin* to *end* match the string. There fore the string occurs $(end - begin + 1)$ number of times in the text, and the locations of the occurrences are the same as the starting positions of the corresponding suffixes.

7.4 Construction of Ternary Suffix Trees

The construction of the ternary suffix tree requires the sorted suffixes, or the suffix array. The sorted suffixes can be obtained in many ways, depending on the form of the input text. If the input is uncompressed text, we can do a quick sort on all the suffixes, which takes $O(m \log m)$ time. If the input is BWT-compressed text, we can use the suffix array (*Hrs*) data structure constructed during the decompression process in BWT [1]. Table I gives an example of the suffixes and corresponding suffix array entry of the string “*abrab*$”.

Once we have the sorted suffixes, the ternary tree can be constructed using different approaches. Here, we consider the median approach and the mid-point approach. In the median approach, the median of the list of characters at the current depth in all the suffixes represented by the node is selected to be the split char, i.e., there will be the same number of nodes on either side that are at the same depth as that of the

TABLE I
SUFFIXES AND SUFFIX ARRAY FOR THE TEXT T = "*abrab*\$"

	Suffixes	Suffix array *Hrs*
1	\$	6
2	*ab*\$	4
3	*abrab*\$	1
4	*b*\$	5
5	*brab*\$	2
6	*rab*\$	3

current node. In the midpoint approach, we select the suffix that is located exactly in the middle of the list of suffixes corresponding to the current node. Calculation of the median requires a scan through all the suffixes corresponding to the current node, where as calculating the midpoint is a single operation. Therefore, construction of a ternary suffix tree based on the median approach is significantly slower than the construction based on midpoint approach. However, the median approach results in a tree that is alphabetically more balanced—at every node, the lesser child and the greater child correspond to approximately the same number of symbols of the alphabet. The midpoint approach results in a tree that is more balanced population wise—the lesser sub tree and the greater sub tree have nearly the same number of nodes.

We provide a comparison between the two approaches. The tree construction is around 300% faster for the midpoint approach as compared to the median approach. The search is around 5% faster for the midpoint approach compared to the median approach.

7.4.1 Searching for Strings

The search procedure involves the traversal of the ternary tree according to the given pattern. To begin, we search for the first character of pattern, starting at the root. We compare the first character of the pattern with the split char at the root node. If the character is smaller than the split char, then we go the lesser (left) child, and compare the same character with the split char at the left child. If the search character is greater than the splitchar, we go to the greater (right) child. If the search character matches the splitchar, we take the equal (middle) child, and advance the search to the next character in the pattern. At any node, if the desired greater child or lesser child is null, it implies that the pattern is not found in the text. If the search advances to the last character in the given pattern, and if the search path reaches a node at which that character matches the splitchar, then we found the pattern. The pattern occurs in the text at locations $Hrs[begin]$, $Hrs[begin + 1]$, and $Hrs[end]$.

7.5 Implementation

7.5.1 Elimination of Leaves

The leaves of the ternary tree do not resolve any conflicts. Therefore, all the leaves can be eliminated. This, however would require slight modifications in the search procedure. If the desired child is a lesser child or greater child, and that child was null, we have to go to the location in the text which would correspond to that leaf if that leaf existed, and linearly compare the text and the pattern. This location is easy to calculate: The leaf, either lesser or greater, corresponds to only one location in the text. Therefore, for the lesser child, this location has to be *Hrs*[*begin*], and for the greater child this has to be *Hrs*[*end*], where begin, end belong to the current node.

On a test run, for a file of size 586,000 characters, this optimization saved 385,000 nodes, which is a saving of almost 14 bytes per character.

Figure 14(a) shows the ternary suffix tree for the string "*abrab*$". Figure 14(a) shows the ternary suffix tree after eliminating the leaves. The edges to equal children are represented by a broken line.

7.5.2 Path Compression

For a node, if both the lesser and greater leaves are null, and if the equal child is not null, then it is not necessary to have that node in the tree. The node can be replaced by its equal child if the equal child can some how store the number of nodes eliminated in this fashion between its immediate parent and itself. We call this *path compression*.

