Multiprocessor Systems-on-Chips

Edited by

Ahmed Amine Jerraya

TIMA Laboratory

Wayne Wolf

Princeton University

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann Publishers is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

Petitioner Samsung Ex-1010, 0001

Senior Editor Denise E. M. Penrose
Publishing Services Manager Andre Cuello
Project Manager Brandy Palacios
Editorial Assistant Valerie Witte
Cover Design Chen Design Associates, San Francisco
Composition Best-Set Typesetter Ltd.
Technical Illustration Graphic World Publishing Services
Copyeditor Graphic World Publishing Services
Proofreader Graphic World Publishing Services
Indexer Graphic World Publishing Services
Interior printer Maple Press
Cover printer Phoenix Color

Morgan Kaufmann Publishers is an Imprint of Elsevier
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

Copyright © 2005, Elsevier Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: Phone (+44) 1865 843830, fax (+44) 1865 853333, e-mail: permissions@elsevier.com.uk. You may also complete your request on-line via the Elsevier homepage (http://elsevier.com), by selecting "Customer Support" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data
Application Submitted

ISBN: 0-12385-251-X

For all information on all Morgan Kaufmann publications visit our website at *www.mkp.com*

Printed in the United States of America

Petitioner Samsung Ex-1010, 0002

Dedication

For Houda, Lydia and Nelly—Ahmed Amine Jerraya

For Nancy and Alec—Wayne Wolf

Petitioner Samsung Ex-1010, 0003

About the Editors

Ahmed Amine Jerraya (*ahmed.jerraya@imag.fr*) is Research Director with CNRS and is currently managing research dealing with Multiprocessor System-on-Chips at TIMA Laboratory, France. He received a degree in engineering from the University of Tunis in 1980 and the D.E.A., "Docteur Ingénieur", and the "Docteur d'Etat" degrees from the University of Grenoble in 1981, 1983, and 1989 respectively, all in computer sciences. From April 1990 to March 1991, he was a Member of the Scientific Staff at Nortel in Canada, working on linking system design tools and hardware design environments. He served as General Chair for the Conference DATE in 2001 and published more than 200 papers in International Conferences and Journals. He received the Best Paper Award at the 1994 ED&TC for his work on Hardware/Software Co-simulation.

Wayne Wolf (*wolf@princeton.edu*) is Professor of Electrical Engineering at Princeton University. Before joining Princeton, he was with AT&T Bell Laboratories, Murray Hill, New Jersey. He received the B.S., M.S., and Ph.D. degrees in Electrical Engineering from Stanford University in 1980, 1981, and 1984, respectively. His research interests include embedded computing, VLSI systems, and multimedia information systems. He is the author of *Computers as Components* and *Modern VLSI Design* (for which he won the ASEE/CSE and HP Frederick E. Terman Award). Wolf has been elected to Phi Beta Kappa and Tau Beta Pi. He is a Fellow of the IEEE and ACM and a member of the SPIE and ASEE.

Petitioner Samsung Ex-1010, 0004

Contents

About the Editors vi

Preface xix

1 The What, Why, and How of MPSoCs 1

Ahmed Amine Jerraya and Wayne Wolf

1.1 Introduction 1

1.2 What are MPSoCs 1

1.3 Why MPSoCs? 5

1.4 Challenges 10

1.5 Design Methodologies 11

1.6 Hardware Architectures 13

1.7 Software 14

1.7.1 Programmer's Viewpoint 14

1.7.2 Software architecture and design reuse viewpoint 15

1.7.3 Optimization Viewpoint 16

1.8 The Rest of the Book 18

Petitioner Samsung Ex-1010, 0005

PART I HARDWARE 19

2 Techniques for Designing Energy-Aware MPSoCs 21
Mary Jane Irwin, Luca Benini, N. Vijaykrishnan, and Mahmut Kandemir

2.1 Introduction 21
2.2 Energy-Aware Processor Design 23
2.2.1 Reducing Active Energy 24
2.2.2 Reducing Standby Energy 26
2.3 Energy-Aware Memory System Design 27
2.3.1 Reducing Active Energy 28
2.3.2 Reducing Standby Energy 28
2.3.3 Influence of Cache Architecture on Energy Consumption 29
2.3.4 Reducing Snoop Energy 33
2.4 Energy-Aware On-Chip Communication System Design 34
2.4.1 Bus Encoding for Low Power 34
2.4.2 Low Swing Signaling 39
2.4.3 Energy Considerations in Advanced Interconnects 41
2.5 Energy-Aware Software 44
2.6 Conclusions 46

3 Networks on Chips: A New Paradigm for Component-Based MPSoC Design 49
Luca Benini and Giovanni De Micheli

3.1 Introduction 49
3.1.1 Technology Trends 49
3.1.2 Nondeterminism in SoC Abstraction Models 50
3.1.3 A New Design Approach to SoCs 51

Petitioner Samsung Ex-1010, 0006

3.2 Signal Transmission on Chip 52
3.2.1 Global Wiring and Signaling 53
3.2.2 Signal Integrity 55
3.3 Micronetwork Architecture and Control 57
3.3.1 Interconnection Network Architectures 58
3.3.2 Micronetwork Control 63
3.4 Software Layers 73
3.4.1 Programming Model 73
3.4.2 Middleware Architecture 75
3.4.3 Software Development Tools 78
3.5 Conclusions 80

4 Architecture of Embedded Microprocessors 81
Eric Rotenberg and Aravindh Anantaraman
4.1 Introduction 81
4.2 Embedded Versus High-Performance Processors: A Common Foundation 82
4.3 Pipelining Techniques 85
4.3.1 Bypasses 86
4.3.2 Branch Prediction 87
4.3.3 Caches 90
4.3.4 Dynamic Scheduling 91
4.3.5 Deeper Pipelining, Multiple-Instruction Issue, and Hardware Multithreading 93
4.4 Survey of General-purpose 32-bit Embedded Microprocessors 96
4.4.1 ARM 98
4.4.2 High-end Embedded MPUs 103
4.4.3 Ubicom IP3023: Deterministic High Performance via Multithreading 105
4.5 Virtual Simple Architecture (VISA): Integrating Non-Determinism Without Undermining Safety 108
4.6 Conclusions 110

Petitioner Samsung Ex-1010, 0007

5 Performance and Flexibility for Multiple-Processor SoC Design 113
Chris Rowen

5.1 Introduction 113
5.2 The Limitations of Traditional ASIC Design 118
5.2.1 The Impact of SoC Integration 120
5.2.2 The Limitations of General-Purpose Processors 120
5.2.3 DSP as Application-Specific Processor 122
5.3 Extensible Processors as an Alternative to RTL 122
5.3.1 The Origins of Configurable Processors 123
5.3.2 Configurable, Extensible Processors 123
5.3.3 Configurable and Extensible Processor Features 130
5.3.4 Extending a Processor 132
5.3.5 Exploiting Extensibility 134
5.3.6 The Impact of Extensibility on Performance 136
5.3.7 Extensibility and Energy Efficiency 142
5.4 Toward Multiple-Processor SoCs 142
5.4.1 Modeling Systems with Multiple Processors 144
5.4.2 Developing an XTMP Model 144
5.5 Processors and Disruptive Technology 147
5.6 Conclusions 149

6 MPSoC Performance Modeling and Analysis 153
Rolf Ernst

6.1 Introduction 153
6.1.1 Complex Heterogeneous Architectures 153
6.1.2 Design Challenges 156
6.1.3 State of the Practice 157
6.1.4 Chapter Objectives 159
6.1.5 Structuring Performance Analysis 160

Petitioner Samsung Ex-1010, 0008

6.2 Architecture Component Performance Modeling and Analysis 161
6.2.1 Processing Element Modeling and Analysis 161
6.2.2 Formal Processing Element Analysis 163
6.2.3 Communication Element Modeling and Analysis 165
6.2.4 Formal Communication Element Analysis 166
6.2.5 Memory Element Modeling and Analysis 167
6.2.6 Architecture Component Modeling and Analysis: Summary 168
6.3 Process Execution Modeling 168
6.3.1 Activation Modeling 168
6.3.2 Software Architecture 170
6.3.3 Process Execution Modeling: Summary 170
6.4 Modeling Shared Resources 171
6.4.1 Resource Sharing Principle and Impact 171
6.4.2 Static Execution Order Scheduling 172
6.4.3 Time-Driven Scheduling 173
6.4.4 Priority-Driven Scheduling 176
6.4.5 Resource Sharing-Summary 178
6.5 Global Performance Analysis 179
6.6 Conclusions 185

7 Design of Communication Architectures for High-Performance and Energy-Efficient Systems-on-Chips 187
Sujit Dey, Kanishka Lahiri, and Anand Raghunathan
7.1 Introduction 187
7.2 On-Chip Communication Architectures 189
7.2.1 Terminology 190
7.2.2 Communication Architecture Topologies 190
7.2.3 On-Chip Communication Protocols 192
7.2.4 Communication Interfaces 194
7.3 System-Level Analysis for Designing Communication Architectures 194
7.3.1 Trace-Based Analysis of Communication Architectures 196

Petitioner Samsung Ex-1010, 0009

7.4 Design Space Exploration for Customizing Communication Architectures 203
7.4.1 Communication Architecture Templates 203
7.4.2 Communication Architecture Template Customization 204
7.5 Adaptive Communication Architectures 210
7.5.1 Communication Architecture Tuners 210
7.6 Communication Architectures for Energy/Battery-Efficient Systems 216
7.6.1 Minimizing Energy Consumed by the Communication Architecture 216
7.6.2 Improving System Battery Efficiency Through Communication Architecture Design 218
7.7 Conclusions 222

8 Design Space Exploration of On-Chip Networks: A Case Study 223
Bishnupriya Bhattacharya, Luciano Lavagno, and Laura Vanzago
8.1 Introduction 223
8.2 Background 225
8.2.1 Function/Architecture Co-Design Methodology 225
8.2.2 Performance Modeling with Architecture Services 227
8.2.3 Mechanics of Architecture Services 229
8.2.4 Architecture Topology Binds Services 230
8.2.5 Communication Patterns 231
8.3 Modeling of Dataflow Networks 233
8.4 Case Study: Hiperlan/2 Application 235
8.4.1 Modeling the Hiperlan/2 Physical Layer 236
8.5 The Architectural Platform 238
8.5.1 Architectural Modeling 240
8.5.2 Mapping and Communication Refinement 241
8.6 Results 243
8.6.1 Communication Refinement 245
8.6.2 FPGA Alternatives 246
8.7 Conclusions 248

Petitioner Samsung Ex-1010, 0010

PART II SOFTWARE 249

9 Memory Systems and Compiler Support for MPSoC Architectures 251
Mahmut Kandemir and Nikil Dutt

9.1 Introduction and Motivation 251
9.2 Memory Architectures 252
9.2.1 Types of Architectures 254
9.2.2 Customization of Memory Architectures 261
9.2.3 Reconfigurability and Challenges 267
9.3 Compiler Support 269
9.3.1 Problems 269
9.3.2 Solutions 271
9.4 Conclusions 281

10 A SystemC-Based Abstract Real-Time Operating System Model for Multiprocessor System-on-Chips 283
Jan Madsen, Kashif Virk, and Mercury Jair Gonzalez

10.1 Introduction 283
10.2 Basic Concepts and Terminology 286
10.2.1 Platform Architecture 286
10.2.2 Tasks 286
10.2.3 Basics of Scheduling 288
10.3 Basic System Model 290
10.4 Uniprocessor Systems 292
10.4.1 Link Model 292
10.4.2 Task Model 293
10.4.3 Scheduler Model 296
10.4.4 Synchronization Model 300
10.4.5 Resource Allocation Model 302

Petitioner Samsung Ex-1010, 0011

10.5 Multiprocessor Systems 303
10.5.1 Multiprocessing Anomalies 306
10.5.2 Interprocessor Communication 308
10.5.3 Multiprocessor Example 309
10.6 Summary 311

11 Cost-Efficient Mapping of Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems 313
Peng Yang, Paul Marchal, Chun Wong, Stefaan Himpe, Francky Catthoor, Patrick David, Johan Vounckx, and Rudy Lauwereins
11.1 Introduction 313
11.2 Platform Based Design 314
11.3 Related Work 315
11.4 Target Platform Architecture and Model 319
11.5 Task Concurrency Management 320
11.5.1 Global TCM Methodology 321
11.5.2 Two-Phase Scheduling Stage 321
11.5.3 Scenarios to Characterize Data-Dependent TFs 324
11.5.4 Platform Simulation Environment 326
11.6 3D Rendering QoS Application 327
11.7 Experimental Results 329
11.7.1 Gray-Box Model 329
11.7.2 Scenario Selection 329
11.7.3 Reference Cases for Comparison 331
11.7.4 Discussion of All Results 332
11.8 Conclusions 335

12 ILP-Based Resource-Aware Compilation 337
Jens Palsberg and Mayur Naik
12.1 Introduction 337
12.2 Examples 339

Petitioner Samsung Ex-1010, 0012

12.2.1 Instruction Scheduling 341
12.2.2 Energy Efficiency 343
12.2.3 Code-Size Minimization 345
12.2.4 Register Allocation 348
12.3 Open Problems 350
12.3.1 Combination 351
12.3.2 Correctness 352
12.3.3 Relationships with Other Approaches 353
12.4 Conclusions 354

PART III METHODOLOGY AND APPLICATIONS 355

13 Component-Based Design for Multiprocessor Systems-on-Chip 357
Wander O. Cesário and Ahmed A. Jerraya
13.1 From ASIC to System and Network on Chip 357
13.1.1 Applications for MPSoC 358
13.2 Basics for MPSoC Design 359
13.2.1 MPSoC Software Architectures 361
13.2.2 MPSoC Design Methods 362
13.2.3 Component Interface Abstraction 364
13.2.4 Component-Based Approach 365
13.3 Design Models for Component Abstraction 367
13.3.1 Conceptual Design Flow 368
13.3.2 Virtual Architecture Model 368
13.3.3 Target Architecture Model 370
13.3.4 The Hardware/Software Wrapper Concept 370
13.4 Component-Based Design Environment 371
13.4.1 Hardware Generation 371
13.4.2 Memory Wrapper Generation 375

Petitioner Samsung Ex-1010, 0013

13.4.3 Software Wrapper Generation 378
13.4.4 Simulation Model Generation 382
13.5 Component-Based Design of a VDSL Application 385
13.5.1 The VDSL Modem Architecture Specification 385
13.5.2 Virtual Architecture Specification 387
13.5.3 Resulting MPSoC Architecture 388
13.5.4 Evaluation 391
13.6 Conclusions 392

