UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Dynamically configurable system-on-chip platforms : architectures and design methodologies

Permalink
https://escholarship.org/uc/item/828847b4

Author
Sekar, Krishna

Publication Date
2005

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Dynamically Configurable System-on-Chip Platforms: Architectures and Design Methodologies

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in
Electrical and Computer Engineering (Computer Engineering)

by

Krishna Sekar

Committee in charge:

Professor Sujit Dey, Chair
Professor Brad Calder
Professor Rajesh Gupta
Professor Bill Lin
Professor Kenneth Yun

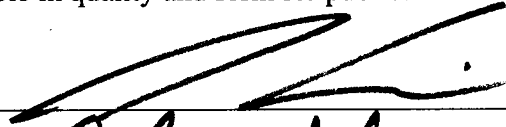
2005

Copyright

Krishna Sekar, 2005

All rights reserved.

The dissertation of Krishna Sekar is approved, and it is acceptable in quality and form for publication on microfilm:



Chair

University of California, San Diego

2005

ACKNOWLEDGMENTS

I would like to take this opportunity to first thank the two people who have been instrumental in shaping this work and guiding me through my doctoral program. I am thankful to Prof. Sujit Dey, my advisor, for being such a wonderful mentor and guide. I am grateful to him for giving me the opportunity to work under his excellent supervision, for providing me the flexibility to work on projects of my interest, for introducing me to the best people in the field, and for always being very approachable and supportive. Towards Kanishka Lahiri, I can only express my deepest respect, admiration and gratitude. I would like to thank him for being such an excellent collaborator, for his insightful ideas and comments, for his role in guiding this work to its final shape, and for being such a good friend. This work would have been far lesser without him.

I would also like to express my gratitude to Anand Raghunathan for providing valuable guidance, insights and feedback, especially for work presented in Chapters 3 and 5. I have been inspired by his excellent research methods, his attention to detail, and his knowledge and insight into the field. A number of other people helped in making this work possible. I would like to thank my thesis committee members, Professors Bill Lin, Kenneth Yun, Bradley Calder and Rajesh Gupta, for providing valuable comments on this work. Thanks are due to Amit Sinha *et al* from MIT for developing the JouleTrack tool, which I used extensively in my experiments. I would also like to thank the anonymous reviewers of my conference and journal papers for providing detailed comments, which were very valuable in improving the quality of this work.

Throughout my graduate studies, I have been fortunate to have the opportunity to work with an incredible group of people at the MESDAT Labs at UC San Diego. I would like to thank Xiaoliang Bai, Li Chen, Saumya Chandra, Dong-Gi Lee, Shoubhik Mukhopadhyay, Debashis Panigrahi, Naomi Ramos, Clark Taylor, Mayank Tiwari, Chong Zhao and Yi Zhao for all their help and encouragement over the years, and for making graduate life such an enriching and positive experience. I will always cherish the lively discussions we had during group meetings and coffee breaks at ROMA. Special thanks are due to Cathy MacHutchin for her efficient handling of all things administra-

tive, for making our work environment so nice and cheerful, and for being such a nice and wonderful person.

I also owe a debt of gratitude to all my friends and family for their encouragement, love and support over all these years. I would especially like to thank Abhishek, Dave, Mahim, Naomi, Ranjita and, above all, Debashis for always being there through all the ups and downs of graduate life.

Last, but foremost, I would like to express my deepest affection and gratitude towards my parents, Ashima and Sekar, and my brother, Kartick. I would like to thank my parents for giving me the chances in life they never had, for making numerous sacrifices so that I could have the best in life, and for their unconditional love and support. I wouldn't be where I am today without them, and I hope that I have made them proud.

The text of the following chapters, in part or in full, is based on material that has been published in conference proceedings or journals, or is pending publication in journals, or is in review. Chapter II is based on material published in the International Conference on VLSI Design, 2004, and material submitted to the IEEE Transactions on Computer-Aided Design of Circuits and Systems. Chapter III is based on material published in the Design Automation Conference, 2005, and material submitted to the IEEE Transactions on VLSI Systems. Chapter IV is based on material published in the International Conference on Computer-Aided Design, 2003, and material submitted to the IEEE Transactions on Computer-Aided Design of Circuits and Systems. Chapter V is based on material that has been accepted for publication in the Design Automation and Test in Europe Conference, 2006. I was the primary researcher and author of each of the above publications, and the coauthors listed in these publications collaborated on, or supervised the research which forms the basis for these chapters.

VITA

1999	B.Tech., Computer Science and Engineering, Indian Institute of Technology, Kharagpur
1999–2005	Research Assistant, Dept. of Electrical and Computer Engineering, University of California, San Diego
2000–2003	Teaching Assistant, Dept. of Electrical and Computer Engineering, University of California, San Diego
2001	M.S., Electrical and Computer Engineering, University of California, San Diego
2000	Summer Research Intern, Fujitsu Laboratories of America, Sunnyvale, California
2001	Summer Intern, Mentor Graphics Corporation, Wilsonville, Oregon
2004	Summer Research Assistant, NEC Laboratories America, Princeton, New Jersey
2005	Ph.D., Electrical and Computer Engineering, University of California, San Diego

PUBLICATIONS

K. Sekar, K. Lahiri, A. Raghunathan and S. Dey, "Dynamically configurable bus topologies for high-performance on-chip communication", *IEEE Transactions on VLSI Systems*. (in review).

K. Sekar, K. Lahiri and S. Dey, "High-performance and energy-efficient platform-based system-on-chip design with dynamic platform management", *IEEE Transactions on Computer-Aided Design*. (in review).

K. Sekar and S. Dey, "LI-BIST: a low-cost self-test scheme for SoC logic cores and interconnects", *Journal of Electronic Testing: Theory and Applications*, vol. 19, no. 2, pp. 113–123, April 2003.

S. Dey, P. Sanchez, D. Panigrahi, L. Chen, C. Taylor and K. Sekar, "Using a soft core in a SoC design: experiences with picoJava", *IEEE Design and Test of Computers*, vol. 17, no. 3, pp. 60–71, July-September 2000.

K. Sekar, K. Lahiri, A. Raghunathan and S. Dey, "Integrated data relocation and bus reconfiguration for adaptive system-on-chip platforms", in *Proc. Design Automation and Test in Europe*, March 2006. (to appear).

K. Sekar, K. Lahiri, A. Raghunathan and S. Dey, "FLEXBUS: a high-performance system-on-chip communication architecture with a dynamically configurable topology", *Proc. Design Automation Conference*, pp. 571–574, Anaheim, June 2005.

K. Sekar, K. Lahiri and S. Dey, "Configurable platforms with dynamic platform management: an efficient alternative to application-specific system-on-chips", *Proc. International Conference on VLSI Design*, pp. 307–315, Mumbai, January 2004.

K. Sekar, K. Lahiri and S. Dey, "Dynamic platform management for configurable platform-based system-on-chips", *Proc. International Conference on Computer-Aided Design*, pp. 641–648, San Jose, November 2003.

K. Sekar and S. Dey, "LI-BIST: A low-cost self-test scheme for SoC logic cores and interconnects", *Proc. VLSI Test Symposium*, pp. 417–422, Monterey, May 2002.

I. Ghosh, K. Sekar and V. Boppana, "Design for verification at the register transfer level", *Proc. International Conference on VLSI Design/Asia South Pacific Design Automation Conference (ASP-DAC)*, pp. 420–425, Bangalore, January 2002.

P. Dasgupta, P. P. Chakrabarti, A. Nandi, K. Sekar and A. Chakrabarti, "Abstraction of word-level linear arithmetic functions from bit-level component descriptions, *Proc. Design, Automation and Test in Europe*, pp. 4–8, Munich, March 2001.

L. Chen, S. Dey, P. Sanchez, K. Sekar and Y.H. Chen, "Embedded hardware and software self-testing methodologies for processor cores", *Proc. Design Automation Conference*", pp. 625–630, Los Angeles, June 2000.