Therefore, the nodes at which path compression has been implemented can be treated as special nodes. These nodes store an extra integer (denoted by pclength), which stores the path compression length. When the search path reaches any of these

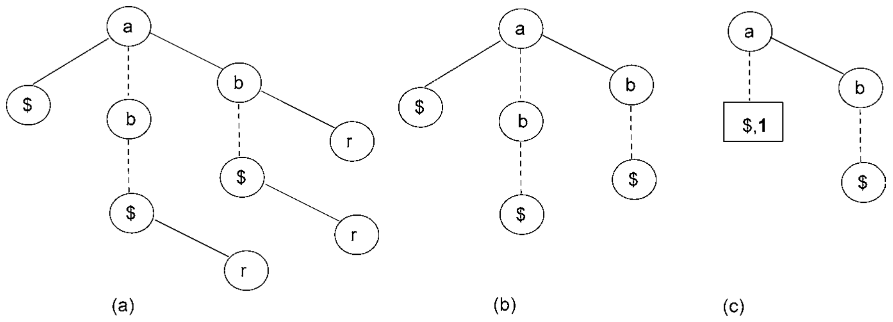


FIG. 14. (a) Ternary suffix tree for "*abrab*$"; (b) The tree after eliminating leaves; (c) The tree after eliminating leaves and applying path compression.

special nodes, the comparison has to be done in the text for a length equal to pclength before continuing the search from the current node. The comparison in the text can be done at any of the locations corresponding to *Hrs*[*begin*], *Hrs*[*end*] of the special node.

On a test run, for a file of size 586,000 characters, this optimization saved 355,000 nodes, which is a saving of almost 10 bytes per character of the original text.

Figure 14(c) shows the ternary suffix tree after applying path compression to the tree in Figure 14(b). The special nodes are represented by rectangular boxes. The path compression length is indicated in bold type.

7.6 Results

For the test file mentioned above, the total number of nodes in the tree were around 549,000. Out of these special nodes were around 128,000, requiring 24 bytes per node. Therefore, the total memory used was around 19.6 bytes per character. Other files produced similar results, using around 20 bytes per character. We compared the search performance of the ternary suffix tree with the that of binary search in the suffix array. The results are shown in Table II. Ternary suffix trees were built for different files in the Calgary corpus. The construction time shown in the table is the construction time required to build the ternary suffix tree from the suffix array of each file. The results are based on searching each word from a dictionary of 59,951 English words, searching each word 10 times. It can be seen form the results that the search performance of the ternary suffix tree is much better than that of binary search in the suffix array. It can also be seen that the extra construction time required

TABLE II
COMPARISON OF SEARCH PERFORMANCE

File	Size	Binary search		Ternary suffix tree			
		Time (ms)	Comparisons	Total time	Search time	Construction time	Search comparisons
Alice29.txt	152,089	16,000	17,312,160	12,800	10,440	2360	17,233,990
Anne11.txt	588,960	56,330	24,285,300	20,260	11,920	8340	19,592,500
Bib.txt	111,261	15,460	17,184,600	13,570	11,020	2550	17,384,840
Book2	610,856	25,890	24,557,380	22,420	11,770	10,650	19,235,470
Lcet10.txt	426,754	23,930	23,282,720	19,070	11,150	7920	18,340,670
Plrabn12.txt	481,861	25,470	24,497,200	18,500	12,230	6270	20,095,030
News	377,109	22,630	22,441,110	31,110	12,370	18,740	19,589,940
World95.txt	2,988,578	38,970	31,760,050	120,010	13,230	106,780	23,196,930
1musk10.txt	1,344,739	33,340	28,236,560	32,300	12,780	19,520	21,010,900

for building the ternary is more than compensated for over a large number (approximately 600,000) searches.

8. Parallel and Distributed Text Processing

Although text processing and retrieval algorithms have been studied extensively, most approaches target on serial computation. There have been some research on parallel text compression [59,38]. But the huge amount of queries on the Internet and local machines as well as the availability of the powerful cluster computing architecture demand further research on parallel and distributed computation for text processing. In this section, we describe some approaches for text processing based on our searching and compression methods.

8.1 Parallel Processing on Modified LZW Compressed Files

We have shown the performance in Section 6 for our public trie LZW with the ability of partial decoding and random access. It seems that there are other methods having a better compression ratio and also allow some levels of partial decoding such as word Huffman and XRAY. However, the XRAY and word Huffman schemes are based on the Huffman code for the final compression output. "The sequential algorithm for Huffman coding is quite simple, but unfortunately it appears to be inherently sequential. Its parallel counterpart is much more complicated" [26]. In our algorithm, each node in the public trie is encoded in 16 bits (2 bytes). It gives us a better flexibility that we can start decoding from any even numbered byte after the header in the sequence to obtain the correct original text. Therefore it is more convenient for our compression scheme to provide a parallel access to the compressed code. We have simulated the parallel decoding of the compressed text by using multiple threading in the Unix system. Each thread can read any part of the compressed text without conflict with the other threads.