14 MPSoCs for Video 395
Santanu Dutta, Jens Rennert, Teihan Lv, Jiang Xu, Shengqi Yang, and Wayne Wolf
14.1 Introduction 395
14.2 Multimedia Algorithms 396
14.2.1 Compression 396
14.2.2 Recognition 401
14.3 Architectural Approaches to Video Processing 402
14.4 Optimal CPU Configurations and Interconnections 406
14.4.1 Monolithic CPUs 406
14.4.2 Reconfigurable CPUs 407
14.4.3 Networked CPUs 408
14.4.4 Smart Interconnects 409
14.4.5 Software Support 409
14.5 The Challenges of SoC Integration and IP Reuse 410
14.6 The Panacea/Promise of Platform-Based Design 413
14.7 The Ever Critical Communication Bus Structures 416
14.7.1 PNX-8500 Structure 417
14.8 Design for Testability 421
14.9 Application-Driven Architecture Design 423
14.9.1 Application Characterization 423
14.9.2 Architectural Characterization 424
14.10 Conclusions 429

Petitioner Samsung Ex-1010, 0014

15 Models of Computation for Systems-on-Chips 431
JoAnn M. Paul and Donald E. Thomas

15.1 Introduction 431
15.1.1 Evolution of Models of Computation 432
15.1.2 What Makes a Good Model of Computation? 434
15.1.3 Toward an MoC for SoCs 436
15.2 MoC Classifications 437
15.2.1 Conventional Classifications 437
15.2.2 Classification Based on Applicability to Computer Design 445
15.3 Models of Computation and Computer Models 448
15.4 Modeling Environment for Software and Hardware 451
15.4.1 Programmable Heterogeneous Multiprocessors 452
15.4.2 A New Simulation Foundation 454
15.4.3 Description of Layered Simulation 455
15.5 Conclusions 462

16 Metropolis: A Design Environment for Heterogeneous Systems 465
Felice Balarin, Harry Hsieh, Luciano Lavagno, Claudio Passerone, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Yosinori Watanabe, and Guang Yang

16.1 Introduction 465
16.1.2 Related Work 471
16.2 The Metropolis Meta-Model 473
16.2.1 Function Modeling 474
16.2.2 Constraint Modeling 475
16.2.3 Architecture Modeling 477
16.2.4 Mapping 478
16.2.5 Recursive Paradigm of Platforms 480
16.3 Tools 481
16.3.1 Simulation 482

Petitioner Samsung Ex-1010, 0015

16.3.2 **Property Verification** 482
16.3.3 **Synthesis** 483
16.4 **The Picture-in-Picture Design Example** 484
16.4.1 **Functional Description** 484
16.4.2 **Architectural Platform** 489
16.4.3 **Mapping Strategy** 492
16.5 **Conclusions** 495

Glossary 497
References 513
Contributor Biographies 557
Subject Index 567

Petitioner Samsung Ex-1010, 0016

Preface

This book had its origins in the Multiprocessor System-on-Chip (MPSoC) Workshop, which has been held every summer since 2001. We started the workshop to bring together a broad range of people who need to be involved in SoC design. MPSoCs are much more complex than ASICs and so require traditionally separate disciplines to converge. Computer architecture, real-time operating systems, embedded software, computer-aided design, and circuit design are all fields that must contribute to successful MPSoC designs.

An outstanding cross-section of experts from these disciplines attended the workshop and gave tutorial lectures that introduced their fields to the broader audience. We started to talk about how to make use of all the information that was presented at the workshop and make it available to a broader audience. After some debate we decided to organize a book of contributed chapters.

The contents of the book evolved somewhat as we developed it. Some of the workshop speakers didn't have enough room in their schedules for the considerable effort it takes to write a chapter. As a result, we ended up asking several other groups of people to write chapters. This allowed us to bring more perspectives into the discussion of MPSoC design.

Thanks to a great deal of effort, the book is now a reality. We believe that it will be useful to both professionals who want to know more about multiprocessor SoCs and students who are studying the subject. We hope that this collection of material will be informative and long-lasting. We'd like to thank the IEEE Circuits and Systems Society and the European Design and Automation Association for their sponsorship of the MPSoC Workshop over the years. We would like to thank all the participants—speakers and attendees—of the MPSoC workshops. We would

Petitioner Samsung Ex-1010, 0017

like to thank all of the authors for their work in creating these chapters. We would like to thank the book's reviewers: Nikil Dutt, University of California, Irvine; Axel Jantsch, Royal Institute of Technology (Stockholm, Sweden); Dan Phillips, Rochester Institute of Technology; Miodrag Potkonjak, UCLA; Gerald E. Sobelman, University of Minnesota; and James M. Ziobro, Rochester Institute of Technology. And we would like to thank the staff at Morgan Kaufman/Elsevier for their tireless efforts to make this book a reality.

Ahmed Amine Jerraya, Grenoble, France
Wayne Wolf, Princeton, New Jersey

Petitioner Samsung Ex-1010, 0018

1 CHAPTER

The What, Why, and How of MPSoCs

Ahmed Amine Jerraya and Wayne Wolf

1.1 INTRODUCTION

Multiprocessor systems-on-chips (MPSoCs) are the latest incarnation of very large-scale integration (VLSI) technology. A single integrated circuit can contain over 100 million transistors, and the International Technology Roadmap for Semiconductors predicts that chips with a billion transistors are within reach. Harnessing all this raw computing power requires designers to move beyond logic design into computer architecture. The demands placed on these chips by applications require designers to face problems not confronted by traditional computer architecture: real-time deadlines, very low-power operation, and so on. These opportunities and challenges make MPSoC design an important field of research.

The other chapters in this book examine various aspects of MPSoC design in detail. This chapter surveys the field and tries to put the problems of MPSoC design into perspective. We will start with a brief introduction to MPSoC applications. We will then look at the hardware and software architectures of MPSoCs. We will conclude with a survey of the remaining chapters in the book.

1.2 WHAT ARE MPSoCS?

We first need to define *system-on-chip* (SoC). An SoC is an integrated circuit that implements most or all of the functions of a complete electronic system. The most fundamental characteristic of an SoC is complexity. A memory chip may have many transistors, but its regular structure makes it a component and not a system.

Petitioner Samsung Ex-1010, 0019

Exactly what components are assembled on the SoC varies with the application. Many SoCs contain analog and mixed-signal circuitry for input/output (I/O). Although some high-performance I/O applications require a separate analog interface chip that serves as a companion to a digital SoC, most of an SoC is digital because that is the only way to build such complex functions reliably. The system may contain memory, instruction-set processors (central processing units [CPUs]), specialized logic, busses, and other digital functions. The architecture of the system is generally tailored to the application rather than being a general-purpose chip: we will discuss the motivations for custom, heterogeneous architectures in the next section.

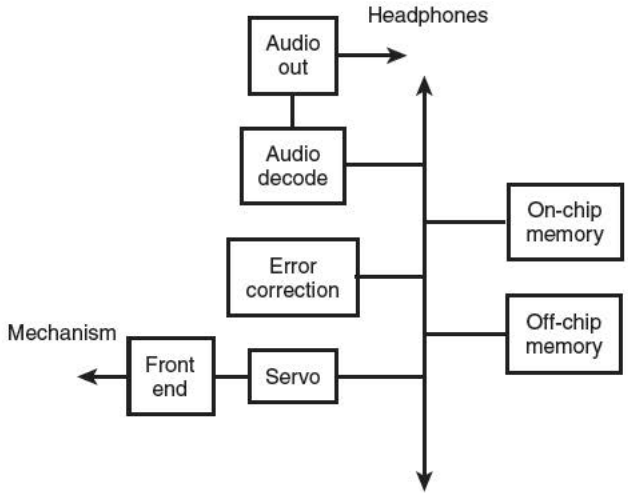
Systems-on-chips can be found in many product categories ranging from consumer devices to industrial systems:

- Cell phones use several programmable processors to handle the signal-processing and protocol tasks required by telephony. These architectures must be designed to operate at the very low-power levels provided by batteries.
- Telecommunications and networking use specialized systems-on-chips, such as network processors, to handle the huge data rates presented by modern transmission equipment.
- Digital televisions and set-top boxes use sophisticated multiprocessors to perform real-time video and audio decoding and user interface functions.
- Television production equipment uses systems-on-chips to encode video. Encoding high-definition video in real time requires extremely high computation rates.
- Video games use several complex parallel processing machines to render gaming action in real time.

These applications do not use general-purpose computer architectures either because a general-purpose machine is not cost-effective or because it would simply not provide the necessary performance. Consumer devices must sell for extremely low prices. Today, digital video disc (DVD) players sell for US $50, which leaves very little money in the budget for the complex video decoder and control system that playing DVDs requires. At the high end, general-purpose machines simply can't keep up with the data rates for high-end video and networking; they also have a hard time providing reliable real-time performance.

So what is an MPSoC? It is simply a system-on-chip that contains multiple instruction-set processors (CPUs). In practice, most SoCs are MPSoCs because it is too difficult to design a complex system-on-chip without making use of

Petitioner Samsung Ex-1010, 0020

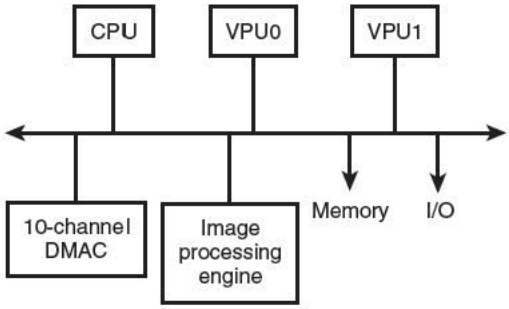


1-1 FIGURE Architecture of a CD/MP3 player.

multiple CPUs. We will discuss the rationale for multiprocessors in more detail in the next section.

Figure 1-1 shows a block diagram for a typical compact disc/MPEG layer-3 (CD/MP3) player, a chip that controls a CD drive and decodes MP3 audio files. The architecture of a DVD player is more complex but has many similar characteristics, particularly in the early stages of processing. This block diagram abstracts the interconnection between the different *processing elements* (PEs). Although interconnect is a significant implementation concern, we want first to focus on the diversity of the PEs used in an SoC. At one end of the processing chain is the mechanism that controls the CD drive. A small number of analog inputs from the laser pickup must be decoded both to be sure that the laser is on track and to read the data from the disc. A small number of analog outputs controls the lens and sled to keep the laser on the data track, which is arranged as a spiral around the disc. Early signal conditioning and simple signal processing is done in analog circuitry because that is the only cost-effective means of meeting the data rates. However, most of the control circuitry for the drive is performed digitally. The CD player is a triumph of signal processing over mechanics—a very cheap and low-quality mechanism is controlled by sophisticated algorithms to very fine tolerances. Several control loops with 16 or more taps are typically performed by a digital signal processor (DSP) in order to control the CD drive mechanism. Once

Petitioner Samsung Ex-1010, 0021



1-2 FIGURE Architecture of the Sony Playstation 2 Emotion Engine.

the raw bits have been read from the disc, error correction must be performed. A modified Reed-Solomon algorithm is used; this task is typically performed by a special-purpose unit because of the performance requirements. After error correction, the MP3 data bits must be decoded into audio data; typically other user functions such as equalization are performed at the same time. MP3 decoding can be performed relatively cheaply, so a relatively unsophisticated CPU is all that is required for this final phase. An analog amplifier sends the audio to headphones.

Figure 1-2 shows the architecture of the Emotion Engine chip from the Sony PlayStation 2 [1]. The Emotion Engine is one of several complex chips in the PlayStation 2. It includes a general-purpose CPU that executes the millions of instructions per second (MIPS) instruction set and two vector processing units, VPU0 and VPU1. The two vector processing units have different internal architectures. The chip contains 5.8 million transistors, runs at 300 MHz, and delivers 5.5 Gflops.

Why do we care about performance? Because most of the applications for which SoCs are used have precise performance requirements. In traditional interactive computing, we care about speed but not about deadlines. Control systems, protocols, and most real-world systems care not just about average performance but also that tasks are done by a given deadline. The vast majority of SoCs are employed in applications that have at least some real-time deadlines. Hardware designers are used to meeting clock performance goals, but most deadlines span many clock cycles.

Why do we care about energy? In battery-operated devices, we want to extend the life of the battery as long as possible. In non-battery-operated devices, we still care because energy consumption is related to cost. If a device utilizes too much power, it runs too hot. Beyond a certain operating temperature, the chip must be

Petitioner Samsung Ex-1010, 0022

put in a ceramic package. Ceramic packages are much more expensive than plastic packages.

The fact that an MPSoC is a multiprocessor means that software design is an inherent part of the overall chip design. This is a big change for chip designers, who are used to coming up with hardware solutions to chip design problems. In an MPSoC, either hardware or software can be used to solve a problem; which is best generally depends on performance, power, and design time. Designing software for an MPSoC is also a big change for software designers. Software that will be shipped as part of a chip must be extremely reliable. That software must also be designed to meet many design constraints typically reserved for hardware, such as hard timing constraints and energy consumption. This melding of hardware and software design disciplines is one of the things that makes MPSoC design interesting and challenging.

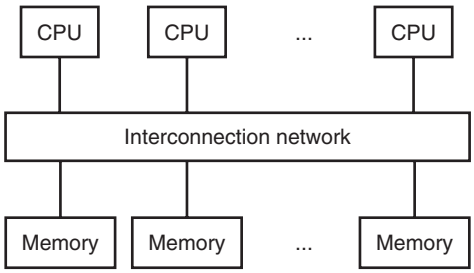
The fact that most MPSoCs are heterogeneous multiprocessors makes them harder to program than traditional symmetric multiprocessors. Regular architectures are much easier to program. Scientific multiprocessors have also gravitated toward a shared-memory model for programmers. Although these regular, simple architectures are simple for programmers, they are often more expensive and less energy efficient than heterogeneous architectures. The combination of high reliability, real-time performance, small memory footprint, and low-energy software on a heterogeneous multiprocessor makes for a considerable challenge in MPSoC software design.

Many MPSoCs need to run software that was not developed by the chip designers. Because standards guarantee large markets, multichip systems are often reduced to SoCs only when standards emerge for the application. However, users of the chip must add their own features to the system to differentiate their products from competitors who use the same chip. This requires running software that is developed by the customer, not the chip designer. Early VLSI systems with embedded processors generally used very crude software environments that would have been impossible for outside software designers to use. Modern MPSoCs have better *development environments*, but creating a different software development kit for each SoC is in itself a challenge.