TABLE OF CONTENTS

Signature Page iii

Acknowledgments iv

Vita, Publications, and Fields of Study vii

Table of Contents ix

List of Figures xii

List of Tables xiv

Abstract xv

I Introduction 1
- A. Landscape of Design Styles 5
- B. General-Purpose Configurable Platforms 8
  - 1. Static Configurability 9
  - 2. Dynamic Configurability 9
- C. Thesis Overview and Contributions 11

II Dynamically Configurable Platforms with Dynamic Platform Management . 14
- A. Introduction 14
  - 1. Chapter Overview 15
- B. Dynamically Configurable Platform Components and Configuration Techniques 16
  - 1. Configurable Processors 16
  - 2. Configurable Caches 19
  - 3. Configurable Memory Sub-Systems 21
  - 4. Configurable On-Chip Communication Architectures 24
  - 5. On-Chip Configurable Fabrics 25
  - 6. Summary 26
- C. Dynamic Platform Management 27
- D. Conclusions 30

III Dynamically Configurable Bus Topologies for High-Performance On-Chip Communication 31
- A. Introduction 31
  - 1. Chapter Overview 32
  - 2. Related Work 33
- B. Background 34
- C. Motivation 36

1. Case Study: IEEE 802.11 MAC Processor 36
2. Statically Configured Topologies 37
3. Dynamic Topology Configuration 39
D. FLEXBUS Architecture 41
1. Overview 41
2. Coarse-Grained Topology Control: Bridge By-Pass Mechanism . . . 42
3. Fine-Grained Topology Control: Component Re-Mapping Mechanism 46
E. Dynamic Configuration Policies for FLEXBUS 49
F. Scalability of the FLEXBUS Approach 51
1. Dynamic Multi-Bridge By-Pass 51
2. Scalability of Component Re-Mapping 53
3. Applying FLEXBUS in Complex Communication Architectures . . . 53
G. Experimental Results 56
1. Experimental Methodology 57
2. Hardware Implementation Results 57
3. Performance Under Synthetic Traffic Profiles 59
4. Application to an IEEE 802.11 MAC Processor 62
5. Application to a UMTS Turbo Decoder Design 63
H. Conclusions 65

IV Dynamic Management of SoC Platforms with Configurable Processors and Dynamic Data Relocation 67
A. Introduction 67
1. Chapter Overview 69
2. Related Work 70
B. Configurable Platform Architecture 71
C. Demonstrating Platform Management for Security Processing 73
1. Case Study: UMTS and WLAN Security Processing 73
2. Platform Configuration Space 74
3. Security Processing: Static Configuration 76
4. Security Processing: Dynamic Platform Management 76
D. Dynamic Platform Management Methodology 79
1. Terminology 79
2. Off-line Task Characterization 81
3. Dynamic Platform Management Algorithms 82
E. Experimental Results 85
1. Application to StrongARM based platform 85
2. Application to Altera Excalibur SOPC 92
F. Conclusions 97

V Dynamic Management of SoC Platforms with Dynamic Data Relocation and Reconfigurable Bus Architectures 98
A. Introduction 98
1. Chapter Overview 99

2. Related Work 100
B. Configurable Platform Architecture 101
C. Motivational Example 102
1. Case Study: Integrated Viterbi-Turbo Decoder Design 102
2. Illustrative Examples 105
D. Dynamic Platform Management Methodology 108
1. Problem Description and Methodology Overview 108
2. Off-line Characterization Phase 109
3. Run-time Platform Configuration Phase 115
E. Experimental Results 117
1. Experimental Methodology 117
2. Application to the Viterbi-Turbo Decoder Design 117
3. Platform Configuration Overhead 118
4. Impact of Dynamic Platform Management on Performance 119
F. Conclusions 121

VI Application-Architecture Co-Adaptation 122
A. Introduction 122
1. Application-Architecture Co-Adaptation: Overview 123
2. Chapter Overview 125
3. Related Work 126
B. Motivation 127
1. Case Study: Wireless Image Delivery System 127
2. Co-Adaptation: Illustrative Examples 129
C. Co-Adaptation Methodology 133
1. Problem Definition 133
2. Methodology Overview 134
D. Application of Co-Adaptation to Wireless Image Delivery 136
1. Parameter Impact Analysis 136
2. Run-Time Co-Adaptation Policy 138
E. Experimental Results 141
1. Experimental Methodology 141
2. Impact on Total Latency and PSNR 142
3. Impact on Energy Consumption 145
F. Conclusions 146

VII Future Research Directions 147
1. Architectural-Level Adaptation 148
2. Cross-Layer Adaptation 149
3. Network-Wide Adaptation 150

Bibliography 151

LIST OF FIGURES

I.1 GeForce 7800 graphics processing unit from NVIDIA with 302 million transistors . . . 2
I.2 Increasing cost and design problems in developing custom SoCs/ASICs 3
I.3 Nomadik mobile multimedia platform from STMicroelectronics . . . 4
I.4 Landscape of design styles and configurable platforms . . . 6

II.1 Dynamically configurable platforms with dynamic platform management . . . 27
II.2 Software architecture for dynamic platform management . . . 28

III.1 Example bus-based communication architecture with two bus segments 35
III.2 IEEE 802.11 MAC processor: (a) functional specification, (b) mapping to a single shared bus, (c) mapping to a multiple bus architecture 37
III.3 Execution of the IEEE 802.11 MAC processor under (a) single shared bus, (b) multiple-bus, (c) FLEXBUS . . . 39
III.4 Dynamic bridge by-pass capability in FLEXBUS . . . 43
III.5 Dynamic component re-mapping capability in FLEXBUS . . . 47
III.6 Example system with four bus segments, and possible assignment of arbiters as virtual masters . . . 52
III.7 Hardware implementation results for example eight master and eight slave system . . . 58
III.8 Performance comparison under varying volume of cross-bridge traffic 59
III.9 Two state Markov model to systematically generate varying traffic profiles . . . 60
III.10 Performance of FLEXBUS with run-time configuration policy . . . 61
III.11 UMTS Turbo encoder and decoder: (a) encoder functional blocks, (b) decoder functional blocks, (c) mapping to a single shared bus architecture, (d) mapping to a multiple bus architecture . . . 64

IV.1 Energy versus speed trade-off [1] . . . 68
IV.2 A general-purpose configurable platform architecture, featuring frequency and voltage scaling, and flexible data relocation . . . 71
IV.3 Dynamic selection of optimized platform configurations for UMTS and WLAN security processing . . . 75
IV.4 Dynamic platform management methodology . . . 80
IV.5 Dynamic platform management for the StrongARM based platform: space of achievable data rates and CPU efficiency . . . 88
IV.6 CPU fraction left-over from security processing under static configuration and dynamic platform management . . . 89
IV.7 Energy savings using dynamic platform management: (a) varying UMTS and WLAN workload; (b) platform configuration sequence; and (c) cumulative energy profile . . . 91

IV.8 Block diagram of the Altera Excalibur SOPC 93
IV.9 Experimental setup for demonstrating dynamic platform management using the Altera Excalibur development board 94
IV.10 Dynamic platform management for Altera Excalibur: space of achievable data rates and CPU efficiency 96

V.1 Integrated Viterbi-Turbo decoder design: functional specification of (a) Viterbi decoder, (b) Turbo decoder, and (c) mapping of functional blocks to integrated decoder design 104
V.2 Viterbi and Turbo decoding data rate requirements and platform configurations that can satisfy them 107
V.3 Off-line Characterization Phase 109
V.4 For two applications: (a) data rate space characterization, and (b) overall application data rate space partitioning 113
V.5 Run-time Platform Configuration Phase 116
V.6 Data rate space achieved under different platform adaptation schemes 120