Another advantage using our public trie method is that encoding and decoding can both be performed in parallel and parallel decoding is not confined by the encoding process. In parallel encoding, we can divide the text into arbitrary blocks and each block is compressed by a processor. The decoder does not need to know the boundaries of each to start a decoding process. The decoder can also arbitrarily separate the compressed text at any even number byte position after the header. That is, the encoder and decoder do not depend on each other in the parallel processing. They only need to know which public trie to use. The parallel algorithm becomes a trivial one that can use simple scheduling algorithm. For example, if we have 3 processors available to compress the text, we can divide the text in to 3 parts and start to encode on each processor and then concatenate the results. If we have 5 processors available during the decoding time. We can also divide the compressed into 5 parts after all the processors reading the hearer information. Thus all the processor can perform decoding independently.

8.2 Ternary Suffix Trees in a Multiprocessor Environment

The ternary suffix trees can be further enhanced to take advantage of a multiprocessor environment. Both the construction and search operations can be parallelized.

A parallel algorithm for the construction of suffix arrays has been proposed by Futamura et al. in [31]. In a general case, the algorithm is expected to give a linear speedup. The algorithm works by dividing suffixes into buckets based on a w-length prefix of each suffix. The buckets are then distributed among the processors. The complete suffix sorting within each bucket is carried out in parallel. In fact, hashing the suffixes into buckets is itself done in parallel. Given a string S of length n and p processors, the string is divided into p chunks, each approximately of length n/p. Each processor takes one of these p chunks, and hashes the suffixes that begin within this chunk into $|\Sigma|^w$ buckets based on the w-length prefix of each suffix. The value of w is chosen experimentally, based on the values of n and p. Load balancing algorithms are then used to distribute the buckets among the processors. Each processor will then do a complete suffix sorting within the buckets allocated to itself. In fact, each processor can build a ternary tree for each bucket that is allocated to it. These ternary trees can be built by ignoring the w-length prefix of each suffix, as this part is common to all the suffixes within the bucket. These ternary trees can be quickly assembled together to form a single ternary tree for the entire string.

The brute force approach to search using ternary trees in a multiprocessor environment would be to maintain a copy of the complete ternary tree at each node. The queries can be distributed among the processors in a round-robin fashion. This will result in a linear speedup in the overall search time. However, the drawback to this approach is that the complete ternary tree has to be stored at each node.

Theoretically, superlinear speedups in search time can be achieved using a slightly different approach. As explained above, each processor builds a ternary tree for each bucket allocated to it. Instead of merging these ternary trees into a single ternary tree, we can simply leave these ternary trees with the processors they were constructed in.

During search, the keywords can be quickly hashed based on a w-length prefix of each keyword. The hashing will determine which bucket the keyword has to be searched in. Based on the bucket number, the keyword can be given to the processor that holds the ternary tree for the corresponding bucket. Theoretically, this can lead to superlinear speedups, since much smaller ternary trees need to be stored at each node. This will improve the spatial locality of the search, there by reducing the search time. However, this needs to be verified experimentally. There might be other factors like the non-uniform distribution of the queries among the buckets, which might lead to load imbalance, there by slowing down the search performance.

In conclusion, using specialized data structures like ternary suffix trees drastically improves the search performance. The improvement comes not only because of the distribution of load between multiple processors, but also because of the much smaller memory requirements at each processor.

9. Conclusion

In this chapter, we described the recent algorithms on text compression using filtering, compressed pattern matching on LZW compressed file, public trie LZW compression algorithm and corresponding searching and parallel processing methods, and data structures that facilitate fast searching and transformation as well as the performance of the algorithms.

We have shown that the preprocessing of the text using a semi-static dictionary model helps to improve the compression of the text. The speed consideration is well taken care of by using ternary tree data structure that brings fast mapping in both compression and decompression stage. The compression also brings smaller bandwidth during transmission that is usually a bottleneck on the Internet.

We have shown that random access and partial decoding can be done efficiently on compressed text with a few modifications to the LZW algorithm. We obtain the ability to randomly access any part of the text given the location index. Compared to a decompress-then-search method with a complexity of $O(z + u)$, where z is the compressed file size and u is the decoded file size, our method performs in a sublinear complexity since we only decode a small portion of the file. Moreover, our modified method improved the compression ratio compared to original LZW. We have justified the feasibility of using a public trie by building a trie from a part of the text collection. The public trie method makes our compression schemes have the advantage of random access and partial decoding of any part of the text, which is difficult or not possible using other compression methods. Parallel computing is straightforward with the public trie methodology. We also show how ternary suffix tree can be used in multiple processor scenario.