1.3 WHY MPSoCS?

The typical MPSoC is a *heterogeneous multiprocessor*: there may be several different types of PEs, the memory system may be heterogeneously distributed around the machine, and the interconnection network between the PEs and the memory

Petitioner Samsung Ex-1010, 0023



1-3 A generic shared-memory multiprocessor.

FIGURE

may also be heterogeneous. MPSoCs often require large amounts of memory. The device may have embedded memory on-chip as well as relying on off-chip commodity memory.

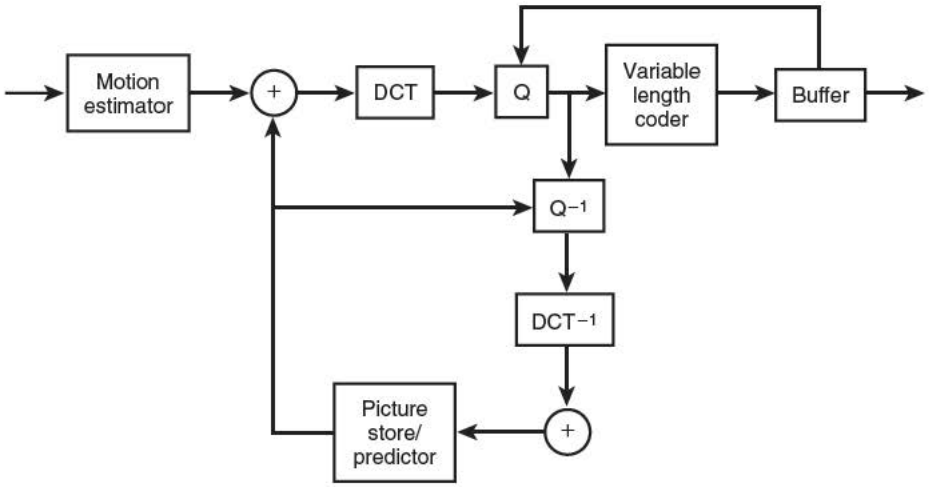
We introduced two examples of SoCs in the last section and they implement, in fact, heterogeneous multiprocessors. In contrast, most scientific multiprocessors today are much more regular than the typical MPSoC. Figure 1-3 shows the traditional view of a *shared-memory multiprocessor* [2]—a pool of processors and a pool of memory are connected by an interconnection network. Each is generally regularly structured, and the programmer is given a regular programming model. A shared-memory model is often preferred because it makes life simpler for the programmer. The Raw architecture [3] is a recent example of a regular architecture designed for high-performance computation.

Why not use a single *platform* for all applications? Why not build SoCs like field programmable gate arrays (FPGAs), in which a single architecture is built in a variety of sizes? And why use a multiprocessor rather than a uniprocessor, which has an even simpler programming model?

Some relatively simple systems are, in fact, uniprocessors. The personal digital assistant (PDA) is a prime example. The architecture of the typical PDA looks something like a PC, with a CPU, peripherals, and memory attached to a bus. A PDA runs many applications that are small versions of desktop applications, so the resemblance of the PDA platform to the PC platform is important for software development.

However, uniprocessors may not provide enough performance for some applications. The simple database applications such as address books that run on PDAs can easily be handled by modern uniprocessors. But when we move to real-time video or communications, multiprocessors are generally needed to keep up with

Petitioner Samsung Ex-1010, 0024



1-4 FIGURE Block diagram of MPEG-2 encoding.

the incoming data rates. Multiprocessors provide the computational concurrency required to handle concurrent real-world events in real time.

Embedded computing applications typically require real concurrency, not just the apparent concurrency of a multitasking operating system running on a uniprocessor. *Task-level parallelism* is very important in embedded computing. Most of the systems that rely on SoCs perform complex tasks that are made up of multiple phases. For example, Figure 1-4 shows the block diagram for MPEG-2 encoding [4]. Video encoding requires several operations to run concurrently: motion estimation, discrete cosine transform (DCT), and Huffman coding, among others. Video frames typically enter the system at 30 frames/sec. Given the large amount of computation to be done on each frame, these steps must be performed in parallel to meet the deadlines. This type of parallelism is relatively easy to leverage since the system specification naturally decomposes the problem into tasks. Of course, the decomposition that is best for specification may not be the best way to decompose the computation for implementation on the SoC. It is the job of software or hardware design tools to massage the decomposition based on implementation costs. But having the original parallelism explicitly specified makes it much easier to repartition the functionality during design.

But why not use a symmetric multiprocessor to provide the required performance? If we could use the same architecture for many different applications, we could manufacture the chips in even larger volumes, allowing lower prices.

Petitioner Samsung Ex-1010, 0025

Programmers could also more easily develop software since they would be familiar with the platforms and they would have a richer tool set. And a symmetric processor would make it easier to map an application onto the architecture.

However, we cannot directly apply the scientific computing model to SoCs. SoCs must obey several constraints that do not apply to scientific computation:

- They must perform real-time computations.
- They must be area-efficient.
- They must be energy-efficient.
- They must provide the proper I/O connections.

All these constraints push SoC designers toward heterogeneous multiprocessors. We can consider these constraints in more detail.

Real-time computing is much more than high-performance computing. Many SoC applications require very high performance—consider high-definition video encoding, for example—but they also require that the results be available at a predictable rate. Rate variations can often be solved by adding buffer memory, but memory incurs both area and energy consumption costs. Making sure that the processor can produce results at predictable times generally requires careful design of all the aspects of the hardware: instruction set, memory system, and system bus. It also requires careful design of the software, both to take advantage of features of the hardware and to avoid common problems like excessive reliance on buffering.

Real-time performance also relies on predictable behavior of the hardware. Many mechanisms used in general-purpose computing to provide performance in an easy programming model make the system's performance less predictable. Snooping caching, for example, dynamically manages cache coherency but at the cost of less predictable delays since the time required for a memory access depends on the state of several caches. One way to provide predictable performance and high performance is to use a mechanism that is specialized to the needs of the application: specialized memory systems or application-specific instructions, for example. And since different tasks in an application often have different characteristics, different parts of the architecture often need different hardware structures.

Heterogeneous multiprocessors are more area-efficient than symmetric multiprocessors. Many scientific computing problems distribute homogeneous data across multiple processors; for example, they may decompose a matrix in parallel using several CPUs. However, the task-level parallelism that embedded

Petitioner Samsung Ex-1010, 0026

computing applications display is inherently heterogeneous. In the MPEG block diagram, as with other applications, each block does something different and has different computational requirements.

Although application heterogeneity does not inherently require using a different type of processor for each task, doing so can have significant advantages. A special-purpose PE may be much faster and smaller than a programmable processor; for example, several very small and fast motion estimation machines have been developed for MPEG. Even if a programmable processor is used for a task, specialized CPUs can often improve performance while saving area. For example, matching the CPU datapath width to the native data sizes of the application can save a considerable amount of area. Choosing a cache size and organization to match the application characteristics can greatly improve performance.

Memory specialization is an important technique for designing efficient architectures. A general-purpose memory system can try to handle special cases on the fly using information gathered during execution, but they do so at a considerable cost in hardware. If the system architect can predict some aspect of the memory behavior of the application, it is often possible to reflect those characteristics in the architecture. Cache configuration is an ideal example—a considerably smaller cache can often be used when the application has regular memory access patterns.

Most SoC designs are power-sensitive, whether due to environmental considerations (heat dissipation) or to system requirements (battery power). As with area, specialization saves power. Stripping away features that are unnecessary for the application reduces energy consumption; this is particularly true for leakage power consumption. Scientific multiprocessors are standard equipment that are used in many different ways; each installation of a supercomputer may perform a different task. In contrast, SoCs are mass-market devices due to the economics of VLSI manufacturing. The design tweaks that save power for a particular architecture feature can therefore be replicated many times during manufacturing, amortizing the cost of designing those power-saving features.

SoCs also require specialized I/O. The point of an SoC is to provide a complete system. One would hope that input and output devices could be implemented in a generic fashion given enough transistors; to some extent, this has been done for FPGA I/O pads. But given the variety of physical interfaces that exist, it can be difficult to create customizable I/O devices effectively.

One might think that increasing transistor counts might argue for a trend away from heterogeneous architectures and toward regularly structured machines. But applications continue to soak up as much computational power as can be supplied by Moore's law. Data rates continue to go up in most applications, for example, data communication, video, audio. Furthermore, new devices

Petitioner Samsung Ex-1010, 0027

increasingly combine these applications. A single device may perform wireless communication, video compression, and speech recognition. SoC designers will start to favor regular architectures only when the performance pressure from application eases and the performance available from integrated circuits catches up. It does not appear that customers' appetites will shrink any time soon.

1.4 CHALLENGES

Before delving into MPSoC design challenges in more detail, let us take a few moments to summarize a few major challenges in the field.

Software development is a major challenge for MPSoC designers. The software that runs on the multiprocessor must be high performance, real time, and low power. Although much progress has been made on these problems, much remains to be done. Furthermore, each MPSoC requires its own software development environment: compiler, debugger, simulator, and other tools.

Task-level behavior also provides a major and related challenge for SoC software. As mentioned above, task-level parallelism is both easy to identify in SoC applications and important to exploit. *Real-time operating systems* (RTOSs) provide scheduling mechanisms for tasks, but they also abstract the process. The detailed behavior of a task—how it accesses memory, its flow of control—can influence its execution time and therefore the system schedule that is managed by the RTOS. We need a better understanding of how to abstract tasks properly to capture the essential characteristics of their low-level behavior for system-level analysis.

Networks-on-chips have emerged over the past few years as an architectural approach to the design of single-chip multiprocessors. A network-on-chip uses packet networks to interconnect the processors in the SoC. Although a great deal is known about networks, traditional network design assumes relatively little about the characteristics of the traffic on the network. SoC applications can often be well characterized; that information should be useful in specializing the network design to improve cost/performance/power.

FPGAs have emerged as a viable alternative to application-specific integrated circuits (ASICs) in many markets. FPGA fabrics are also starting to be integrated into SoCs. The FPGA logic can be used for custom logic that could not be designed before manufacturing. This approach is a good complement to software-based customization. We need to understand better where to put FPGA fabrics into SoC architectures so they can be most effectively used. We also need tools to help designers understand performance and allocation using FPGAs as processing elements in the multiprocessor.

Petitioner Samsung Ex-1010, 0028

As SoCs become more sophisticated, and particularly as they connect to the Internet, security becomes increasingly important. Security breaches can cause malfunctions ranging from annoying to life-threatening. Hardware and software architectures must be designed to be secure. Design methodologies must also be organized to ensure that security considerations are taken into account and that security-threatening bugs are not allowed to propagate through the design.

Finally, MPSoCs will increasingly be connected into networks of chips. Sensor networks are an example of networks of chips; automotive and avionics systems have long used networks to connect physically separated chips. Clearly, networking must be integrated into these chips. But, more important, MPSoCs that are organized into networks of chips do not have total control over the system. When designing a single chip, the design team has total control over what goes onto the chip. When those chips are assembled into networks, the design team has less control over the organization of the network as a whole. Not only may the SoC be used in unintended ways, but the network of the configuration may change over time as nodes are added and deleted. Design methodologies for these types of chips must be adapted to take into account the varying environments in which the chips will have to operate.

1.5 DESIGN METHODOLOGIES

In the preceding sections, we explained the "What" and "Why" of MPSoC architectures and presented the main challenges designers must face for the "How" to design an MPSoC. It is clear that many kinds of tools and specialized design methodologies are needed. Although advanced tools and methodologies exist today to solve partial problems, much remains to be done when considering heterogeneous MPSoC architectures as a whole. On this new design scenario, fast design time, higher level abstractions, predictability of results, and meeting design metrics are the main goals.

Fast design time is very import in light of typical applications for MPSoC architectures—game/network processors, high-definition video encoding, multimedia hubs, and base-band telecom circuits, for example—that have particularly tight time-to-market and time window constraints.

System-level modeling is the enabling technology for MPSoC design. Register-transfer level (RTL) models are too time consuming to design and verify when considering multiple processor cores and associated peripherals—a higher abstraction level is needed on the hardware side. When designers use RTL abstractions, they can produce, on average, the equivalent of 4 to 10 gates per line of RTL code.

Petitioner Samsung Ex-1010, 0029

Thus, hypothetically, in order to design a 100 million-gate MPSoC circuit using only RTL code—even if we assume that 90% of this code can be reused—more than 1 million lines of code would need to be written to describe the remaining 10 million gates for the custom part. Of course, this tremendous design effort is unrealistic for most MPSoC target markets.

MPSoCs may use hundreds of thousands of lines of dedicated software and complex software development environments; programmers cannot use mostly low-level programming languages anymore—higher level abstractions are needed on the software side too. Design components for MPSoC are heterogeneous: they come from different design domains, have different interfaces, and are described using different languages at different refinement levels and have different granularities. A key issue for every MPSoC design methodology is the definition of a good system-level model that is capable of representing all those heterogeneous components along with local and global design constraints and metrics.

High-level abstractions make global MPSoC design methodologies possible by hiding precise circuit behavior—notably accurate timing information—from system-level designers and tools. However, MPSoCs are mostly targeted for real-time applications in which accurate performance information must be available at design time in order to respect deadlines. Thus, when considering the huge design space allowed by MPSoC architectures, high-level design metrics and performance estimation are essential parts in MPSoC design methodologies. This notwithstanding, high-level estimation metrics and evaluation remains an active research subject since a system's design metrics are not easy to compose from design metrics of its components.

MPSoC design is a complex process involving different steps at different abstraction levels. Design steps can be grouped into two major tasks: design space exploration (hardware/software partitioning, selection of architectural platform and components) and architecture design (design of components, hardware/software interface design). The overall design process must consider strict requirements, regarding time-to-market, system performance, power consumption, and production cost. The reuse of predesigned components from several vendors—for hardware and software parts—is necessary for reducing design time, but their integration into a system also presents a variety of challenges.

A complete design flow for MPSoCs includes refinement processes that require multiple competences and tools because of the complexity and diversity of the current applications. Current (and previous) work tries to reduce the gap between different design steps [5] and to master the integration of heterogeneous components [6], including hardware and software parts. Existing approaches deal only with a specific part of the MPSoC design flow. A full system level flow

Petitioner Samsung Ex-1010, 0030

is quite complex, and to our knowledge very few existing tools cover both system design space exploration and system architecture design.