VI.1 Application-architecture co-adaptation for the integrated configuration of both applications and the platform architecture 124
VI.2 Wireless image delivery system 128
VI.3 Adaptive Wavelet Image Compression (AWIC) 129
VI.4 Concurrent execution of MPEG and different versions of the image application: timeline of (a) processor utilization, (b) PSNR and (c) energy consumption 131
VI.5 Application-architecture co-adaptation methodology 135
VI.6 Impact of image transformation level (TL) and quantization level (QL) on image quality, size and decompression time 137
VI.7 QL-TL Table used by run-time co-adaptation policy 138
VI.8 Total Latency and $PSNR$ for different versions of the image application under varying network bandwidth and CPU availability 143
VI.9 Distribution of total latency for image application under uniform random variation of CPU availability 144
VI.10 Energy savings under co-adaptation 145

VII.1 Future research directions: (a) architectural-level adaptation, (b) cross-layer adaptation, and (c) network-wide adaptation 148

LIST OF TABLES

III.1 Performance of the IEEE 802.11 MAC processor under different communication architectures 62
III.2 Performance of the UMTS Turbo decoder under different communication architectures 65

IV.1 Task characterization tables 81
IV.2 Characterization of the security processing tasks for the StrongARM based platform 87

V.1 Viterbi and Turbo decoding data objects 105
V.2 Viterbi decoding data rates under different data placement and bus configurations 106
V.3 Candidate data placement and bus configurations 118
V.4 Platform Configuration Table 119

ABSTRACT OF THE DISSERTATION

Dynamically Configurable System-on-Chip Platforms:

Architectures and Design Methodologies

by

Krishna Sekar

Doctor of Philosophy in Electrical and Computer Engineering (Computer Engineering)

University of California, San Diego, 2005

Professor Sujit Dey, Chair

Rapid advances in semiconductor technology have led to an era where entire systems, consisting of complex, heterogeneous components, can be integrated on a single chip, referred to as System-on-Chip (SoC). Due to the escalating cost in designing customized application-specific SoCs, recent years have witnessed the emergence of ”platform-based” SoCs. These systems consist of largely pre-designed, general-purpose components that can be re-targeted towards numerous applications, thereby amortizing design costs. A key determinant to the success of such platforms is the extent to which they can be customized to meet the diverse requirements imposed by different applications. Modern SoC platforms are mostly limited to providing a one-time (static) customization of the platform hardware. However, with the convergence of multiple diverse applications on the same platform, each imposing time-varying requirements, there is a growing need for SoC platforms that can be dynamically configured. Provisioning for such configurability and exploiting it at run-time is the focus of this dissertation.

This dissertation proposes SoC platforms featuring multiple, dynamic configurability options, and illustrates their advantages over existing design styles. It introduces the concept of Dynamic Platform Management, a methodology for the run-time customization of such platforms in a coordinated manner, to satisfy the time-varying requirements imposed by the executing applications. The dissertation addresses the prob-

lem of provisioning for dynamic configurability by proposing a novel, on-chip communication architecture that features a dynamically configurable topology. Dynamic management techniques are then presented for platforms featuring multiple, run-time configurable components. In particular, it considers platforms consisting of configurable processors, flexible memory architectures, and configurable communication architectures. Finally, it investigates the benefits of synergistically combining techniques for configuring platform hardware with techniques for adapting application behavior.

Experiments conducted on a large number of SoC designs, and an implementation of dynamic platform management on the Altera Excalibur development board demonstrate that configurable platforms with dynamic platform management result in significantly superior application performance, more efficient utilization of platform resources, and improved energy efficiency compared to conventional static approaches. Hence, the techniques described in this dissertation will facilitate more wide-spread adoption of the platform-based approach, leading to low-cost, yet function-rich and energy-efficient devices.

I

Introduction

Over the past four decades, the semiconductor industry has witnessed incredible improvements in the manufacturing process, resulting in integrated circuit feature sizes shrinking from 5.0 μm in 1975 to 65 nm (predicted) in 2007 [2]. Such remarkable technology scaling has led to the projected advent of the billion transistor chip within the end of this decade, keeping in line with Gordon Moore's prediction in 1965, famously known as Moore's Law, that the number of transistors on integrated circuits would double roughly every two years [3]. Designers today have the ability to integrate entire systems consisting of complex, heterogeneous components onto a single chip, referred to as a System-on-Chip (SoC). For example, the recently released GeForce 7800 GTX graphics processing unit from NVIDIA (Figure I.1) contains more than 300 million transistors on the same die [4].

The phenomenal complexity of today's SoCs has resulted in escalating costs and time associated with their design, verification, manufacture and test, making it economically infeasible to build custom, application-specific SoCs for many systems. Figure I.2(a) shows the cost of mask sets for integrated circuits over different technology generations [5]. Mask costs at the 65 nm node are estimated to run at more than $3 million, a 400% increase over the 130 nm node, while the total non-recurring engineering (NRE) costs may exceed $11 million. Also, while the available silicon real estate on chips has been rapidly increasing, our capability to meaningfully design and verify

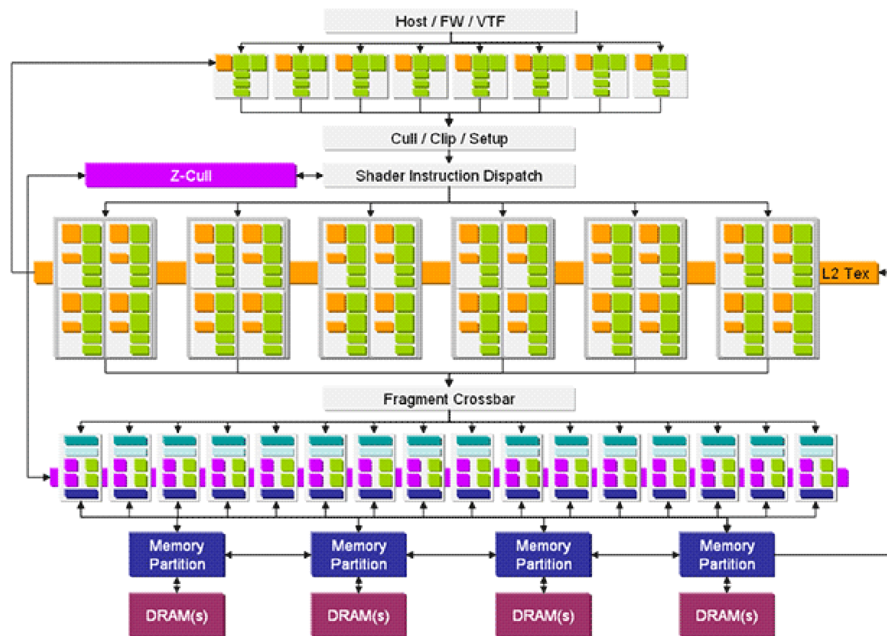
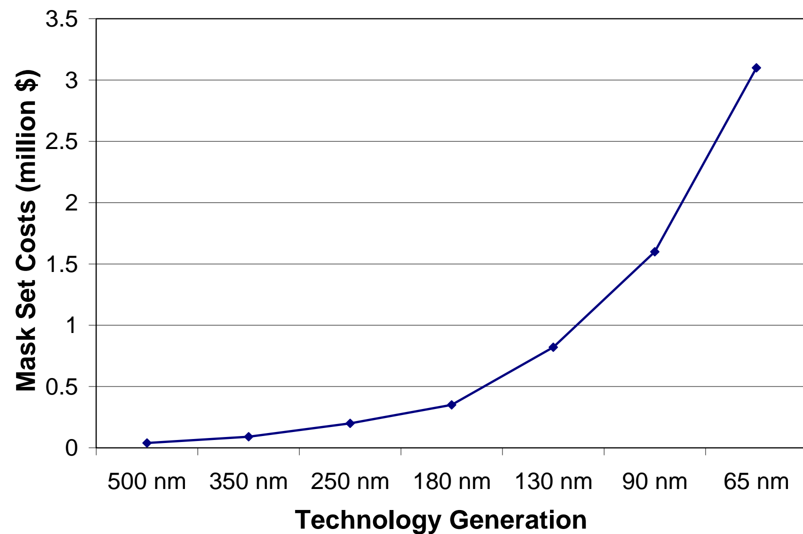
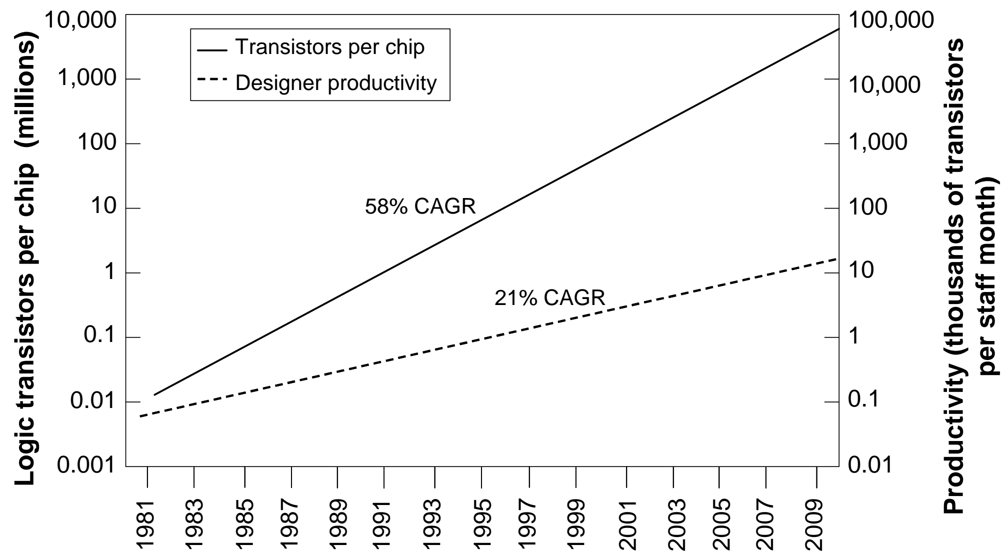