Compressed pattern matching algorithms based on LZW are proposed. They can either facilitate searching directly com the compressed file or be a component of retrieval system such as searching on compressed index file.

In general, text compression is one of the keys to the next generation archival and retrieval system. How to adopt the efficient storage and searching using the parallel and distributed computing will still be a challenge in the Internet age.

ACKNOWLEDGEMENT

Our work is partially supported by grants from the National Science Foundation, IIS-0207819, IIS-0312724.

REFERENCES

[1] Adjeroh D.A., Mukherjee A., Bell T., Powell M., Zhang N., “Pattern matching in BWT-transformed text”, in: *Proceedings, IEEE Data Compression Conference*, 2002, p. 445.

[2] Adjeroh D.A., Powell M., Zhang N., Mukherjee A., Bell T., “Pattern matching on BWT-text: Exact matching”, Manuscript, 2002.

[3] Aho A.V., Corasick M.J., “Efficient string matching: An aid to bibliographic search”, *Communications of the ACM* **18** (6) (1975) 333–340.

[4] Amir A., Benson G., Farach M., “Let sleeping files lie: Pattern matching in Z-compressed files”, *Journal of Computer and System Sciences* **52** (1996) 299–307.

[5] Amir A., Calinescu G., “Alphabet independent and dictionary scaled matching”, in: *Combinatorial Pattern Matching*, in: *Lecture Notes in Computer Science*, vol. 1075, Springer-Verlag, Berlin/New York, 1996, pp. 320–334.

[6] Apostolico A., Landau G.M., Skiena S., “Matching for run-length encoded strings”, in: *Proceedings, Complexity and Compression of Sequences*, 1997.

[7] Arnavut Z., “Move-to-front and inversion coding”, in: *Proceedings of the Conference on Data Compression*, IEEE Computer Society, Los Alamitos, CA, 2000, p. 193.

[8] Baeza-Yates R., Ribeiro-Neto B., *Modern Information Retrieval*, Addison–Wesley, Harlow, England, 1999.

[9] Barcaccia P., Cresti A., Agostino S.D., “Pattern matching in text compressed with the ID heuristic”, 1998, pp. 113—118.

[10] Bell T., Moffat A., “A note on the DMC data compression scheme”, *Computer Journal* **32** (1) (1989) 16–20.

[11] Bell T., Powell M., Mukherjee A., Adjeroh D.A., “Searching BWT compressed text with the Boyer–Moore algorithm and binary search”, in: *Proceedings, IEEE Data Compression Conference*, 2002, pp. 112–121.

[12] Bentley J., Sleator D., Tarjan R., Wei V., “A locally adaptive data compression scheme”, *Communications of the ACM* **29** (1986) 320–330.

[13] Bentley J.L., Sedgewick R., “Fast algorithms for sorting and searching strings”, in: *Proceedings of the 8th Annual ACM–SIAM Symposium on Discrete Algorithms*, 1997, pp. 360–369.

[14] Boyer R., Moore J., “A fast string searching algorithm”, *Communications of the ACM* **20** (10) (1977) 762–772.

[15] Bunke H., Csirik J., “An algorithm for matching run-length coded strings”, *Computing* **50** (1993) 297–314.

[16] Bunke H., Csirik J., “An improved algorithm for computing the edit distance of run-length coded strings”, *Information Processing Letters* **54** (1995) 93–96.