1.6 HARDWARE ARCHITECTURES

We can identify several problems in MPSoC architecture starting from the bottom and working to the highest architectural levels:

- Which CPU do you use? What instruction set and cache should be used based on the application characteristics?
- What set of processors do you use? How many processors do you need?
- What interconnect and topology should be used? How much bandwidth is required? What *quality-of-service* (QoS) characteristics are required of the network?
- How should the memory system be organized? Where should memory be placed and how much memory should be provided for different tasks?

The following discussion describes several academic and industrial research projects that propose high-performance MPSoC architectures for high-performance applications.

Philips Nexperia™ DVP [7] is a flexible architecture for digital video applications. It contains two software-processing units, a very long instruction word (VLIW) media processor (32-bit or 64-bit @ 100 to 300+MHz) and a MIPS core (32-bit or 64-bit @ 50 to 300+MHz). It also contains a library of dedicated hardware processing units (image coprocessors, DSPs, universal asynchronous receiver transmitter [UART], 1394, universal serial bus [USB], and others). Finally, it integrates several system busses for specialized data transfer, a peripheral interface (PI) bus and a digital video platform (DVP) memory bus for intensive shared data transfer. This fixed memory architecture, besides the limited number and types of the incorporated CPU cores, reduces the application field of the proposed platform. However, the diversity of the hardware components included in the Device Block library makes possible the design of application-specific computation.

The Texas Instruments (TI) OMAP platform is an example of an architecture model proposed for the implementation of wireless applications. The two processing units consist of an ARM9 core (@ 150 MHz) and a C55x DSP core (@ 200 MHz). Both of them have a 16-Kb I-cache, an 8-Kb D-cache, and a two-way set

Petitioner Samsung Ex-1010, 0031

associative global cache. There is a dedicated memory and traffic controller to handle data transfers. However, the proposed architecture for data transfer is still simple and the concurrency still limited (only two processing cores).

Virtex-II Pro™ from Xilinx is a recent FPGA architecture that embeds 0, 1, 2, or 4 PowerPC cores. Each PowerPC core occupies only about 2% of the total die area. The rest of the die area can be used to implement system busses, interfaces, and hardware IPs. The IP library and development tools provided by Xilinx especially support the IBM CoreConnect busses.

Given the examples of commercially available MPSoC chips, we see that most of them:

- limit the number and types of integrated processor cores
- provide a fixed or not well-defined memory architecture
- limit the choice of interconnect networks and available IPs
- do not support the design from a high abstraction level.

1.7 SOFTWARE

The software for MPSoC needs to be considered from the following three viewpoints: the programmer's viewpoint, the software architecture and design reuse viewpoint, and the optimization viewpoint.

1.7.1 Programmer's Viewpoint

From the viewpoint of the application programmer, an MPSoC provides a parallel architecture that consists of processors connected with each other via a communication network. Thus, to exploit the parallel architecture, parallel programming is required. Conventionally, there are two types of parallel programming model: shared-memory programming and *message-passing* programming. OpenMP and *message-passing interface* (MPI) are examples, respectively.

When using conventional parallel programming models for SoC, we face conventional issues in parallel programming, e.g., shared memory versus message passing. We also need to identify the difference between conventional parallel programming and MPSoC programming. We need to exploit the characteristics specific to MPSoC to use the parallel programming models in a more efficient way.

Petitioner Samsung Ex-1010, 0032

Differences appear in two aspects of MPSoC software design: application and architecture. Conventional parallel programming models need to support any type of program. Thus, they have a huge number of programming features, e.g., the user-defined complex data types and multicasting in MPI. However, MPSoC is application-specific. That is, an MPSoC supports only one or a set of fixed applications. Thus, for parallel programming, the designer does not need full-featured parallel programming models. Instead, we need to be able to find a suitable or application-specific subset of a parallel programming model for MPSoC programming.

MPSoC architectures have two main characteristics that differ from conventional multiprocessor architectures. One is heterogeneity and the other is massive parallelism. MPSoC can have different types of processors and any arbitrary topology of interconnection of processors. An MPSoC can also have a massive number of (fine-grained) processors, or processing elements. Thus, considering heterogeneous multiprocessor architecture with massive parallelism, parallel programming for MPSoC is more complicated than conventional parallel programming. To cope with this difficulty, it is necessary to obtain efficient models and methods of MPSoC programming.

1.7.2 Software Architecture and Design Reuse Viewpoint

We define software architecture to be the component that enables application software to run on the MPSoC architecture. The software architecture includes the *middleware* (for communication), the operating system (OS), and the *hardware abstraction layer* (HAL). HAL is the software component that is directly dependent on the underlying processor and peripherals. Examples of HAL include context switching, bus drivers, configuration code for the memory management unit (MMU), and interrupt service routines (ISRs).

From the viewpoint of application software, the architecture provides a virtual machine on which the application software runs. The basic role of software architecture is to enable (1) communication between computation units in the application software, i.e., software tasks; (2) task scheduling to provide task-level parallelism described in parallel programming models; and (3) external event processing, e.g., processing interrupts.

The *application programming interfaces* (APIs) of the middleware, OS, and HAL provide an abstraction of the underlying hardware architecture to upper layers of software. The HAL API gives an abstraction of underlying processor and processor local architecture to upper layers of software including application software,

Petitioner Samsung Ex-1010, 0033

middleware, and OS. The middleware and OS API gives an abstraction of underlying multiprocessor architecture to application software.

The hardware abstraction represented by the API can play the role of a contract between software and hardware designers. That is, from the viewpoint of the software designer, the hardware design guarantees at least the functionality represented by the API. Thus, the software designer can design application software based on the API, independently of the hardware design. Thus, in terms of design flow, software and hardware design can be performed concurrently. Such a concurrent design reduces the design cycle in conventional design flow, in which software and hardware design is sequential.

In terms of software design reuse, the software architecture may enable several levels of software design reuse. First, the middleware API or OS API enables reuse of application software. For instance, in the case that application software is written in an OS API, e.g., POSIX API, the same application software can be reused in other MPSoC designs in which the same OS API is provided. The same applies to the HAL API. If the same HAL API is used in OS and middleware code, OS and middleware can be reused over different designs.

Key challenges in software architecture for SoC are as follows. Determining which abstraction of MPSoC architecture is most suitable at each of the design steps. Determining how to obtain application-specific optimization of software architecture.

Since MPSoC has cost and performance constraints, the software architecture needs to be designed to minimize its overhead. For instance, instead of using a full-featured OS, the OS needs only to support the functions required by the application software and MPSoC architecture.

1.7.3 Optimization Viewpoint

MPSoC is mostly used in cost-sensitive real-time systems, e.g., cellular phone, high-definition digital television (HDTV), game stations. Thus, the designer has stringent cost requirements (chip area, energy consumption) as well as real-time performance requirements. To satisfy those requirements, MPSoC software needs to be optimized in terms of code size, execution time, and energy consumption.

Two of the major factors that affect the cost and performance of software are processor architecture and memory hierarchy. The processor architecture affects the cost and performance in terms of parallelism and application-specific processor architecture. In terms of parallelism, the processor can offer *instruction-level parallelism* (ILP) dynamically (superscalar) and statically (VLIW) and thread-level

Petitioner Samsung Ex-1010, 0034

parallelism statically (clustered VLIW) and dynamically (simultaneous multithreading [SMT]). A great deal of compiler research has exploited the above-mentioned parallelism in the processor architecture. Recently, the dynamic reconfiguration (explicitly parallel instruction computing [EPIC]) processor [8] has provided another degree of parallelism in the temporal domain.

Application-specific processor architectures can provide orders of magnitude better performance than general-purpose processor architectures. The DSP and application-specific instruction set processor (ASIP) are examples of application-specific processors. Recently, configurable processor architectures, e.g., Tensilica Xtensa, are receiving more and more attention. Compared with the DSP and ASIP, configurable processors have a basic set of general processor instructions and add application-specific instructions to accelerate the core functionality of the intended application. From the viewpoint of the designer who optimizes the software performance and cost, the design of an application-specific processor and corresponding compiler is a challenging design task, although there are a few computer-assisted design (CAD) tools to assist in the effort. More automated solutions, such as an automatic instruction selection method, are still needed.

Another factor of MPSoC software performance and cost is memory architecture. In the case of uniprocessor architecture, cache parameters such as size, associativity, replacement policies, and others affect the software performance. In the case of MPSoC architecture, there are two common types of memory architecture: shared memory and distributed memory. Shared-memory architecture usually requires local caches and cache coherency protocols to prevent the performance bottlenecks. Distributed memory can be classified into several levels according to the degree of distribution. For instance, a massively parallel MPSoC architecture, which consists of arrays of small processors (e.g., arithmetic and logic units [ALUs]) that contain small memory elements, has a finer grained distribution of memory than an MPSoC architecture, which consists of general-purpose processors, each of which has a distributed memory element.

Most of the above-mentioned issues of processor architecture and memory hierarchy have been studied in the domains of processor architecture, compiler, and multiprocessor architecture. In the domain of MPSoC, researchers consider the same problem in a different context with more design freedom in hardware architecture and with a new focus on energy consumption.

For instance, in the case of memory hierarchy design, conventional design methods assume that a set of regular structures are given. Application software code is then transformed to exploit the given memory hierarchy. However, in the case of MPSoC, the designer can change the memory hierarchy in a way specific to the given application. Thus, further optimization is possible with such hardware design freedom.

Petitioner Samsung Ex-1010, 0035

Another focus is low-power software design. Conventional software design on multiprocessor architecture focuses on performance improvement, i.e., fast or scalable performance. In the software design for MPSoC, the designer needs to reconsider the existing design problems with energy consumption in mind. For instance, when a symmetric multiprocessor architecture is used for MPSoC design, processor affinity can be defined to include the energy consumption as well as the system runtime.

1.8 THE REST OF THE BOOK

This book is divided into three major sections: (1) hardware, (2) software, and (3) applications and methodologies. Each section starts with its own introduction that describes the chapters in more detail.

The hardware section looks at architectures and the constraints imposed on those architectures. Many MPSoCs must operate at low power and energy levels; several chapters discuss low-power design. Adequate communication structures are also important given the relatively high data rates and real-time requirements in many embedded applications. The multiprocessors on SoCs may be more specialized than in traditional architectures, but SoC design must clearly rest on the foundations of computer architecture. Several chapters discuss processor and multiprocessor architectures.

The software section considers several levels of abstraction. Scheduling the tasks on the multiprocessor is critical to efficient use of the processors. RTOSs and more custom software synthesis methods are important mechanisms for managing the tasks in the system. Compilers are equally important, and several chapters discuss compilation for size, time constraints, memory system efficiency, and other objectives.

The applications and methodology section looks at sample applications such as video computing. It also considers design methodologies and the abstractions that must be built to manage the design process.

Petitioner Samsung Ex-1010, 0036

PART I

Hardware

The levels of integration provided by modern VLSI technology have fundamentally altered the hardware design process. When designing a 50-million or 100-million chip, design teams can no longer afford to design a gate-at-a-time or a wire-at-a-time. Chip architects and designers must learn to think in much larger blocks of intellectual property (IP). Some of those blocks will be synthesized from higher-level hardware description language (HDL) sources. Some blocks will be manually designed in order to provide maximum performance and minimum power consumption for critical modules. In either case, most of these IP blocks will be designed for reuse on multiple chips.

The design of multiprocessors also requires designers to change their ways of thinking. Traditional hardware design focuses on a single unit in which communication occurs at very low levels of abstraction. Multiprocessors are composed of loosely-coupled processing elements (PEs) that communicate via well-defined interfaces. Those interfaces are reflected in hardware structures that may themselves be complex modules. Those interfaces must, of course, be designed with the system software in mind.

MPSOC designers must choose their components and design architectures to meet tough requirements. Energy/power consumption is often the toughest constraint in modern designs. High performance is always important, since it makes little sense to incur the expense of system-on-chip design without a compelling performance requirement. And cost is still an important factor that can make or break the feasibility of an MPSOC project.

Chapter 2, by Mary Jane Irwin, Luca Benini, Vijaykrishnan Narayanan, and Mahmut Kandemir, concentrates on energy and power consumption. It both looks at the sources of energy consumption in nanometer-scale VLSI and potential approaches to mitigating energy and power consumption problems. The

Petitioner Samsung Ex-1010, 0037

traditional source of energy consumption in CMOS is dynamic dissipation but leakage is becoming increasingly important. Controlling leakage in particular encourages architectural solutions.

Chapter 3, by Luca Benini and Giovanni De Micheli, describes network-on-chip-based design. Networks-on-chips are becoming increasingly popular because they help solve problems at both the circuit level and the architecture level. An on-chip network can be built from carefully designed components that encapsulate high-performance circuits. Packet-based communication also helps structure the application's communication. Because different applications require different structures and characteristics, Benini and De Micheli discuss methodologies for designing networks-on-chips.

Chapter 4, by Eric Rotenberg and Aravindh Anandtaraman, surveys processor architectures. Many of the architectural techniques from general-purpose computing have found their way into embedded processors. But because of the strict characteristics of embedded applications—hard real-time, low-energy operation—system architects must carefully consider how the mechanisms inside the CPUs they use as building blocks will affect their embedded software and overall application characteristics.

Chapter 5, by Chris Rowen, looks at CPUs from a different angle. Configurable processors are IP blocks that have been generated for a particular application. By customizing the instruction set of a processor, architects can create a CPU that much more closely matches the characteristics of the application. Since the CPU modifications are described as instruction set extensions, they can be made easily and quickly. However, the system designer is still responsible for characterizing the application to determine what instructions are most cost-effective.

Chapter 6, by Rolf Ernst, considers directly the problem of performance modeling and analysis. The system architect must consider not only the performance of individual processes but also how the processes interact through the system schedule. Careful system modeling looks at both the detailed operation of the processes on the processing elements as well as the overall system behavior.

Chapter 7, by Kanishka Lahiri, Sujit Dey, and Anand Raghunathan, looks at communication design for MPSOCs. They describe a methodology that breaks on-chip communication design into several tasks: topology design, on-chip protocol design, and communication mapping. All these steps take both performance and energy into account.

Chapter 8, by Bishnupriya Bhattacharya, Luciano Lavagno, and Laura Vanzango, uses a case study to consider the design of on-chip networks. They work through a design of the physical layer of Hiperlan/2. They use this detailed example to illustrate their methodology for on-chip network design.