Figure I.1: GeForce 7800 graphics processing unit from NVIDIA with 302 million transistors

these complex chips continues to lag behind, creating the so-called "design productivity gap". This is illustrated by Figure I.2(b), which shows that the number of transistors per chip is increasing at a compound annual growth rate of about 58%, but designers' rate of productivity is increasing by only about 21% [6]. This gap adversely impacts the time-to-market for custom SoCs (or ASICs), a key factor determining the success or failure of a product in the fiercely competitive semiconductor market. It has been estimated that a three-month delay in the time-to-market of a high-value, high-volume application could cost $500 million [7]. These trends indicate that the custom SoC or ASIC approach may not be feasible except in the highest volume markets (quarter million plus units per year). This trend is confirmed by market surveys which indicate that the number of new commercial ASIC designs undertaken in 2002 was less than 1500, down from roughly 5000 in 1998-99, and that this slowdown was expected to continue even after the semiconductor industry recovered from its recession [8].



(a) Mask set costs for integrated circuits over different technology generations [5]



(b) Design productivity gap: difference in compound annual growth rates between transistors per chip and designer productivity [6]

Figure I.2: Increasing cost and design problems in developing custom SoCs/ASICs

The above trends are driving the research and development of new silicon architectures and design methodologies to combat the challenges faced by the custom SoC approach. One of the important results has been the emergence of the notion of "platform-based" SoCs, which consist of largely pre-designed and pre-verified *standard* (general-purpose) components integrated on the same chip. Examples of such components include processors, DSPs, embedded memories, standardized communication architectures, and peripherals such as UARTs, external memory controllers, timers, *etc*. Figure I.3 shows the Nomadik platform from STMicroelectronics, developed for the mobile multimedia domain [9]. Since platform-based SoCs consist of standard components, they can be targeted towards multiple applications, thereby amortizing the high cost of platform development over larger markets.

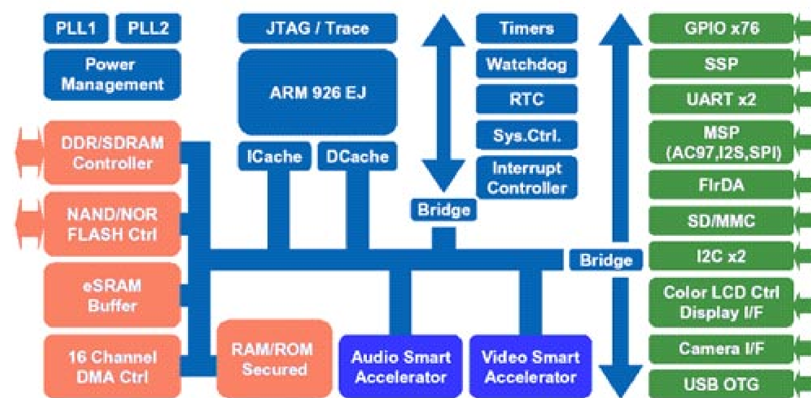


Figure I.3: Nomadik mobile multimedia platform from STMicroelectronics

A key aspect on which the success of an SoC platform depends is its ability to satisfy the performance requirements imposed by different applications that can be potentially mapped to the platform, while meeting desired system-level design goals (energy consumption, battery-life, cost, *etc*). Due to the diversity of application characteristics, it is imperative to be able to effectively customize such platforms in an application-specific manner, in order to best satisfy the requirements imposed by the executing application(s). Emerging trends in system design indicate that such configurable platform-based SoCs will play an increasingly important role in the future.

This chapter introduces the importance of provisioning for and exploiting configurability in SoC platforms, as a means of meeting many of the challenges faced by designers today. In the next section, we survey the landscape of design styles in detail, and illustrate the advantages and disadvantages of different design approaches. Next, we describe how configurable platforms can potentially combine the best benefits of different design styles. We motivate the need for dynamic configurability in platforms and run-time configuration techniques, as opposed to just static (design-time) configurability. Finally, we describe the contributions made by this thesis, and provide an overview of the remaining chapters.

I.A Landscape of Design Styles

Figure I.4 shows the landscape of different design styles, illustrating the tradeoff between general-purpose and application-specific design approaches. The x-axis represents increasing flexibility (*i.e.*, the ease with which the design can be targeted to different applications or varying application requirements), lower engineering costs and smaller time-to-market. The y-axis represents increasing performance and decreasing power consumption. We next describe each node in the landscape in further detail.

Custom ASICs/SoCs, ASSPs: Figure I.4 illustrates the position of custom SoCs/ASICs and ASSPs (application-specific standard products) in the design landscape. These designs are tailored to the requirements of a specific application and marketed to either one customer (ASICs), or a few customers (ASSPs). They are developed using the traditional IC development flows (*e.g.*, gate arrays, standard cells, full custom physical design). Such approaches result in highly customized, hard-wired solutions, which enable high application performance and low power dissipation. However, as described earlier, it is increasingly apparent that the large NRE costs and time-to-market associated with such designs may make them infeasible except in markets that command extremely high volumes.