[17] Burrows M., Wheeler D., "A block-sorting lossless data compression algorithm", Technical report, Digital Equipment Corporation, Palo Alto, CA, 1994.
[18] Cannane A., Williams H.E., "A general-purpose compression scheme for large collections", *ACM Transactions on Information Systems* **20** (2002) 329–355.
[19] Cleary J., Teahan W., "Unbounded length contexts for PPM", *The Computer Journal* **36** (5) (1997) 1–9.
[20] Cleary J., Witten I., "Data compression using adaptive coding and partial string matching", *IEEE Transactions on Communications* **32** (1984) 396–402.
[21] Cormack G.V., Horspool R.N.S., "Data compression using dynamic Markov modelling", *Computer Journal* **30** (6) (1987) 541–550.
[22] Corpus C, "The Canterbury Corpus" http://corpus.canterbury.ac.nz, 2000.
[23] Corpus C, "The Calgary Corpus", ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus, 2000.
[24] Corpus G, "The Gutenberg Corpus", http://www.promo.net/pg/.
[25] Crochemore M., Mignosi F., Restivo A., Salemi S., "Data compression using antidictionaries", *Proceedings of the IEEE* **88** (11) (2000) 1756–1768.
[26] Crochemore M., Rytter W., *Text Algorithms*, Oxford Press, New York, 1994.
[27] Farach M., Thorup M., "String matching in Lempel–Ziv compressed strings", in: *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing*, ACM, New York, 1995, pp. 703–712.
[28] Farach M., Thorup M., "String matching in Lempel–Ziv compressed strings", *Algorithmica* **20** (1998) 388–404.
[29] Fenwick P., "The Burrows–Wheeler transform for block sorting text compression", *The Computer Journal* **39** (9) (1996) 731–740.
[30] Franceschini R., Mukherjee A., "Data compression using encrypted text", in: *Proceedings of the 3rd Forum on Research and Technology, Advances in Digital Libraries*, 1996, pp. 130–138.
[31] Futamura N., Aluru S., Kurtz S., "Parallel sux sorting", 2001.
[32] Gasieniec L., Karpinski M., Plandowski W., Rytter W., "Randomized efficient algorithms for compressed strings: the finger-print approach", in: *Proceedings, Combinatorial Pattern Matching*, in: *Lecture Notes in Computer Science*, vol. 1075, Springer-Verlag, Berlin/New York, 1996, pp. 39–49.
[33] Gibson J.D., Berger T., Lookabaugh T., Lindbergh D., Baker R.L., *Digital Compression for Multimedia: Principles and Standards*, Morgan Kaufmann, San Francisco, CA, 1998.
[34] Huffman D., "A method for the construction of minimum redundancy codes", *Proceedings of IRE* **40** (9) (1952) 1098–1101.
[35] Karkkainen J., Ukkonen E., "Lempel–Ziv index for q-grams", *Algorithmica* **21** (1998) 137–154.
[36] Kida T., Takeda M., Miyazaki M., Shinohara A., Arikawa S., "Multiple pattern matching in Izw compressed text", *Journal of Discrete Algorithm* **1** (1) (2000).
[37] Kida T., Takeda M., Shinohara A., Miyazaki M., Arikawa S., "Multiple pattern matching in Izw compressed text", in: *Proceedings, Data Compression Conference*, 1998, p. 103.
[38] Klein S.T., Wiseman Y., "Parallel Lempel Ziv coding", in: *Lecture Notes in Computer Science*, vol. 2089, Springer-Verlag, Berlin/New York, 2001, pp. 18–30.

[39] Knuth D., Morris J., Pratt V., “Fast pattern matching in strings”, *SIAM Journal of Computing* **6** (2) (1977) 323–350.
[40] Kobayashi M., Takeda K., “Information retrieval on the web”, *ACM Computing Surveys* **32** (2) (2000) 144–173.
[41] Kruse H., Mukherjee A., “Preprocessing text to improve compression ratios”, in: *Proceedings of IEEE Data Compression Conference*, 1998.
[42] Lyman P., Varian H.R., Swearingen K., Charles P., Good N., Jordan L.L., Pal J., “How much information?”, http://www.sims.berkeley.edu/research/projects/how-much-info-2003/, Technical report, School of Information Management and Systems at the University of California at Berkeley, 2003.
[43] Mäkinen V., Navarro G., Ukkonen E., “Approximate matching of run-length compressed strings”, in: *Proceedings, Combinatorial Pattern Matching*, 2001, pp. 31–49.
[44] Manber U., “A text compression scheme that allows fast searching directly in the compressed file”, *ACM Transactions on Information Systems* **15** (2) (1997) 124–136.
[45] Moffat A., “Linear time adaptive arithmetic coding”, *IEEE Transactions on Information Theory* **36** (2) (1990) 401–406.
[46] Moffat A., Sacks-Davis R., Wilkinson R., Zobel J., “Retrieval of partial documents”, in: *Proceedings of TREC Text Retrieval Conference*, 1993, pp. 181–190.
[47] Moffat A., Turpin A., *Compression and Coding Algorithms*, Kluwer Academic Publishers, Dordrecht/Norwell, MA, 2002.
[48] Moura E.S., Navarro G., Ziviani N., Baeza-Yates R., “Fast and flexible word searching on compressed text”, *ACM Transactions on Information Systems* **18** (2) (2000) 113–139.
[49] Mukherjee A., Awan F., “Text compression”, in: Sayood K. (Ed.), *Lossless Compression Handbook*, Academic Press, San Diego, CA, 2002, Chapter 10.
[50] Navarro G., Raffinot M., “A general practical approach to pattern matching over Ziv–Lempel compressed text”, in: *Proceedings, Combinatorial Pattern Matching*, in: *Lecture Notes in Computer Science*, vol. 1645, Springer-Verlag, Berlin/New York, 1999, pp. 14–36.
[51] Rissanen J., Langdon G.G., “Arithmetic coding”, *IBM Journal of Research and Development* **23** (2) (1979) 149–162.
[52] Salomon D., *Data Compression: The Complete Reference*, second ed., Springer-Verlag, Berlin/New York, 2000.
[53] Sayood K., *Introduction to Data Compression*, second ed., Morgan Kaufmann, San Mateo, CA, 2000.
[54] Shannon C., “A mathematical theory of communication”, *Bell Systems Technical Journal* **27** (1948) 379–423, 623–656.
[55] Shannon C., “Prediction and entropy of printed English”, *Bell Systems Technical Journal* **30** (1951) 55.
[56] Shannon C.E., Weaver W., *The Mathematical Theory of Communication*, University of Illinois Press, Champaign, 1998.
[57] Shibata Y., Kida T., Fukamachi S., Takeda T., Shinohara A., Shinohara S., Arikawa S., “Byte-pair encoding: A text compression scheme that accelerates pattern matching”, Technical report, Department of Informatics, Kyushu University, Japan, 1999.