Petitioner Samsung Ex-1010, 0038

2 CHAPTER

Techniques for Designing Energy-Aware MPSoCs

Mary Jane Irwin, Luca Benini, N. Vijaykrishnan, and Mahmut Kandemir

2.1 INTRODUCTION

Power and energy consumption have become significant constraints in modern day microprocessor systems. Whereas energy-aware design is obviously crucial for battery-operated mobile and embedded systems, it is also important for desktop and server systems due to packaging and cooling requirements. In such systems, power consumption has grown from a few watts per chip to over 100 watts. As desktop/server (and even embedded) systems evolve from the uniprocessor space to the *multiprocessor system-on-chips* (MPSoCs) space, energy-aware design will take on new dimensions.

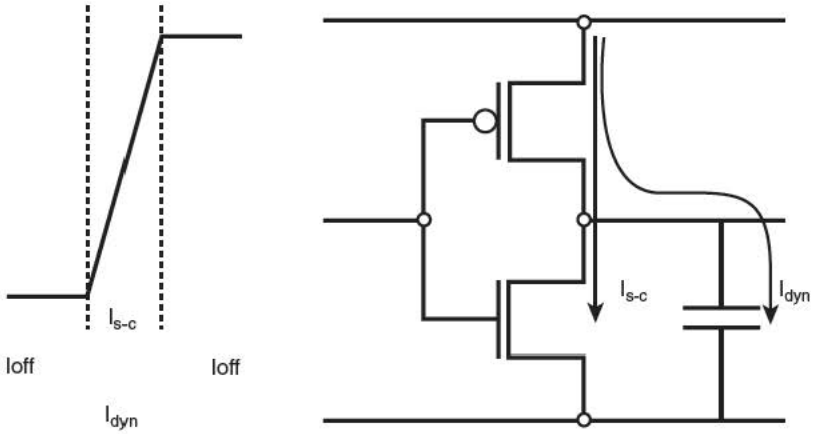
Techniques for energy and power consumption reduction have been successfully applied at all levels of the design space in uniprocessor systems: circuit, logic gate, functional unit, processor, system software, and application software levels. The primary focus has been on reducing active (dynamic) power. As technology continues to scale up accompanied by reductions in the supply and threshold voltages, the percentage of the power budget due to standby (leakage) energy has driven the development of additional techniques for reducing standby energy as well [9,10].

Figure 2-1 illustrates the components of power consumption in a simple CMOS circuit. In well-designed circuits, $I_{s\text{-}c}$ is a fixed percentage (less than 10%) of I_{on}. Thus, *active power consumption* is usually estimated as:

$$P_{Act} = C_{avg} V_{dd}^2 (Act) f_{clock}$$

where C_{avg} is average capacitive load of a transistor, V_{dd} is the power supply, f_{clock} is the operating frequency, and Act is the activity factor that accounts for the

Petitioner Samsung Ex-1010, 0039



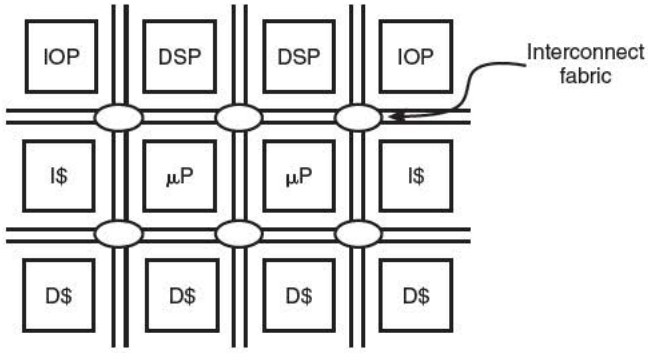
2-1 FIGURE Components of power consumption. $I_{s\text{-}c}$, short-circuit current; Ioff, leakage current; I_{dyn}, dynamic switching power.

number of devices that are actually switching (drawing current from the power supply) at a given time. This definition of active power consumption illustrates the significant benefit of supply voltage scaling—a quadratic reduction in active power consumption. However, it is important for high performance that $V_{DD} > 3V_T$ so that there is sufficient current drive. Thus, as supply voltages scale, threshold voltages must also scale, causing leakage power to increase. As an example, the leakage current increases from $20\rho A/\mu m$ when using a Taiwan Semiconductor Manufacturing Corporation (TSMC) CL018G process with a threshold voltage of 0.42 V to $13{,}000\rho A/\mu m$ when using a TSMC CL013HS process with a threshold voltage of 0.25 V. *Standby power consumption* due to subthreshold leakage current is estimated as:

$$P_{leak} = V_{dd} I_{off} K_L$$

where I_{off} is the current that flows between the supply rails in the absence of switching and K_L is a factor that accounts for the distribution/sizing of P and N devices, the stacking effect, the idleness of the device, and the design style used. In a CMOS-based style, at most half of the transistors are actually leaking whereas the remaining are ON (in the resistive region). As oxide thicknesses decrease, gate tunneling leakage will also become a significant source of standby power. This component of standby power is not included in the above equation and is not a target of the techniques presented in this chapter.

Petitioner Samsung Ex-1010, 0040



2-2 FIGURE Model MPSoC. I$, instruction cache; D$, data cache.

This chapter surveys a number of energy-aware design techniques for controlling both active and standby power consumption that are applicable to the MPSoC design space and points to emerging areas that will need special attention in the future. Our model MPSoC system is depicted in Figure 2-2. The chapter is organized as follows. Section 2.2 covers techniques that can be used to design energy-aware MPSoC processor cores. Section 2.3 focuses on the on-chip memory hierarchy, Section 2.4 on the on-chip communication system, and Section 2.5 on energy-aware software techniques that can be used in MPSoC designs. We close with conclusions.

2.2 ENERGY-AWARE PROCESSOR DESIGN

Many techniques for controlling both active and standby power consumption have been developed for processor cores [11]. Almost all of these will transition to the MPSoC design space, with some of them taking on additional importance and expanded design dimensions. Figure 2-3 shows the processor power design space for both active and standby power [12]. The second column lists techniques that are applied at design time and thus are part of the circuit fabric. The last two columns list techniques that are applied at run time, the middle column for those cases when the component is idle, and the last column when the component is in active use. (Run time techniques can additionally be partitioned into those that reduce leakage current while retaining state and those that are state-destroying.) The underlying circuit fabric must provide the "knobs" for controlling the run

Petitioner Samsung Ex-1010, 0041

	Constant Throughput/Latency		Variable Throughput/Latency
	Design Time	**Sleep Mode**	**Run Time**
Active (Dynamic) [C V_2 f]	Logic design Trans sizing Multiple V_{dd}'s	Clock gating	DFS DVS DTM
Standby (Leakage) [V I_{off}]	Stack effect Multiple V_T's Multiple t_{ox}'s	Sleep trans Multi-V_{dd} Variable V_T Input control	Variable V_T

2-3 FIGURE Processor power design space. DFS, dynamic frequency scaling; DVS, dynamic voltage scaling; DTM, dynamic thermal management.

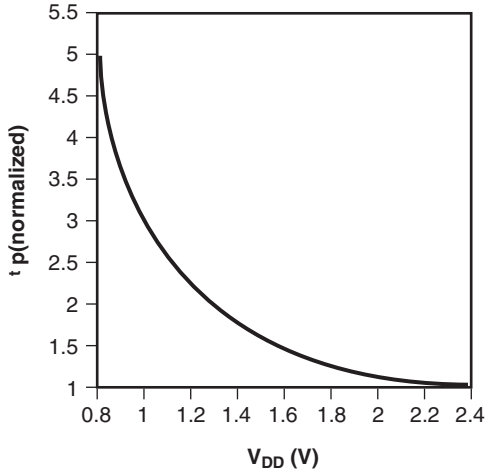
time mechanisms—by either the hardware (e.g., in the case of clock gating) or the system software (e.g., in the case of dynamic voltage scaling [DVS]). In the case of software control, the cost of transitioning from one state to another (in terms of both energy and time) and the relative energy savings need to be provided to the software for decision making.

2.2.1 Reducing Active Energy

As has already been pointed out, the best knob for controlling power is setting the supply voltage appropriately to meet the computational load requirements. Although lowering the supply voltage has a quadratic impact on active energy, it decreases systems performance since it increases gate delay, as shown in Figure 2-4 [4]. Multiple supply voltages can be used very effectively in MPSoCs since they contain multiple processors of different types with different performance requirements (e.g., as in Fig. 2-2: an MPSoC will contain high-speed digital signal processors (DSPs), moderate-speed processor cores, and low-speed I/O processors [IOPs]). Choosing the appropriate supply voltage at design time for the entire component will minimize (or even eliminate) the overhead of level converters that are needed whenever a module at a lower supply drives a module at a higher supply [13].

The most popular of the techniques for reducing active power consumption at run time is DVS combined with *dynamic frequency scaling* (DFS). Most embed-

Petitioner Samsung Ex-1010, 0042



2-4 FIGURE Delay as a function of *VDD*.

ded and mobile processors contain this feature (triggered by thermal on-chip sensors when thermal limits are being approached or by the run-time system when the CPU load changes). DFS + DVS requires a power supply control loop containing a buck converter to adjust the supply voltage and a programmable Phase locked loop (PLL) to adjust the clock [10]. As long as the supply voltage is increased before increasing the clock rate or decreased after decreasing the clock rate, the system only need stall when the PLL is relocking on the new clock rate (estimated to be around 20 msec). Future MPSoCs in which all (or selected) processing cores are DFS + DVS capable would require each to have its own converter and PLL (or ring oscillator) as well as system software capable of determining the optimum setting for each core, taking into account the computational load. The overall system will have to be designed such that cores are tolerant of periodic dropouts of other cores that are stalling when transitioning clock rates. An additional complication is that the voltage converter and PLL, being analog components, are susceptible to substrate noise induced by digital switching. Thus, uniprocessor systems typically distance the digital and analog components as much as possible to reduce this effect. Whether this technique will carry over to MPSoCs, in which the ability to separate the analog and digital components physically becomes much more complex, remains an open question.

Petitioner Samsung Ex-1010, 0043

2.2.2 Reducing Standby Energy

Several techniques have recently evolved for controlling subthreshold current, as shown in Figure 2-3. Since increasing the threshold voltage, V_T, decreases subthreshold leakage current (exponentially), adjusting V_T is one such technique. As shown in Figure 2-5, a 90-mV reduction in V_T increases leakage by an order of magnitude. Unfortunately, increasing V_T also negatively impacts gate delay. As with multiple supply voltages, multiple threshold voltages can be employed at design time or run time. At run time, multiple levels of V_T can be provided using *adaptive body-biasing* whereby a negative bias on V_{SB} increases V_T, as shown in Figure 2-5 [11]. Simultaneous DVS, DFS, and variable V_T has been shown to be an effective way to trade off supply voltage and body biasing to reduce total power—both active and standby—under variable processor loads [14]. Once again, whether similar techniques will carry over to MPSoCs remains an open question.

Another technique that will apply to reducing standby power in MPSoC's is the use of *sleep transistors* like those shown in Figure 2-6. Standby power can be greatly reduced by gating the supply rails for idle components. In normal mode (non-idle), the sleep transistors must present as small a resistance as possible (via sizing) so as not to negatively affect performance. In sleep mode

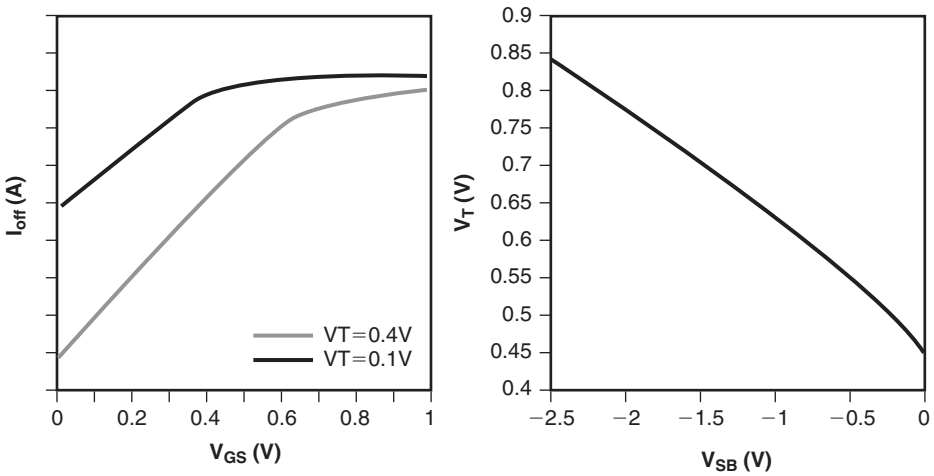
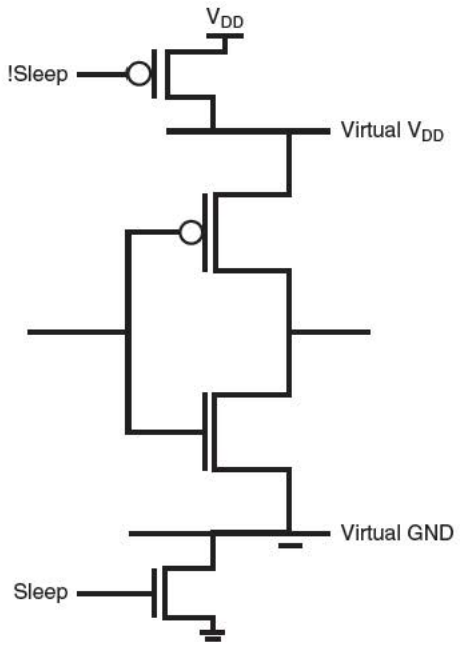


FIGURE 2-5 VT effects.

Petitioner Samsung Ex-1010, 0044



2-6 FIGURE Gating supply rails.

(idle), the transistor stack effect [10] reduces leakage by orders of magnitude. Alternatively, standby power can be completely eliminated by switching off the supply to idle components. An MPSoC employing such a technique will require system software that can determine the optimal scheduling of tasks on cores and can direct idle cores to switch off their supplies while taking into account the cost (in terms of both energy and time) of transitioning from the on-to-off and off-to-on states.

2.3 ENERGY-AWARE MEMORY SYSTEM DESIGN

In an MPSoC system, memories constitute a significant portion of the overall chip resources, as there are various memory structures ranging from private memories for individual processors to large shared cache structures. Energy is expended in these memories due to data accesses (reads/writes), coherence activity required

Petitioner Samsung Ex-1010, 0045

to maintain consistency between shared data, and leakage energy expended in just storing the data.