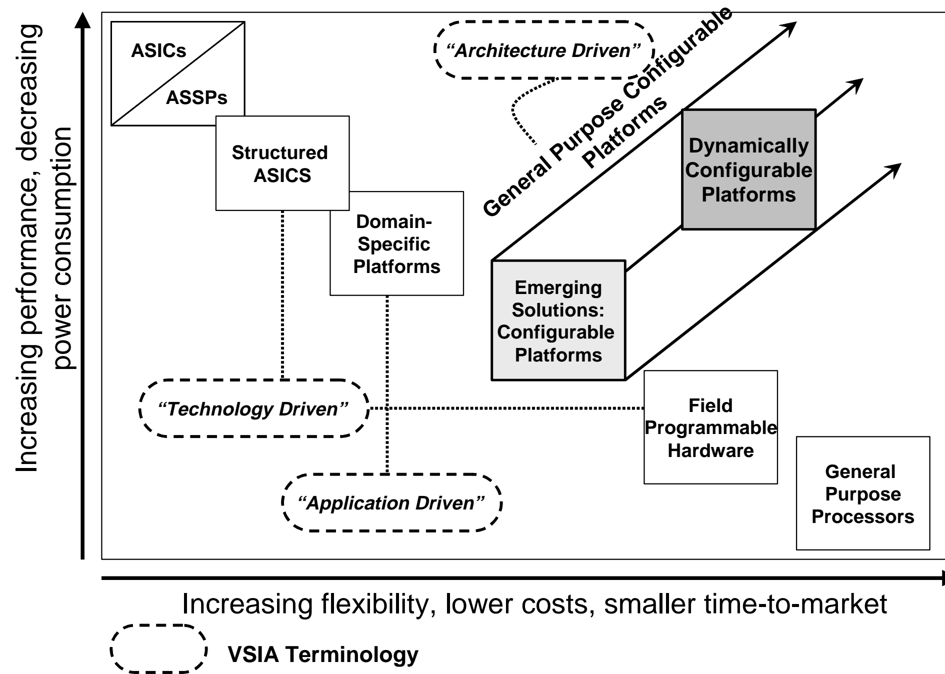


Figure I.4: Landscape of design styles and configurable platforms

General-Purpose Processors: At the other end of the spectrum are general-purpose processors, which provide maximum flexibility, allowing the same hardware architecture to be used across a variety of applications and application domains. Since they are programmed through software, they also result in short time-to-market and low engineering costs for their customers. Examples include processors from Intel, ARM and MIPS. However, software-only solutions often result in failure to satisfy performance and energy-efficiency requirements for many applications.

Structured ASICs: In recent years, several alternatives have started to appear that attempt to bridge the gap between application-specific and general-purpose architectures. Among them are emerging architectures that are based on the notion of a "structured ASIC". In these architectures, more than 50% of the metal layers are prefabricated, while a limited number of higher metal layers (typically 3 to 12) are available for application-specific customization. These devices are classified as "technology-driven" platforms as defined by the VSIA (Virtual Socket Interface Alliance) [10]. Sev-

eral semiconductor vendors are offering structured ASICs today, including NEC [11], ChipX [12] and LSI Logic [13]. These approaches are expected to reduce development time and mask costs, while paying a penalty in terms of reduced performance and higher power consumption compared to traditional ASICs.

Field Programmable Hardware: Advances in field programmable hardware (FPGAs) have started to make them a viable alternative to the custom ASIC approach in certain application areas [14]. These architectures consist of an array of programmable logic blocks with programmable interconnections, which allows the device to be reprogrammed multiple times after manufacture [15, 16]. In many small to mid-volume markets, where the costs of ASIC-style designs are not justifiable, variants of programmable hardware have started gaining substantial ground. The advantages of such pre-fabricated, "technology-driven" platforms are reduced NRE costs, smaller time-to-market and significant flexibility. However, limitations with respect to logic density, performance, power consumption and per unit cost impose large barriers to their introduction in larger volume markets.

Domain-Specific Platforms: Domain-specific platforms refer to systems that aim at catering to the needs of more than one customer/application, but within a particular application domain. Examples of these platforms, which can be classified as "application-driven" platforms [10], can be found in several domains, such as wireless handsets [17], network processing [18] and multimedia processing [9]. These platforms are characterized by hardware and software architectures that are customized towards the requirements of applications typically encountered in a certain domain, such that, with minimal engineering effort, they can be modified to meet the needs of several customers. Since these platforms are customized towards particular application domains, they provide good performance, low engineering costs and short time-to-market for users of the platform. However, domain-specific platforms suffer from three key limitations. First, in this approach, it is critical to identify the defining characteristics of a domain, and develop platforms that best satisfy them. This calls for large develop-

ment efforts for the platform developer. Second, the need to develop extensive software infrastructure for each new platform, such as compilers, platform simulators, operating system support, API's *etc.*, can mean significant cost and effort for the platform vendor. Finally, such platforms have to be sold in volume in order to justify their development costs, which is difficult since they are targeted towards a specific application domain.

All the above approaches have limitations that can be addressed by an emerging approach to system design, namely the use of general-purpose configurable platforms. These platforms fall into the category of "architecture-driven" platforms, as recently defined by the VSIA [10]. In the next section, we examine trends in these types of systems in more detail.

I.B General-Purpose Configurable Platforms

General-purpose configurable platforms are characterized by the use of general-purpose components (*e.g.*, processors, caches, memory sub-systems), which enable the use of the same platform across a variety of applications and domains. To address the inefficiencies associated with general-purpose architectures (poor performance, high power consumption), these platforms provide opportunities for application-specific customization through configurability of the underlying platform hardware. Figure I.4 shows where such platforms can be placed in the design landscape.

The success of such platform-based systems largely depends on the extent to which, and the ease with which, the platform can be customized for an application. These requirements translate into the need for (i) configurable architectural components and parameters in the platform architecture, and (ii) methodologies that help optimize the configuration of the platform to the characteristics of the executing application(s). The configurable features available in a general-purpose configurable platform can be distinguished based on whether they enable static or dynamic platform customization, although the platform architecture may in general, consist of both statically and dynamically configurable components and parameters.

I.B.1 Static Configurability

Statically configurable platform components allow system designers to customize the platform architecture once, either during platform design using soft, configurable IP, or through one-time programming or configuration of pre-fabricated, hardware platforms. For example, Xtensa from Tensilica [19] and Nios from Altera [20] are configurable processor cores, which allow designers to add custom instructions to the instruction set and associated extra logic to the processor data-path. These processor cores and the MIPS 4KE processor family [21] also feature configurable caches whose size and associativity can be customized at design time. Various popular bus architectures, such as AMBA from ARM [22] and CoreConnect from IBM [23], enable the bus topology, widths and protocol to be statically customized.

As statically configurable platforms have started to appear commercially, numerous on-going efforts aim at providing accompanying tools and methodologies to design systems based on such platforms. These methods aim at efficiently mapping applications to the platform through static customization of the underlying architecture, so as to improve the performance and/or energy-efficiency of the system. Examples of such tools include PICO Express from Synfora [24] and Platune from UC Irvine [25].

I.B.2 Dynamic Configurability

The emerging trend of convergence of different applications on the same platform is driving the need for even greater configurability in SoC platforms than those offered by the approaches described in the previous sub-section. An illustration of this trend is provided in the domain of wireless handsets, where diverse, numerous applications, with widely different characteristics are converging onto the same device. Examples of such applications include phone, personal digital assistant (PDA), camera, web browser, email, global positioning (GPS) and MP3 player. As illustrated in this thesis, as different applications execute on the same platform (perhaps simultaneously), statically customizing the platform for a particular application may prove insufficient. This is because (i) different applications have different characteristics and impose dif-

ferent requirements on the underlying platform, and (ii) the requirements imposed by an application can also change dynamically, depending on application phase, performance requirements, and the properties of the application data.

These trends motivate the need to develop platform architectures that provision for *dynamic* configurability. Technologies for providing such configurability in individual platform components have started emerging, and are surveyed in Chapter II. Examples of such components include dynamically configurable processors, caches, memory sub-systems, on-chip communication architectures, and on-chip configurable fabrics. In order to best optimize platforms to meet the time-varying requirements of the executing applications, there is a need for SoC platforms that consist of *multiple* dynamically configurable components. Note that, we distinguish such platforms from those based entirely on symmetric arrays of processing elements, such as QuickSilver [26] and PACT XPP [27], which feature reconfigurability at the micro-architecture level through the use of programmable interconnections. While these platforms hold promise for providing dynamic configurability, their widespread adoption is challenged by the lack of established programming models.

Techniques for the dynamic configuration of individual platform components and parameters to particular applications have been studied in the past (discussed in detail in Chapter II). However, the need to execute multiple time-varying applications as well as the multiple opportunities for dynamic customization available in SoC platforms, creates new system design challenges. There is a need for new methodologies and design tools that can understand the changing requirements imposed by applications, and select and apply optimized configurations for the platform components at run-time. For the best benefit, such techniques need to take a holistic approach to on-line platform customization by taking into consideration the interactions between platform components and their configurations.

In the next section, we describe the contributions that this thesis makes in addressing many of the above challenges. We then present an overview of the remaining chapters.

