



MULTIPROCESSOR SYSTEMS-ON-CHIPS

AHMED
AMINE
JERRAYA
WAYNE
WOLF

SYSTEMS
ON
SILICON



MK[®]
MORGAN KAUFMANN

Multiprocessor Systems-on-Chips

The Morgan Kaufmann Series in Systems on Silicon

Series Editors: Peter Ashenden, Ashenden Designs Pty. Ltd. and Adelaide University, and Wayne Wolf, Princeton University

The rapid growth of silicon technology and the demands of applications are increasingly forcing electronics designers to take a systems-oriented approach to design. This has led to new challenges in design methodology, design automation, manufacture and test. The main challenges are to enhance designer productivity and to achieve correctness on the first pass. *The Morgan Kaufmann Series in Systems on Silicon* presents high quality, peer-reviewed books authored by leading experts in the field who are uniquely qualified to address these issues.

The Designer's Guide to VHDL, Second Edition

Peter J. Ashenden

The System Designer's Guide to VHDL-AMS

Peter J. Ashenden, Gregory D. Peterson, and Darrell A. Teegarden

Readings in Hardware/Software Co-Design

Edited by Giovanni De Micheli, Rolf Ernst, and Wayne Wolf

Modeling Embedded Systems and SoCs

Axel Jantsch

ASIC and FPGA Verification: A Guide to Component Modeling

Richard Munden

Multiprocessor Systems-on-Chips

Edited by Ahmed Amine Jerraya and Wayne Wolf

Forthcoming Titles

Functional Verification

Bruce Wile, John Goss, and Wolfgang Roesner

Rosetta User's Guide: Model-Based Systems Design

Perry Alexander, Peter J. Ashenden, and David L. Barton

Rosetta Developer's Guide: Semantics for Systems Design

Perry Alexander, Peter J. Ashenden, and David L. Barton

Multiprocessor Systems-on-Chips

Edited by

Ahmed Amine Jerraya

TIMA Laboratory

Wayne Wolf

Princeton University



ELSEVIER

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

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

Dedication

For Houda, Lydia and Nelly—Ahmed Amine Jerraya

For Nancy and Alec—Wayne Wolf

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.

Contents

About the Editors	vi
Preface	xix

1	The What, Why, and How of MPSoCs	1
	<i>Ahmed Amine Jerraya and Wayne Wolf</i>	
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

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

- 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

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

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

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

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

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

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

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

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

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

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

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

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.

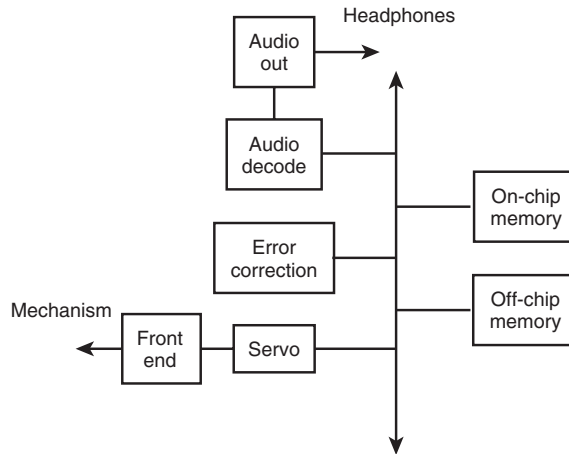
Exactly what components are assembled on the SoC varies with the application. Many SoCs contain analog and mixed-signal circuitry for input/output (I/O). Although some high-performance I/O applications require a separate analog interface chip that serves as a companion to a digital SoC, most of an SoC 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



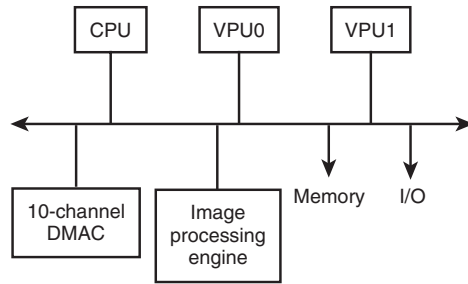
1-1

Architecture of a CD/MP3 player.

FIGURE

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



1-2

Architecture of the Sony Playstation 2 Emotion Engine.

FIGURE

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 300MHz, 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

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

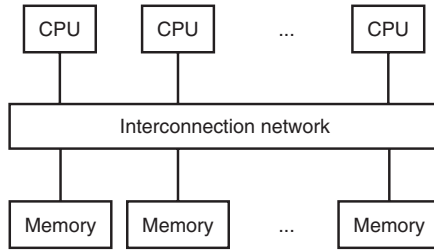
The fact that an MPSoC is a multiprocessor means that software design is an inherent part of the overall chip design. This is a big change for chip designers, who are used to coming up with hardware solutions to chip design problems. In an MPSoC, either hardware or software can be used to solve a problem; 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



1-3

A generic shared-memory multiprocessor.

FIGURE

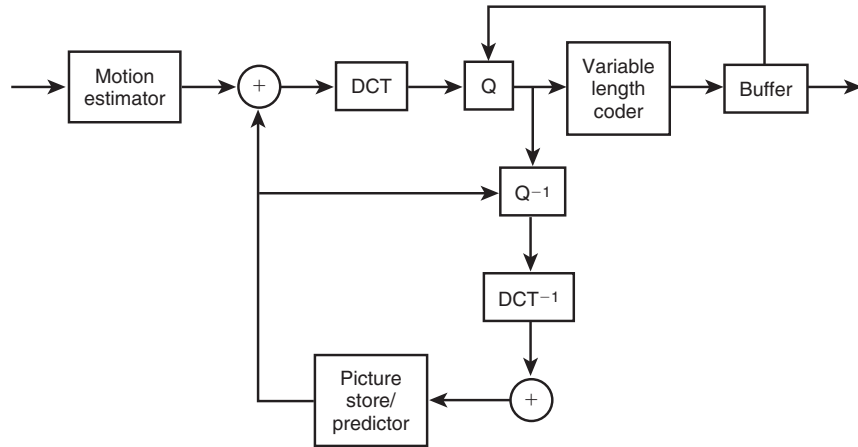
may also be heterogeneous. MPSoCs often require large amounts of memory. The device may have embedded memory on-chip as well as relying on off-chip commodity memory.

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



1-4

Block diagram of MPEG-2 encoding.

FIGURE

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.

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

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

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.

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 important 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.

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

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 (@ 150MHz) and a C55x DSP core (@ 200MHz). Both of them have a 16-Kb I-cache, an 8-Kb D-cache, and a two-way set

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.

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,

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

parallelism statically (clustered VLIW) and dynamically (simultaneous multi-threading [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.

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.