2.3.1 Reducing Active Energy

Many techniques have been proposed in the past to reduce cache energy consumption. Among these are partitioning large caches into smaller structures to reduce the dynamic energy [15,16] and the use of a memory hierarchy that attempts to capture most accesses in the smallest size memory. By accessing the tag and data array in series, Alpha 21164's L2 cache [17] can access the selected cache bank for energy efficiency. In Inoue et al. [18] and Powell et al. [19], *cache way-prediction* is used to reduce energy consumption of set-associative caches. *Selective way caches* [20] varies the number of ways for different application requirements. In Kin et al. [21], a small *filter cache* is placed prior to the L1 cache to reduce energy consumption. *Dynamic zero compression* [22] employs single-bit access for zero-valued byte in the cache to reduce energy consumption. Many of these techniques applied in the context of single processors are also applicable for the design of caches associated with the individual processors of the MPSoC system. In addition to the hardware design techniques, software optimizations that reduce the number of memory accesses through code and data transformations [23] can be very useful in reducing energy consumption.

2.3.2 Reducing Standby Energy

However, most of the above techniques do little to alleviate the leakage energy problem as the memory cells in all partitions and all levels of the hierarchy continue to consume leakage power as long as the power supply is maintained to them, irrespective of whether they are used or not. Various circuit technologies have been designed specifically to reduce leakage power when the component is not in use. Some of these techniques focus on reducing leakage during idle cycles of the component by turning off the supply voltage. One such scheme, *gated*-V_{dd}, was integrated into the architecture of caches [24] to shut down portions of the cache dynamically. This technique was applied at a cache block granularity in Kaxiras et al. [25] and used in conjunction with software to remove dead objects in Chen et al. [26]. However, all these techniques assume that the state (contents) of the supply-gated cache memory is lost. Although totally eliminating the supply voltage results in the state of the cache memory being lost, it is possible to apply a state-preserving leakage optimization technique if a small supply voltage is

Petitioner Samsung Ex-1010, 0046

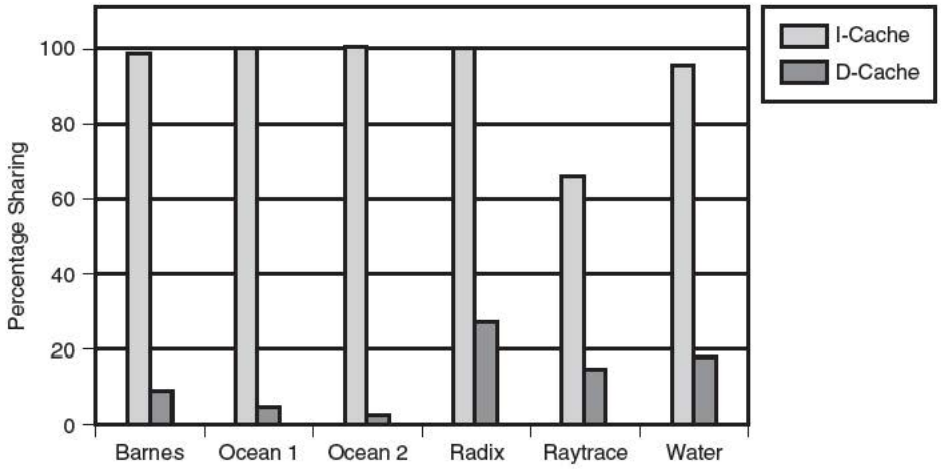
maintained to the memory cell. Many alternate implementations have recently been proposed at the circuit level to achieve such a state-preserving leakage control mechanism [27–29]. As an abstraction of these techniques, the choice between the state-preserving and state-destroying techniques depends on the relative overhead of the additional leakage required to maintain the state as opposed to the cost of restoring the lost state from other levels of the memory hierarchy.

An important requirement to reduce leakage energy using either a state-preserving or a state-destroying leakage control mechanism is the ability to identify unused resources (or data contained in them). In Yang et al. [24], the cache size is reduced (or increased) dynamically to optimize the utility of the cache. In Kaxiras et al. [25], the cache block is supply-gated if it has not been accessed for a period of time. In Zhou et al. [30], hardware tracks the hypothetical miss rate and the real miss rate by keeping the tag line active when deactivating a cache line. Then the turn-off interval can be dynamically adjusted based on such information. In Flautner et al. [31] and Kim et al. [32], dynamic supply voltage scaling is used to reduce the leakage in the unused portions of the memory. In contrast to the other schemes, their *drowsy cache* scheme also preserves data when a cache line is in low leakage mode. The usefulness and practicality of such state-preserving voltage scaling schemes for embedded power-optimized memories is demonstrated in Qin and Rabaey [33]. The focus in Heo et al. [34] is on reducing bitline leakage power using leakage-biased bitlines. The technique turns off precharging transistors of unused subbanks to reduce bitline leakage, and actual bitline precharging is delayed until the subbank is accessed. All these techniques for cache power optimization can also be applied to reduce the leakage energy in the MPSoC.

2.3.3 Influence of Cache Architecture on Energy Consumption

In addition to applying these techniques for active and standby energy reduction, the choice of cache architecture plays an important role in the amount of utilized memory storage space and, consequently, the leakage energy. There are two popular alternatives for building a cache architecture for on-chip multiprocessors. The first one is to adopt a single multi-ported cache architecture that is shared by the multiple processors. There are two major advantages of this alternative: (1) constructive interference can reduce overall miss rates, and (2) inter-processor communication is easy to implement. However, a large multi-ported cache can consume significant energy. Additionally, this is not a scalable option. The second alternative is to allow each processor to have its own private cache [35–37]. The

Petitioner Samsung Ex-1010, 0047



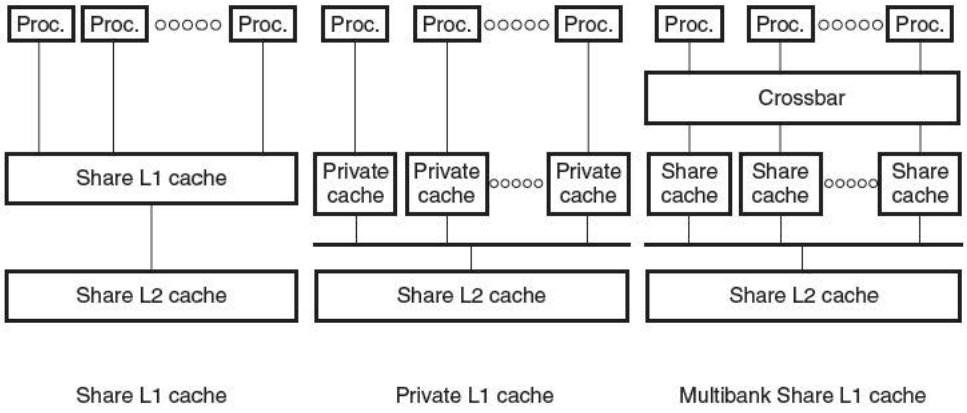
2-7 FIGURE Percentage sharing of instructions and data in an 8-processor MPSoC.

main benefits of the private cache option are low power per access, low access latency, and good scalability. Its main drawback is duplication of data and instructions in different caches. In addition, a complex cache coherence protocol is needed to maintain consistency.

In this section, we explore a cache configuration, referred to as the *crossbar-connected cache* (CCC) that tries to combine the advantages of the two options discussed above without their drawbacks. Specifically, the shared cache is divided into multiple banks using an $N \times M$ crossbar to connect the N processors and the M cache banks. In this way, the duplication problem is eliminated (as logically there is a single cache), a consistency mechanism is no longer needed, and the solution is scalable. These advantages are useful in reducing the energy consumption.

Figure 2-7 illustrates the percentage of sharing for instructions and data when SPLASH-2 benchmarks [38] are executed using eight processors each with private L1 caches. Percentage sharing is the percentage of data or instructions shared by at least two processors when considering the entire execution of the application. The first observation is that instruction sharing is extremely high. This is a direct result of the data parallelism exploited in these benchmarks, that is, different processors typically work on (mostly) different sets of data using the same set of instructions. Instruction sharing is above 90% for all applications except *raytrace*. These results clearly imply a large amount of instruction duplication across

Petitioner Samsung Ex-1010, 0048



2-8 FIGURE Different cache organizations: multiported shared cache, private caches, and CCC.

private caches. Consequently, a strategy that eliminates this duplication can bring large benefits (as there will be more available space), i.e., the effective cache capacity is increased by eliminating instruction duplication. Eliminating duplications also eliminated the extra leakage energy expended in powering duplicated cache lines. When one looks at the data sharing in Figure 2-7, the picture changes. The percentage sharing for data is low, averaging around 11%. However, employing CCC can still be desirable from the cache coherence viewpoint (since complex coherence protocols associated with private caches can be a major energy consumer). Based on these observations, using CCC might be highly beneficial in practice.

Figure 2-8 illustrates the three different cache organizations evaluated here: a multi-ported shared cache, private caches, and CCC. A common characteristic of these configurations is that they all have a shared L2 cache. For performance reasons, a CCC architecture would need a crossbar interconnect between the processors and L1 cache banks. Compared with a bus-based architecture, the crossbar-based architecture provides simultaneous accesses to multiple cache banks (which, in turn, helps to improve performance). However, due to the presence of the crossbar, the pipeline of the CCC processors needs to be restructured. For example, if the pipeline stages of the private cache (and multi-ported cache) architecture are instruction fetch (IF), instruction decode (ID), execution (EX), and write-back (WB), the pipeline stages in the CCC system are instruction crossbar transfer (IX), IF, ID, data crossbar transfer (DX), EX, and WB. In the IX stage, the processor transfers its request to an instruction cache bank through the

Petitioner Samsung Ex-1010, 0049

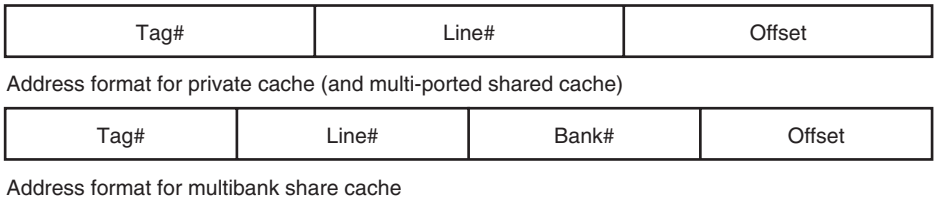


FIGURE 2-9 Address formats used by the different cache organizations.

crossbar. If more than one request targets the same bank, only one request can continue with the next pipeline stage (IF), and all other processors experience a pipeline stall. In the DX stage, if the instruction is load or store, another transfer request through the crossbar is issued.

Figure 2-9 shows the address formats used by the private cache system (and also the shared multiported cache system) and the CCC architecture. In the latter, a new field, called the bank number, is added. This field is used to identify the bank to be accessed. Typically, continuous cache blocks are distributed across different banks in order to reduce bank conflicts. Although CCC is expected to be preferable over the other two architectures, it can cause processor stalls when concurrent accesses to the same bank occur. To alleviate this, there are two possible optimizations. The first optimization is to use more cache banks than the number of processors. For instance, for a target MPSoC with four processors, utilize 4, 8, 16, or 32 banks. The rationale behind this is to reduce conflicts for a given bank. The second optimization deals with the references to the same block. When two such references occur, instead of stalling one of the processors, just read the block once and forward it to both the processors. This optimization should be more successful when there is a high degree of inter-processor block sharing.

To compare the cache energy consumption of the three architectures depicted in Figure 2-8, we ran a set of simulations assuming 70-nm technology (in which standby energy is expected to be comparable to active energy). We used the cache leakage control strategy proposed in Flautner et al. [31] in both the L1 and L2 caches whereby a cache line that has not been used for some time is placed into a leakage control mode. The experimental results obtained through cycle-accurate simulations and the SPLASH-2 benchmark suite indicate that the energy benefits of CCC range from 9% to 26% with respect to the private cache option. These savings come at the expense of a small degradation in performance. These results show that the choice of cache architecture is very critical in designing energy-aware MPSoCs.

Petitioner Samsung Ex-1010, 0050

2.3.4 Reducing Snoop Energy

In bus-based symmetric multi-processors (SMPs), all bus-side cache controllers snoop the bus (*bus snooping*) to ensure data coherence. Specifically, snoops occur when writes from the bus-side of the cache are issued to already cached blocks or on cache misses that require a cache fill. Thus, another class of memory system energy optimizations is that associated with snoops.

Since the cache snoop activity primarily involves tag comparisons and mostly incur tag misses, tag and data array accesses are separated, unlike normal caches in which both are accessed simultaneously to improve performance. Thus, energy optimizations include the use of dedicated tag arrays for snoops and the serialization of tag and data array accesses. For example, servers based on both Intel Xeon 2 and Alpha 21164 fetch tag and data serially.

An optimization that goes beyond reducing the data array energy of these tag misses is presented in Moshovos et al. [39]. In this work, a small, cache-like structure, called JETTY, is introduced between the bus and the backside of each processor's cache to filter the vast majority of snoops that would not find a locally cached copy. Energy is reduced, as accesses are mostly confined to JETTY, and thus accesses to the energy consuming cache tag arrays are decreased. Their work demonstrates that JETTY filters 74% (average) of all snoop-induced tag accesses that would miss, resulting in an average energy reduction of 29% in the cache.

Another approach to reducing the snoop energy is proposed in Saldhana and Lipasti [40]. In this work, the authors suggest a serial snooping of the different caches as opposed to a parallel snoop of all the caches attached to the bus. Whenever a request is generated on the bus, the snooping begins with the node closest to the requester and then propagates to the next successive node in the path only if the previous nodes have not satisfied the request. The tradeoff in this approach is the increased access latency due to the sequential snooping in order to reduce energy.

In Ekman et al. [41], an evaluation of JETTY and the serial snooping schemes is performed in the context of on-chip multiprocessors. The authors concluded that serial snooping is not very effective, as all the caches need to be searched when none of the caches have the requested data.

This performance penalty also translates to an energy penalty. Furthermore, they also found the JETTY approach not as attractive in the case of MPSoCs because the private cache associated with each processor in a MPSoC is comparable in size to the JETTY structure.

Another important impact of the snooping activity is the traffic generated on the interconnects. The memory accesses and organization have a significant

Petitioner Samsung Ex-1010, 0051

impact on the interconnect activity and power consumption. The next section explores on-chip communication energy issues.