I.C Thesis Overview and Contributions

This thesis proposes the use of general-purpose SoC platforms consisting of multiple, dynamically configurable components, as a means of combining the benefits of both application-specific and general-purpose design styles. However, naive use of such platforms can result in a large performance and energy-efficiency gap compared to more customized solutions. To this end, this thesis introduces the concept of *Dynamic Platform Management*, a methodology for the run-time customization of a general-purpose configurable platform to the time-varying requirements of the applications executing on the platform. Dynamic platform management is implemented as a platform middleware layer that monitors the run-time requirements imposed by the applications, and when appropriate, optimizes the configuration of the underlying platform by exploiting knowledge about the application characteristics. In this approach, targeting a platform to a new application involves customizing the platform management algorithms, hence, avoiding the cost and effort of re-designing, or introducing new platform hardware.

We present a detailed survey of existing and emerging technologies for provisioning for dynamic configurability in platform components. From this, we identify the on-chip communication architecture as a critical determinant of overall performance in complex SoC designs, making it crucial to dynamically customize it to run-time application traffic characteristics. For this, we propose FLEXBUS, a flexible, bus-based on-chip communication architecture featuring a dynamically configurable *topology*. FLEXBUS is designed to detect run-time variations in system-wide communication traffic characteristics, and efficiently adapt the *logical connectivity* of the communication architecture and the components connected to it. This is achieved using two novel techniques, *bridge by-pass* and *component re-mapping*, which address configurability at the system and component levels, respectively. We also present dynamic bus configuration policies for choosing optimized FLEXBUS configurations under time-varying traffic characteristics.

We next propose two types of general-purpose platforms featuring multiple configurability options: (i) platforms that feature fine-grained frequency and supply

voltage scalable processors, and flexible data relocation, which allows application data structures to be dynamically partitioned among the system memories, and (ii) platforms that feature flexible data relocation and reconfigurability of the communication architecture topology (FLEXBUS). We present methodologies for dynamic platform management for both types of platforms, taking into consideration the interaction between the configurable components, to optimize the usage of available CPU, hardware, memory and on-chip communication resources.

Complementary to platform customization is the requirement that the applications themselves also be customized to the characteristics of the platform on which they execute. Many existing and emerging applications provide run-time configurability in terms of algorithmic parameters, or even which algorithms to use, thereby enabling a tradeoff between application quality and the load imposed on the platform. This thesis explores *application-architecture co-adaptation* techniques, in which both the applications as well as the underlying platform architecture are dynamically and *synergistically* configured to improve application performance and energy-efficiency. The proposed approach, which is described in the context of a wireless image delivery system, results in highly customized application-architecture solutions.

Detailed experiments were conducted on a wide variety of example SoC designs to evaluate each of the proposed techniques. In addition, dynamic platform management was implemented on the Altera Excalibur development board [28] as a proof-of-concept. The experiments demonstrate that dynamically configurable platforms with dynamic platform management result in significantly superior application performance, more efficient utilization of platform resources and improved energy efficiency compared to conventional static approaches.

The remainder of this thesis is organized as follows:

- In Chapter II, we present a survey of technologies for providing dynamic configurability in individual platform components, and techniques for exploiting such configurability. We then present the concept of dynamic platform management for the integrated, run-time, application-specific customization of such platforms.

- In Chapter III, we present FLEXBUS, a flexible, bus-based on-chip communication architecture featuring a dynamically configurable topology, and describe techniques for customizing it at run-time.

- In Chapter IV, we present configurable platforms featuring frequency and voltage scalable processors, and flexible data relocation, and describe dynamic platform management techniques for optimizing the platform configuration depending on time-varying application requirements.

- In Chapter V, we present configurable platforms featuring flexible data relocation, and dynamically configurable communication architectures (FLEXBUS), and describe dynamic platform management techniques for the integrated, application-specific configuration of such platforms.

- In Chapter VI, we present application-architecture co-adaptation techniques for the integrated, run-time customization of both the applications and the platform architecture, and describe it in the context of a wireless image delivery system.

- Finally, in Chapter VII, we discuss future research directions that can be pursued based on the work presented in this thesis.

II

Dynamically Configurable Platforms with Dynamic Platform Management

II.A Introduction

As discussed in Chapter I, platform-based SoC design proffers significant benefits over custom SoC approaches. Consequently, the market for SoC platforms has been rapidly increasing, with a number of semiconductor vendors offering platforms targeted towards different market segments. One of the areas in which such platforms are playing an important role is the wireless handset market. In this domain, severe limitations on device cost, size and power consumption, together with the need for high performance (to support demanding wireless applications and protocols), software upgradability (due to evolving standards and applications), and short product cycles make configurable platforms an attractive approach. Commercially available SoC platforms for this domain include OMAP from Texas Instruments [17], Nexperia Mobile from Philips [29] and PrimeXsys from ARM [30]. Such platforms are typically customized statically (at design-time) to the application(s) they need to support. For example, several versions of the OMAP platform are available, with different configurations (different processors, hardware accelerators, *etc.*) for different wireless market segments [31], while Improv

Systems' Programmable System Architecture (PSA) is a soft platform that enables designers to configure it for a given application [32].

The need for dynamic configurability in SoC platforms stems from the increasing number of domains in which systems need to execute multiple (possibly concurrent) applications. For example, smartphones are expected to support applications such as phone, personal digital assistant (PDA), camera, video, web browser, email, global positioning (GPS) and MP3 player. Similarly, television, internet and telephony are all expected to be available through the same home set-top box. Since different applications can have widely different characteristics, significant temporal variation may occur in the manner in which underlying platform resources are used, depending on which application is executing, or the concurrent mix of applications. Furthermore, applications, and their operating environments, can impose a wide range of processing requirements, due to variations in performance criteria, available battery capacity, and properties of the data being processed. This makes it imperative to be able to adapt the underlying platform architecture to these changing requirements. Along with dynamic configurability in SoC platforms, techniques for efficiently exploiting such configurability are crucial. These techniques should be able to understand the time-varying requirements imposed by the executing applications, and optimize the platform configuration in an integrated manner at run-time.

II.A.1 Chapter Overview

In this chapter, we present a detailed survey of existing and emerging technologies for providing dynamic configurability in SoC platforms. We consider, in turn, configurable processors, caches, memory sub-systems, on-chip communication architectures, and on-chip configurable fabrics. We also describe associated techniques for exploiting the configurability available in such platforms. However, most of these techniques focus only on the configuration of individual platform components. SoC platforms featuring multiple opportunities for dynamic customization, executing multiple time-varying applications, create new system design challenges, requiring new method-

ologies and design tools. For this, we introduce the concept of dynamic platform management, a methodology for the run-time, application-specific customization of configurable SoC platforms. Dynamic platform management is implemented as a platform middle-ware layer that understands and exploits knowledge of the applications and their characteristics, and manages and configures the platform resources in a holistic manner. We conclude by describing the advantages of dynamically configurable platforms with dynamic platform management.

The rest of this chapter is organized as follows. In Section II.B, we present a survey of dynamically configurable platform components, and associated configuration techniques. In Section II.C, we describe the concept of dynamic platform management. We conclude this chapter in Section II.D.

II.B Dynamically Configurable Platform Components and Configuration Techniques

Recent years have witnessed the emergence of several dynamically configurable features in platform components, as well as techniques to exploit such configurability, in order to improve system performance and/or power efficiency.

II.B.1 Configurable Processors

There has been a growing interest in dynamically configurable processors, especially in the low-power domain, where researchers have proposed several techniques to provide processors with the capability to dynamically tradeoff performance for power-efficiency. Two broad dimensions of configurability available in such processors include (i) micro-architectural flexibility, and (ii) frequency and voltage scalability.

Micro-Architectural Flexibility

Configurable Features: Several configurable architectural features have been proposed in the literature recently, mainly in the context of dynamically scheduled,

superscalar processors. These techniques provide the capability to activate or deactivate processor components at run-time, enabling dynamic upgrades to processor performance when required, and energy savings at other times. Examples of micro-architectural features that can be dynamically configured include the number of processor functional units and the instruction issue width (which defines the number of instructions that can execute simultaneously) [33, 34, 35], the instruction window size [36, 37, 38], the processor pipeline [39], speculation control logic [40, 41], and sizes of the register update unit [35], reorder buffers and load-store queue [38].