[58] Shibata Y., Takeda M., Shinohara A., Arikawa S., "Pattern matching in text compressed by using antidictionaries", in: *Proceedings, Combinatorial Pattern Matching*, in: *Lecture Notes in Computer Science*, vol. 1645, Springer-Verlag, Berlin/New York, 1999, pp. 37–49.
[59] Stauffer L.M., Hirschberg D.S., "Parallel text compression", Technical report, University of California, Irvine, CA, 1993.
[60] Tao T., Mukherjee A., "LZW based compressed pattern matching", in: *Proceedings of IEEE Data Compression Conference*, 2004.
[61] Teahan W.J., Cleary J.G., "The entropy of English using PPM-based models", in: *Data Compression Conference*, 1996, pp. 53–62.
[62] TREC, "Official webpage for TREC—Text REtrieval Conference series", http://trec.nist.gov.
[63] Welch T., "A technique for high performance data compression", *IEEE Computer* **17** (1984) 8–20.
[64] Witten I.H., Moffat A., Bell T.C., *Managing Gigabytes: Compressing and Indexing Documents and Images*, second ed., Morgan Kaufman, San Mateo, CA, 1999.
[65] Zhang N., Mukherjee A., Adjeroh D.A., Bell T., "Approximate pattern matching using the burrows-wheeler transform", in: *Proceedings of Data Compression Conference, Snowbird, UT*, IEEE Computer Society, Los Alamitos, CA, 2003.
[66] Zhang N., Tao T., Satya R.V., Mukherjee A., "Modified LZW algorithm for efficient compressed text retrieval", in: *Proceedings of International Conference on Information Technology: Coding and Computing, Las Vegas, NV*, 2004, pp. 224–228.
[67] Ziv J., Lempel A., "A universal algorithm for sequential data compression", *IEEE Transactions on Information Theory* **IT-23** (1977) 337–343.
[68] Ziv J., Lempel A., "Compression of individual sequences via variable rate coding", *IEEE Transactions on Information Theory* **IT-24** (1978) 530–536.
[69] Ziviani N., Moura E.S., Navarro G., Baeza-Yates R., "Compression: A key for next generation text retrieval systems", *IEEE Computer* **33** (11) (2000) 37–44.

Further Reading

[70] Gusfield D., *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, Cambridge, UK, 1997.
[71] Myers E.W., "A sublinear algorithm for approximate keyword searching", *Algorithmica* **12** (1994) 345–374.
[72] Navarro G., "A guided tour of approximate string matching", *ACM Computing Surveys* **33** (1) (2001) 31–88.
[73] Ukkonen E., "Finding approximate patterns in strings", *Journal of Algorithms* **6** (1985) 132–137.

Author Index

Numbers in *italics* indicate the pages on which complete references are given.

Subject Index

Contents of Volumes in This Series