2.4 ENERGY-AWARE ON-CHIP COMMUNICATION SYSTEM DESIGN

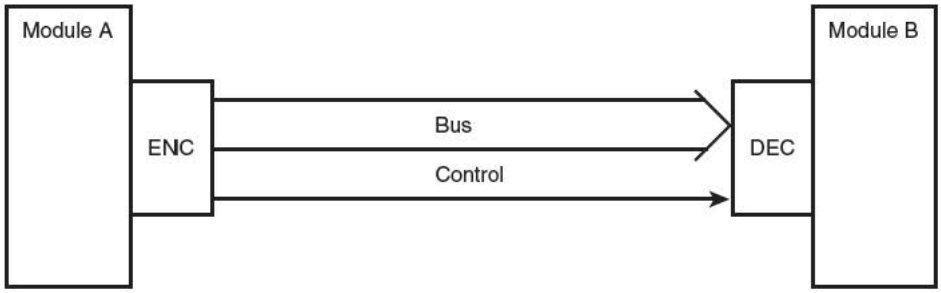
Computation and storage energy directly benefit from device scaling (smaller gates, smaller memory cells), but unfortunately the energy for global communication does not scale down. Projections assuming aggressive physical and circuit level optimizations for global wires [42] show that global communication on chip will require increasingly higher energy, the majority of it being active energy. Hence, communication-related energy is a significant concern in future MPSoC systems that will create many new challenges that have not been addressed in traditional high-performance on-chip interconnect design.

In this section we will explore various communication energy minimization approaches, moving from simple architectures (shared busses and point-to-point links), to complex advanced interconnects (networks on-chip [NoCs]) that are applicable to MPSoCs. Also, refer to Chapter 3 in this volume for additional material related to this section. The simplest on-chip communication channel is a bundle of wires, often called a *bus*. To be more precise, however, a bus can be significantly more complex than a bundle of wires. On-chip busses are highly shared communication infrastructures, with complex logic blocks for arbitration and interfacing. In an effort to maintain consistency with the surveyed literature, in the following we will use the term "bus" just as a synonym for a set of bundled wires. It is, however, important to remember that implementing encoding schemes in real-life designs requires maintaining compliance with complex SoC bus protocols. This is not an easy task, as outlined in Osborne et al. [43].

2.4.1 Bus Encoding for Low Power

Bus data communication has traditionally adopted straightforward encodings and modulation. The transfer function of the encoder and modulator of binary non-return-to-zero input streams was implicitly assumed to be unitary. In other words, no modulation and encoding were applied to the binary data before sending it on the bus. Low-energy communication techniques remove this implicit assumption, by introducing the concept of *signal coding for low power*.

Petitioner Samsung Ex-1010, 0052



2-10 FIGURE Bus encoding: one-directional communication.

The basic idea behind these approaches is to encode data sent through the communication channel to minimize its average switching activity, which is proportional to dynamic power consumption in CMOS technology. Ramprasad et al. [44] studied the data encoding for minimum switching activity problem and obtained upper and lower bounds on transition activity reduction for any encoding algorithm. This important theoretical result can be summarized as follows: *the savings obtainable by encoding depend on the entropy rate*[1] *of the data source and on the amount of redundancy in the code. The higher the entropy rate, the lower the energy savings that can be obtained by encoding with a given code redundancy.*

Even though the work by Ramprasad et al. [44] provides a theoretical framework for analyzing encoding algorithms, it does not provide general techniques for obtaining effective encoders and decoders. The complexity and energy cost of encoding and decoding circuits must be taken into account when evaluating any bus-encoding scheme.

Several authors have proposed low-transition activity encoding and decoding schemes. To illustrate the characteristics of these schemes in a simple setting, we consider a point-to-point, one-directional bus connecting two modules (e.g., a processor and its memory), as shown in Figure 2-10. Data from the source module are encoded, transmitted on the bus, and decoded at the destination. A practical instance of this configuration is the address bus for the processor/memory system. If the bus has very large parasitic capacitance, the energy dissipated in driving the

[1] In intuitive terms, the entropy of a data stream is related to the probability distribution of its symbols: the more evenly distributed, the higher the entropy. The entropy rate is the limit for $n \to \infty$ of the entropy of streams obtained by concatenating n symbols of the original data stream.

Petitioner Samsung Ex-1010, 0053

lines during signal transitions dominates the total energy cost of communication (including the cost of encoding and decoding). However, as discussed in the following, this assumption has to be carefully validated whenever a new encoding scheme is proposed.

Encoding for Random White Noise

A few encoding schemes have been studied starting from the assumption that the data sent on the bus can be modeled as random white noise (RWN), i.e., having a maximum entropy rate. Under this assumption, it is possible to focus solely on how to exploit redundancy to decrease switching activity, because all irredundant codes have the same switching activity (this result is proved in the paper by Ramprasad et al. [44]). Data are formatted in words of equal width, and a single word is transmitted every clock cycle.

In its simplest form, redundant encoding requires adding *redundant* wires to the bus. This can be seen as extending the word's width by one or more redundant bits. These bits inform the receiver about how the data were encoded before the transmission (see Fig. 2-10.) Low-energy encodings exploit the correlation between the word currently being sent and the previously transmitted one. The rationale is that energy consumption is related to the number of switching lines, i.e., to the Hamming distance between the words. Thus, transmitting identical words will consume no power, but alternating a word and its complement would produce the largest power dissipation, because all bus lines would be switching.

A conceptually simple and powerful scheme was proposed by Stan and Burleson [45] and called *bus invert* (BI) encoding. To reduce the switching, the transmitter computes the Hamming distance between the word to be sent and the previously transmitted one. If the distance is larger than half the word width, the word to be transmitted is inverted, i.e., complemented. An additional wire carries the *bus invert* information, which is used at the receiver end to restore the data.

This encoding scheme has some interesting properties. First, the worst-case number of transitions of an n-bit bus is $n/2$ at each time frame. Second, if we assume that data are uniformly randomly distributed, it is possible to show that the average number of transitions with this code is lower than that of any other encoding scheme with just one redundant line [45].

An unfortunate property of the 1-bit redundant *bus-invert code* is that the average number of transitions per line increases as the bus gets wider and asymptotically converges to 0.5, which is also the average switching per line of an unencoded bus. Moreover, the average number of transitions per line is already close to 0.5 for 32-bit busses. Thus, this encoding provides small energy saving for busses of typical width.

Petitioner Samsung Ex-1010, 0054

A solution to this problem is to partition the bus into fields and to use bus inversion in each field independently. If a word is partitioned in m fields, then m control lines are needed. Although this scheme can be much more energy efficient compared with 1-bit bus invert, m-bit bus invert is no longer the best among m-redundant codes. Nevertheless, it is conceptually simpler than other encoding schemes based on redundancy, and thus its implementation overhead (in terms of power) is small.

Extensions to the bus invert encoding approach include the use of limited-weight codes and transition signaling. A k-limited-weight code is a code having at most k 1's per word. This can be achieved by adding appropriate redundant lines [46]. Such codes are useful in conjunction with transition signaling, i.e., with schemes in which 1's are transmitted as a 0-1 (or 1-0) transition and 0's by the lack of a transition. Thus, a k-limited-weight code would guarantee at most k transitions per time frame (if we neglect the transitions on the redundant lines).

As a general note on bus invert and all redundant encoding schemes, although a redundant code can be implemented using additional bus lines, there are other options. In particular, it is possible to communicate with redundant codes without increasing the number of bus lines if the bus transmission rate is increased by $(m + n)/n$ with respect to the rate of the unencoded input stream, where n is the data word width and m is the number of redundant bits. This approach is often called *time redundancy* as opposed to *space redundancy* (i.e., adding extra lines). Obviously, many hybrid schemes can be envisioned.

Encoding for Correlated Data

Even though the RWM data model is useful for developing redundant codes with good worst-case behavior, in many practical cases data words have significant correlations in time and space. From an information theoretical viewpoint, the data source is not maximum entropy. This fact can be profitably exploited by advanced encoding schemes that outperform codes developed under the RWN model [46], even without adding redundant bus lines.

A typical example of a highly correlated data stream is the address stream in between a processor and its private memory. Addresses show a high degree of sequentiality. This is typical for instruction addresses (within basic blocks) and for data addresses (when data are organized in arrays). Therefore, in the limiting case of addressing a stream of data with consecutive addresses, Gray coding would be beneficial [47], because the Hamming distance between any pair of consecutive words is one, and thus the transitions on the address bus are minimized.

By using Gray encoding, instruction addresses need to be converted to Gray addresses before being sent on the bus. The conversion is necessary because offset

Petitioner Samsung Ex-1010, 0055

addition and arithmetic address manipulation is best done with standard binary encoding [48]. Moreover, address increments depend on the word width n. Since most processors are byte-addressable, consecutive words require an increment by $n/8$, e.g., by 4 [8] for 32-bit (64-bit) processors. Thus the actual encoding of interest is a partitioned code, whose most significant field is Gray encoded and whose least significant field has $(\log n/8)$ bits.

Musoll et al. [49] proposed a different partitioned code for addresses, which exploits the locality of reference, namely, most software programs favor *working zones* of their address space. The proposed approach partitions the address into an offset within a working zone and an identifier of the current working zone. In addition, a bit is used to denote a hit or a miss of the working zone. When there is a miss, the full address is transmitted through the bus. In the case of a hit, the bus is used to transmit the offset (using 1-hot encoding, and transition signaling), and additional lines are used to send the identifier of the working zone (using binary encoding). As an improvement over working zone, an irredundant code with similar properties, called *sector-based encoding* has been proposed by Aghaghiri et al. [50].

The T 0 code [51] uses one redundant line to denote when an address is consecutive to the previously transmitted one. In this case, the transmitter does not need to transmit the address and freezes the information on the bus, thus avoiding any switching. The receiver updates the previous address. When the address to be sent is not consecutive, then it is transmitted *tout court*, and the redundant line is de-asserted to inform the receiver to accept the address as is. When one is transmitting a sequence of consecutive addresses, this encoding requires no transition on the bus, compared with the single transition (per transmitted word) of the Gray code. In this context, an irredundant code with zero transition activity for address streams was also demonstrated, called INC-XOR [44], and later improved on by Aghaghiri et al. [52].

The highly sequential nature of addresses is just one simple example of spatio-temporal correlation on address busses. For instance, DRAM address busses use a time multiplexed addressing protocol, whose transition activity can be reduced by a tailored encoding scheme, as outlined in Cheng and Pedram [53]. Furthermore, several encoding schemes have been proposed for dealing with more general correlations [44,54–56] than those found in address busses. Unfortunately, many of these approaches assume that data stream statistics can be collected and analyzed at design time, making them far less useful for MPSoCs.

Practical Guidelines

The literature on low-power encoding has flourished in the last decade, and choosing the best encoding scheme is today a challenging task. A few practical

Petitioner Samsung Ex-1010, 0056

guidelines can be of help. First, in estimating the power savings obtained on the bus, the capacitive load of bus lines should be estimated with great care. In fact, in sub-micron technologies, coupling capacitance between adjacent wires dominates with respect to substrate capacitances; hence one must account for the impact of multiple transitions on adjacent lines. In this area, several coupling-aware low-power encoding schemes have recently been proposed [57,58].

Second, bus load capacitance estimates should be used as a selection criteria for encoding schemes. For typical on-chip bus capacitances, many encoding schemes are not practical, because encoders and decoders are too big and power-hungry. A good rule of thumb is to convert the expected switched capacitance reduction on the target bus $SW_{red} = C_{line}W_{bus}\alpha_{red}$ (i.e., the product between bus capacitance, number of bus lines, and bus switching activity reduction) into an equivalent number of inverter loads switching $Nb = SW_{red}/C_{in,INV}$ (i.e., how many inverter input switches are needed to switch the same amount of capacitance that we save with encoding). If the complexity of encoder and decoder, expressed in terms of inverter-equivalents, is similar to N_{eq}, then it is very likely that most of the transition activity savings are swamped by encoder and decoder power consumption. Also, it is very important to account for the speed and area penalty of the pure error-detecting circuit (codec). In general, most of the encoding schemes outlined above are well suited for off-chip busses with very large capacitances (e.g., 10 pF or more). Only the simplest schemes are suitable for typical on-chip busses (e.g., 1 pF or less).

2.4.2 Low Swing Signaling

An effective approach to high-speed energy-efficient communication is low swing signaling [59]. Even though it requires the design of receivers with good adaptation to line impedance and high sensitivity (often achieved by means of simplified sense-amplifiers), power savings on the order of 10× have been estimated with reduced interconnect swings of a few hundreds of mV in a 0.18-μm process [60]. The use of low swing signaling poses a critical challenge for design: communication reliability has to be provided in spite of the decreased noise margins and the strong technology limitations, under limited power budgets.

With present technologies, most chips are designed under the assumption that electrical waveforms can always carry correct information on chip. As technology scales, communication is likely to become inherently unreliable because of the increased sensitivity of interconnects to on-chip noise sources, such as crosstalk and power-supply noise. As a consequence, solutions for combined energy minimization and communication reliability control have to be developed for

Petitioner Samsung Ex-1010, 0057

NoCs. Redundant bus encoding provides a degree of freedom for spanning the energy-reliability tradeoff [61,62]. The key point of this approach is to model on-chip interconnects as noisy channels and to exploit the error detection capability coding schemes, which would provide a link transfer reliability in excess with respect to the constraint, to decrease the voltage swing, resulting in an overall energy saving (compared with the unencoded link) in spite of the overhead associated with the code implementation. The energy efficiency of a code is tightly related to its error recovery technique, namely, error correction or retransmission of corrupted data. This issue resembles the tradeoff investigation between forward error correction (FEC) and automatic repeat request (ARQ), well known to network engineers, but for on-chip communication networks this study is still in its early stage.

Bertozzi et al. [61] have explored the energy-reliability tradeoff for on-chip low-swing busses. Starting from a standard on-chip bus (AMBA), various reliability-enhancement encoding techniques are explored. Hamming codes are explored in both their error correction and error detection embodiments. Other linear codes, namely, Cyclic Redundancy Check (CRC) codes, are considered as well. For error-detecting codes, an AMBA-compliant retransmission mechanism is also proposed. Experimental results in Bertozzi et al. [61] point out that the detection capability of a code plays a major role in determining its energy efficiency, because the higher the error detection capability the lower we can drive the voltage swing, knowing that the increased error rate caused by the lowered signal-to-noise ratio would not lead to system failures as long as the errors are detected. As far as error recovery is concerned, error correction is beneficial in terms of recovery delay but has two main drawbacks: it limits the detection capability of a code, and it makes use of high-complexity decoders. On the contrary, when the higher recovery delay (associated with the retransmission time) of retransmission mechanisms can be tolerated, they provide a higher energy efficiency, thanks to the lower swings and simpler codecs they can use while preserving communication reliability.