Another type of micro-architectural flexibility is provided by the integration of an FPGA-based reconfigurable functional unit into the pipeline of a processor [42, 43]. This enables performance improvements by allowing the addition of new application-specific instructions to the processor, which are executed by dynamically loading the appropriate instruction-specific configuration into the reconfigurable functional unit.

Configuration Techniques: Techniques to configure processor components are in large part, based on monitoring the instruction-level parallelism (ILP) exhibited by a program during execution, which can vary by up to a factor of three within the same program [44]. During periods of low ILP, under-utilized processor components are disabled to save energy; they are activated when the ILP is high so that performance is minimally impacted. *Pipeline balancing* is a technique that dynamically regulates the issue width as well as the number of active functional units depending on the available ILP [33, 34]. The advantages of exploiting ILP have also been applied to dynamically configure the instruction issue queue size [36, 37], as well as the sizes of the reorder buffer and load-store queues [38] .

Pipeline gating is a technique that uses an adaptive speculation control algorithm to determine if a branch is likely to incur a misprediction, and if so, prevents wrong-path instructions from entering the pipeline [40]. Adaptive speculation control has also been used for thermal management, which refers to the prevention of processor "hot spots" [41].

Techniques for exploiting configurable pipelines have been proposed, wherein the processor pipeline is dynamically switched between out-of-order, in-order and pipeline gating modes, depending on application performance goals, which may be externally specified either by the application itself, or by the operating system [39].

Frequency and Voltage Scalability

Configurable Features: Frequency and voltage scalable processors feature the ability to change their operating frequency and voltage level at run-time. Static CMOS based processors have a voltage-dependent maximum operating frequency. Hence, when the operating frequency is lowered, the supply voltage can be lowered as well, leading to quadratic improvements in energy consumption [45]. Dynamic frequency and voltage scaling (DVS) is enabled through the use of programmable clock generators (PLLs) and programmable, variable voltage DC/DC converters [46, 47]. Numerous commercial processors are DVS enabled today, including Transmeta's Crusoe [48], Intel's XScale [49], and AMD's Mobile K6-2+ [50] processors.

Configuration Techniques: The flexibility of being able to change the processor voltage and frequency enables exploitation of the fact that workloads of processors exhibit significant run-time variation. With these processors, it is possible to reduce the clock frequency and voltage during periods of reduced activity, and thereby save energy. Several DVS algorithms have been proposed in the past, and it continues to be an active area of research. Such algorithms can be classified based on their applicability to either non real-time systems or real-time systems. Most DVS algorithms for non real-time systems (*e.g.*, workstation-like environments) are based on *interval-based* voltage scheduling, where the frequency and voltage is set for a fixed-time interval based on the processor utilization over the previous time-interval(s); if the processor utilization is low, the frequency and voltage is scaled down, if it is high the frequency and voltage is scaled up [51, 52, 53, 54]. DVS algorithms for real-time systems (both hard and soft) exploit the slack available in real-time workloads and choose the minimum frequency and voltage setting such that the tasks "just" meet their deadlines, thus saving

energy without impacting performance [55, 56, 57, 1, 58]. Real-time DVS algorithms are usually integrated or closely interact with the OS task scheduler.

Techniques for exploiting multiple such configurable features of processors have also been proposed in the literature. These techniques have the notion of a *configuration period*, which defines the granularity of configuration. This could be a fixed time interval, or a period during which program "phase" remains constant. The configuration period consists of two phases: (i) a testing/tuning phase, when all the available configurations are tested back to back to identify the best configuration, and (ii) the adaptation phase, when the best configuration is applied for the rest of the configuration period. An integrated processor configuration framework that exploits both micro-architectural flexibility and DVS is described in [59]. A similar framework targeted towards multimedia applications, in which configuration decisions are made at frame granularity, is proposed in [60]. The use of working set *signatures* for guiding configuration decisions has been proposed in [61]. These highly compressed representations of working sets help detect program "phase" changes, and trigger new configuration periods. In contrast to these *temporal adaptation* schemes, where each configuration period is tied to successive time intervals, *positional adaptation* schemes have also been proposed, where configuration periods are associated with position, namely particular code sections, in order to better track dynamic program behavior [62].

II.B.2 Configurable Caches

Traditional cache architectures are fixed and optimized to perform well in an average sense, often incurring large performance variations across applications and even across different phases of the same application. Recognizing the performance and power advantages of configurable caches, a number of configurable cache architectures and techniques to exploit such configurability have been proposed.

Configurable Features: The Motorola M*CORE M340 processor is an example of a commercial processor with a configurable cache architecture [63]. The 8-KB, 4-way set associative unified cache features three configurability options programmable

via a cache configuration register: (i) write mode selection between write-through or copyback (write-back), (ii) way management to selectively enable or disable one or more ways of the cache, and (iii) the ability to enable or disable the store buffer and the push buffer. Setting the write mode to copyback enables better performance and power efficiency, while setting it to write-through enables more efficient system coherency management when the cache is shared. Options (ii) and (iii) can be used to achieve a tradeoff between performance and power consumption.

Several dynamically configurable features for caches have also been proposed in the literature. Such configurable features include the ability to change the cache size, associativity and line size, as well as the ability to perform dynamic cache partitioning. Set-associative caches can be dynamically resized by selectively enabling or disabling a subset of the ways to achieve performance-power tradeoffs as in *selective cache ways* [64]. For direct-mapped caches, a subset of the cache sets can be selectively shutdown such as in the *DRI i-cache* (dynamically re-sizable instruction cache) [65], which targets low leakage power consumption in instruction caches. *Way concatenation* is a technique for changing the cache associativity, while still utilizing the full capacity of the cache, by concatenating different ways of the cache [66]. Cache line size configuration through the use of a small fixed-size physical line but a variable-size "virtual line" has also been proposed [67].

Dynamic cache partitioning refers to the ability to divide the cache memory space into multiple partitions at run-time, which can be used for different processor activities. *Column caching* is one such technique, where different caches and memories (*e.g.*, spatial and temporal caches, scratch pad memory) can be mapped to different sets of cache columns or ways [68]. Such cache partitioning at the granularity of cache ways is also proposed by [69]. The fixed size cache memory can also be dynamically partitioned into variable sized L1 and L2 caches [70].

Configuration Techniques: Several techniques have been proposed for performance and/or power efficiency through intelligent run-time application-specific configuration of caches. In the *accounting cache*, the least-recently-used (LRU) state infor-

mation is used to determine cache activity and select the optimal subset of cache ways that should be active [71]. This enables power savings by shutting down parts of the cache during periods of modest cache activity, while minimally impacting performance by making the full cache operational during more cache intensive periods. Similarly, in the DRI i-cache, selected sets of the cache are automatically shut down to save leakage power, but here the cache miss-rate is used as an indicator of cache performance [65]. Techniques for exploiting configurable cache line sizes have also been proposed, where the line size is adapted to application-specific requirements based on cache line usage monitoring [67].

Dynamically partitionable caches have been used to automatically partition the cache memory among multiple simultaneously executing processes or threads in order to prevent cache "pollution", and hence improve overall system performance [72]. Such caches can also be used to improve the performance of media processing applications by using a portion of the cache for storing an instruction reuse buffer [69].

II.B.3 Configurable Memory Sub-Systems

We survey two types of flexibility available in the memory sub-system: (i) dynamic data relocation, and (ii) operating power mode setting.