In many practical cases, communication noise depends on operating conditions and environmental factors. It is then possible to devise adaptive schemes that dynamically adjust signal levels (i.e., the voltage swing on bus lines) depending on noise levels. In the approach proposed by Worm et al. [62], the frequency of detected errors by an error-detecting code is monitored at run time. When the frequency rises too much, the voltage swing is increased, whereas it is decreased if error frequency becomes very small. This closed-loop control of voltage provides much increased robustness and flexibility, even though the complexity of the encoding and decoding circuitry is increased, and variable voltage swing circuits are also required.

Petitioner Samsung Ex-1010, 0058

2.4.3 Energy Considerations in Advanced Interconnects

Network architecture heavily influences communication energy. As hinted in the previous section, shared-medium networks (busses) are currently the most common choice, but it is intuitively clear that busses are not energy-efficient as network size scales up [63], as it must for MPSoCs. In bus-based communication, data are always broadcast from one transmitter to all possible receivers, whereas in most cases messages are destined to reach only one receiver, or a small group. Bus contention, with the related arbitration overhead, further contributes to the energy overhead.

Studies on energy-efficient on-chip communication indicate that hierarchical and heterogeneous architectures are much more energy-efficient than busses [64,65]. Zhang et al. [65] developed a *hierarchical generalized mesh* whereby network nodes with a high communication bandwidth requirement are clustered and connected through a programmable generalized mesh consisting of several short communication channels joined by programmable switches. Clusters are then connected through a generalized mesh of global long communication channels. Clearly such architecture is heterogeneous because the energy cost of intra-cluster communication is much smaller than that of inter-cluster communication. Although the work of Zhang et al. [65] demonstrates that power can be saved by optimizing network architecture, many network design issues are still open, and we need tools and algorithms to explore the design space and to tailor network architecture to specific applications or classes of applications.

Network architecture is only one facet of network layer design, the other major facet being network control. A critical issue in this area is the choice of a switching scheme for indirect network architectures. From the energy viewpoint, the tradeoff is between the cost of setting up a circuit-switched connection once and for all, and the overhead of switching packets throughout the entire communication time on a packet-based connection. In the former case the network control overhead is "lumped" and incurred once, whereas in the latter case, it is distributed over many small contributions, one for each packet. When communication flow between network nodes is extremely persistent and stationary, circuit-switched schemes are likely to be preferable, whereas packet switched schemes should be more energy-efficient for irregular and non-stationary communication patterns. Needless to say, circuit switching and packet switching are just two extremes of a spectrum, with many hybrid solutions in between.

Packetization and Energy Efficiency

The current trend in on-chip network design is toward packet-switched architectures [65–66]. This choice is mostly driven by the need for utilizing limited global

Petitioner Samsung Ex-1010, 0059

wiring resources as efficiently as possible. (It is a well-known fact that circuit-switched networks have a low utilization with respect to packet-switched networks.) Ye et al. [67] have studied the impact of packet size on energy and performance for a regular on-chip network (a mesh) connecting homogeneous processing elements, in a cache-coherent MPSoC. Packetized data flow on the MPSoC network is affected by: (1) the number of packets in the network, (2) the energy consumed by each packet on one hop, and (3) the number of hops each packet travels. Different packetization schemes will have different impact on these factors and, consequently, affect the network power consumption. Several findings are reported in Ye et al. [67] and are summarized below.

Larger packet size will increase the energy consumed per packet, because there are more bits in the payload. Furthermore, larger packets will occupy the intermediate node switches for a longer time and cause other packets to be re-routed to non-shortest datapaths. This leads to more contention that will increase the total number of hops needed for packets traveling from source to destination. As a result, as packet size increases, energy consumption on the interconnect network will increase.

Although an increase in packet size will increase the energy dissipated by the network, it will decrease the energy consumption in cache and memory. Because larger packet sizes will decrease the cache miss rate, both cache energy consumption and memory energy consumption will be reduced. The total energy dissipated by the MPSoC comes from non-cache instructions (instructions that do not involve cache access) executed by each processor, the caches, and the shared memories, as well as the interconnect network. In order to assess the impact of packet size on the total system energy consumption, all MPSoC energy contributors should be considered together.

Ye et al. [67] showed that the overall MPSoC energy initially decreases as packet size increases. However, when the packets are too large, the total MPSoC energy starts increasing again. This is because the interconnect network energy outgrows the decrease of energy on cache, memory, and non-cache instructions. As a result, there is an optimal packet size, which, for the architecture analyzed by Ye et al., is around 128 bytes. It is important to stress, however, that different processor architectures and system interconnects may have different optimal packet sizes.

Energy and Reliability in Packet-Switched NoCs

As MPSoC communication architectures evolve from shared busses to packet-based micro-networks, packetization issues have to be considered when assessing the energy efficiency of error recovery techniques. In general, packets are often broken into message flow control units or *flits*. In the presence of channel width constraints,

Petitioner Samsung Ex-1010, 0060

multiple physical channel cycles may be used to transfer a single flit. A *phit* is the unit of information that can be transferred across a physical channel in a single cycle. Flits represent logical units of information, as opposed to phits, which correspond to physical quantities, i.e., the number of bits that can be transferred in a single cycle. In many implementations, a flit is set to be equal to a phit.

Communication reliability can be guaranteed at different levels of granularity. We might refer control bits (i.e., a checksum) to an entire packet, thus minimizing control bits overhead, although this would prevent us from stopping the propagation of corrupted flits, as routing decisions would be taken in advance with respect to data integrity checking. In fact, control bits would be transmitted as the last flit of the packet. In this scenario, the cost for redundancy would be paid in the time domain. The alternative solution is to provide reliability at the flit level, thus refining control granularity but paying for redundancy in the space domain (additional wiring resources for check bits). The considerations that follow will refer to this latter scenario, wherein two different solutions are viable:

- The error recovery strategy can be *distributed* over the network. Each communication switch is equipped with error detecting/correcting circuitry, so that error propagation can be immediately stopped. This is the only way to avoid routing errors: should the header get corrupted, its correct bit configuration could be immediately restored, preventing the packet from being forwarded across the wrong path to the wrong destination. Unfortunately, retransmission-oriented schemes need power-hungry buffering resources at each switch, so their advantage in terms of higher detection capability has to be paid for with circuit complexity and power dissipation.
- Alternatively, an *end-to-end* approach to error recovery is feasible: only end-nodes are able to perform error detection/correction. In this case, retransmission may not be convenient at all, especially when source and destination are far apart from each other, and retransmitting corrupted packets would stimulate a large number of transitions, beyond giving rise to large delays. For this scenario, error correction is the most efficient solution, even though the proper course of action has to be taken to handle incorrectly routed packets (retransmission time-outs at the source node, deadlock avoidance, and so on).

Another consideration regards the way retransmissions are carried out in an NoC. Traditional shared bus architectures can be modified to perform retransmissions in a "stop and wait" fashion: the master drives the data bus and waits for the slave to carry out sampling on one of the following clock edges. If the slave detects corrupted data, a feedback has to be given to the master, scheduling a retransmission. To this purpose, an additional feedback line can be used, or built-in mechanisms of the bus protocol can be exploited.

Petitioner Samsung Ex-1010, 0061

In packet-switched networks, data packets transmitted by the master can be seen as a continuous flow, so the retransmission mechanism must be either "go-back-N" or "selective repeat." In both cases, each packet has to be acknowledged (ACK), and the difference lies in the receiver (switch or network interface) complexity. In a "go-back-N" scheme, the receiver sends a not ACK (NACK) to the sender relative to a certain incorrectly received packet. The sender reacts by retransmitting the corrupted packet as well as all other following packets in the data flow. This alleviates the receiver from the burden to store packets received out of order and to reconstruct the original sequence. On the other hand, when this capability is available at the receiver side (at the cost of further complexity), retransmissions can be carried out by selectively requiring the corrupted packet without the need to retransmit also successive packets. The tradeoff here is between switch and network interface complexity and number of transitions on the link lines.

2.5 ENERGY-AWARE SOFTWARE

Although employing energy-efficient circuit/architecture techniques is a must for energy-aware MPSoCs, there are also important roles for the software to play. Perhaps one of the most important of these is to parallelize a given application across on-chip processors. It is to be noted that, in this context, it may not be acceptable to use a conventional application parallelization strategy from the high-end computing domain since such strategies focus only on performance and do not account for the energy behavior of the optimized code. More specifically, such high-end computing-oriented techniques can be very extravagant in their use of processors (or machine resources in general), i.e., they tend to use a large number of parallel processors in executing applications even though most of these processors bring only marginal performance benefits.

However, a compiler designed for energy-aware MPSoCs may not have this luxury. In other words, it should strive to strike an acceptable balance between increased parallelism (reduced execution cycles) and increased energy consumption as the number of processors is increased. Therefore, our belief is that it is very important to determine the ideal number of processors to use in executing a given MPSoC application. In addition, it is conceivable to assume that different parts of the same application can demand different numbers of processors to generate the best behavior at run-time. Consequently, it may not be sufficient to fix the number of processors to use at the same value throughout the execution. During execution of a piece of code, unused processors can be placed in a low power state for saving energy [68].

Petitioner Samsung Ex-1010, 0062

Most published work on parallelism for high-end machines [69] is static, that is, the number of processors that execute the code is fixed for the entire execution. For example, if the number of processors that execute a code is fixed at eight, all parts of the code (for example, all loop nests) are executed using the same eight processors. However, this may not be a very good strategy from the viewpoint of energy consumption. Instead, one can consume much less energy by using the minimum number of processors for each loop nest (and shutting down the unused processors). In adaptive parallelization [70], the number of processors is tailored to the specific needs of each code section (e.g., a nested loop in array-intensive applications). In other words, the number of processors that are active at a given period of time changes dynamically as the program executes. For instance, an adaptive parallelization strategy can use four, six, two, and eight processors to execute the first four nests in a given code. There are two important issues that need to be addressed for energy-aware parallelization (assuming that interprocessor communication occurs through shared memory):

- *Determining the ideal number of processors to use*. There are at least two ways to determine the number of processors needed per nest. The first option is to adopt a fully dynamic strategy whereby the number of processors (for each nest) is decided on in the course of execution (at run-time). Kandemir et al. [71] present such a dynamic strategy. Although this approach is expected to generate better results once the number of processors has been decided (as it can take run-time code behavior and dynamic resource constraints into account), it may also incur some performance overhead during the process of determining the number of processors. This overhead can, in some cases, offset the potential benefits of adaptive parallelism. In the second option, the number of processors for each nest is decided statically at compile-time. This approach has a compile-time overhead, but it does not lead to any run-time penalty. It should be emphasized, however, that although this approach determines the number of processors statically at compile-time, the activation/deactivation of processors and their caches occurs dynamically at run-time. Examples of this strategy can be found in Kadayif et al. [68,72].
- *Selection criterion for the number of processors.* An optimizing compiler can target different objective functions such as minimizing the execution time of the compiled code, reducing executable size, improving power or energy behavior of the generated code, and so on. In addition, it is also possible to adopt complex objective functions and compilation criteria such as minimizing the sum of the cache and main memory energy consumptions while keeping the total execution cycles bounded by M. For example, the framework

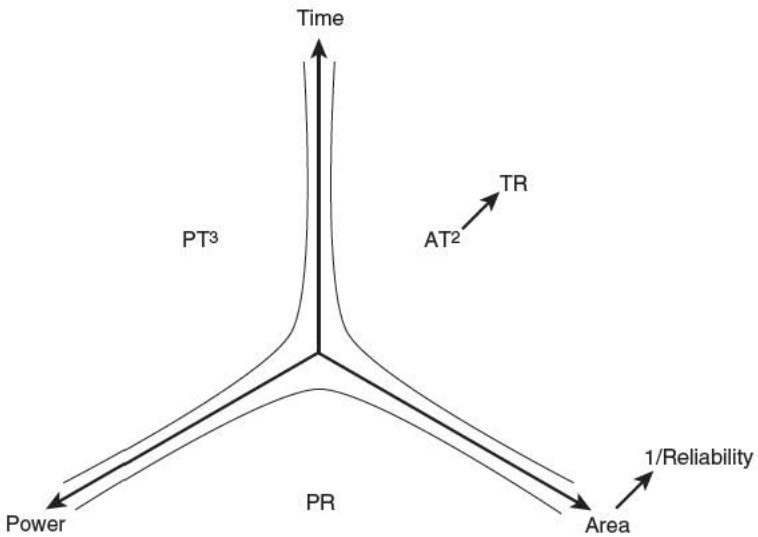
Petitioner Samsung Ex-1010, 0063

described in Kandemir et al. [71] accepts as objective function and constraints any linear combination of energy consumptions and execution cycles of individual hardware components.

2.6 CONCLUSIONS

This chapter has surveyed a number of energy-aware design techniques for controlling both active and standby power consumption that are applicable to MPSoCs. In particular, it covered techniques used to design energy-aware MPSoC processor cores, the on-chip memory hierarchy, the on-chip communication system, and MPSoC energy-aware software techniques. The vast majority of the techniques are derived from existing uniprocessor energy-aware systems that take on new dimensions in the MPSoC space. As energy-aware MPSoC design is a relatively new area, many open questions remain.

Other issues to be considered but not discussed in this chapter are the possible promise of globally asynchronous, locally synchronous (GALS) clocking for MPSoCs, as well as the inextricable link between energy-aware design techniques and their impact on system reliability. As shown in Figure 2-11, design tradeoffs



2-11 FIGURE Evolving design tradeoffs. T, time; R, reliability; A, area; P, power; PT, product of power and time.

Petitioner Samsung Ex-1010, 0064

have changed over the years from a focus on area to a focus on performance and area to a focus on performance and power. In the future, the metrics of performance, power, and reliability and their tradeoffs will be of increasing importance.

ACKNOWLEDGMENTS

The authors have been supported in part by NSF Career Award Grants CCR-0093082 and CCR-0093085, NSF grant ITR-0082064, and a MARCO 98-DF-600 grant from GSRC. The authors would also like to acknowledge the contributions of various graduate students from their labs who worked on several projects whose results are abstracted here.

Petitioner Samsung Ex-1010, 0065