Data Relocation/Remapping

Configurable Features: Data relocation refers to the ability to change the location or placement of application data objects or items in memory after they have been allocated (*i.e.*, at run-time). This enables a wide range of data layout optimizations to enhance cache and memory system performance, and performs better than a statically optimized data layout approach by being able to adapt to dynamic program behavior. To apply data relocation, the correctness of the program after re-mapping has to be guaranteed, *i.e.*, all future references to the relocated object must find it at its new location. For data objects where perfect aliasing information about all references to the object can be computed (*e.g.*, for regular arrays), the compiler can ensure program correctness using

pointer update and array renaming techniques. When such pointer aliasing cannot be done (*e.g.*, for heap allocated objects in languages like C), the virtual memory system can provide a limited form of safe data relocation at the granularity of a page by copying the page and updating the virtual-to-physical address mapping [73]. Finer granularity data relocation can be enabled using hardware base pointer registers, which store the base address of each relocatable data object. Each address is generated by adding an offset to the value in the base pointer register, and hence for correct address generation on data relocation, the corresponding base pointer registers' values are changed to the new base address of the relocated data. Word level data relocation granularity can be provided using a hardware-assisted technique called *memory forwarding*, where on object relocation, its new address is stored in its old location, and the old location is marked as a *forwarding address* [74]. If the program accidentally accesses the old address, the hardware automatically forwards the reference to the new location, thereby guaranteeing the correct result.

Configuration Techniques: We next survey techniques that take advantage of such flexible data relocation to improve cache and memory system performance. By packing data objects, which are accessed close together in time but scattered sparsely throughout the address space, into adjacent memory locations, the spatial locality and prefetch effectiveness of caches can be improved, leading to lower capacity, compulsory and conflict misses. Such an approach has been proposed for affine programs, where the layout of the data arrays for different program segments is determined and associated bookkeeping code to perform data relocation is inserted into the program at compile-time [75]. The actual data relocation takes place at run-time when the bookkeeping codes are executed. *Data copying* is a technique to reduce conflict misses within tiled (or blocked) applications, in which a tile is first copied to a contiguous set of memory addresses before usage since these locations do not conflict with each other [76, 77]. Another technique called data coloring reduces conflict misses in pointer-based data structures like trees, where data elements with high temporal locality are relocated to memory addresses that map to non-conflicting regions of the cache [78]. *Dynamic data*

packing based on the inspector-execute method has been proposed for improving the spatial locality of dynamic and irregular programs where effective static analysis is not possible [79].

Embedded systems often use on-chip scratch-pad memories instead of data caches to provide predictable execution time, reduce power consumption, and exploit application characteristics [80]. Compiler-directed techniques based on relocating portions of the data arrays from the off-chip memory to the scratch-pad memory when required, can be used to efficiently manage the scratch-pad memory space among multiple data arrays [81].

Operating Power Mode Setting

Configurable Features: Modern DRAMs support multiple operating power modes, where in each mode, different components of the memory system are disabled leading to different power consumption characteristics. For example, RDRAM technology [82, 83] provides six different operating modes (power down, nap, standby, attention, attention read and attention write) with power consumption ranging from 1.4 mA in the power down mode to 635 mA in the attention write mode (for Direct RDRAM 128/144-Mbit [82]). The operating mode can be set by programming control registers in the memory controller. This can be used to place currently inactive memory banks in lower power modes, leading to lower energy consumption. However, memory banks in low power modes incur an additional *re-synchronization delay* when they have to service a request; typically lower the power mode, higher the re-synchronization delay.

Configuration Techniques: The multiple operating modes of the memory system can be exploited using either hardware-based self-monitoring techniques, or under software control. Current memory controllers feature a limited amount of self-monitored power-down [84, 85], where the memory system automatically transitions to a power-down state if there is no memory activity for a specified number of clock cycles. More sophisticated schemes involving multiple memory power modes and hardware-based idle time prediction have also been proposed [86, 87]. Under software-based

schemes, page allocation strategies of the operating system to exploit the memory low power modes have been studied [87]. A compiler-based approach is investigated in [86], where the application program is statically analyzed to detect memory idleness and corresponding mode transition code is inserted into the program at compile-time.

An energy reduction scheme exploiting both dynamic data relocation and multiple power modes is proposed in [88]. In this scheme, data arrays with temporal affinity are dynamically relocated to the same set of memory banks, thus increasing the number of memory banks that can be transitioned to low power modes.

II.B.4 Configurable On-Chip Communication Architectures

On-chip communication architectures (*e.g.*, shared buses, crossbar switches) are increasingly playing an important role in determining the performance and power consumption of SoCs. Dynamic configuration of the communication architecture promises significant improvements in system throughput and/or power efficiency. We next survey configurable features and configuration techniques in this domain.

Configurable Features: Several commercial communication architectures feature dynamic configurability of the communication protocol. The Sonics Silicon-Backplane [89] is a time-division multiple access (TDMA) based communication architecture featuring, for each bus-master, software-programmable arbitration mechanism (TDMA or fair round-robin) and bandwidth allocation. Both variable-length burst sizes and software programmable bus-master arbitration priority are also available in commercial communication architectures such as CoreConnect from IBM [23] and AMBA from ARM [22]. CoreConnect provides an additional degree of configurability by allowing each bus master to indicate a desired priority to the arbiter for each bus request.

In the research domain, we have proposed FLEXBUS, a flexible, bus-based, on-chip communication architecture featuring a dynamically configurable topology (details in Chapter III). Also, a hybrid current/voltage mode signaling based bus architecture has been proposed, that can be switch between the two modes [90].

Configuration Techniques: We have proposed automatic configurations policies for FLEXBUS that changes the communication architecture topology depending on application traffic characteristics (Chapter III). *Communication architecture tuners* have been proposed to monitor the communication requirements of each SoC component and configure the bus protocol parameters for improved system performance [91]. A circuit-level technique to automatically switch a hybrid current/voltage mode signaling bus to the current mode (higher throughput) during periods of high bandwidth requirement, and to voltage mode (lower power) during relatively idle periods, based on bus transition counting, is proposed in [90].

II.B.5 On-Chip Configurable Fabrics

The recent appearance of single-chip platforms combining a reconfigurable fabric with a microprocessor provides designers with another degree of configurability. These platforms enable flexible migration of compute-intensive software functionality to hardware, thus providing significant performance and power benefits. We next survey such commercially available platforms and platform configuration techniques.

Configurable Features: The on-chip reconfigurable fabric can either be fine-grained (*e.g.*, FPGA-based) or coarse-grained (*e.g.*, array of functional blocks). The Altera Excalibur SOPC (system-on-a-programmable-chip) is an example of a platform featuring fine-grained reconfigurability, combining an ARM922T core along with up to a million programmable gates on the same chip [28]. Other examples include the Xilinx Virtex-II Pro [92] with up to four PowerPC 405 processors and over 125 K logic cells, and the Triscend [93] A7 and E5 featuring an ARM7 and an 8051 microprocessor, respectively, with up to 40 K logic gates. Atmel [94] offers a similar platform combining an 8-bit AVR microprocessor with up to 40 K gates.

Elixent offers a coarse-grained configurable fabric called D-Fabrix [95], consisting of an array of ALUs, which has been integrated with the Toshiba MeP (media embedded processor) configurable processor core [96] on the same chip [97].

Configuration Techniques: Most techniques for configuring such platforms are compile-time, consisting of off-line application profiling to identify the performance critical code sections (*e.g.*, functions, subroutines, loops), generation of FPGA configuration bit-streams to map them to the FPGA, and application code instrumentation to load the appropriate bit-stream into the FPGA when required [98, 99]. The actual reconfiguration takes place at run-time when the application code is executed. Dynamic hardware/software partitioning has also been proposed, where the above steps are performed on-chip at run-time, in order to ease designer burden [100].

II.B.6 Summary

Dynamic configurability in individual platform components has been studied extensively as surveyed in this section. However, it is important to provide more configurability so that SoC platforms can be adapted better to application characteristics. In the next chapter, we focus on the on-chip communication architecture, a crucial system component in determining the performance of complex SoCs, and describe techniques to provision for and exploit dynamic configurability in the communication architecture topology. Furthermore, in order to realize the full potential offered by configurable platforms, there is a need for platforms featuring multiple configurable components. In addition, while techniques for the configuration of individual platform components have been proposed, there is a lack of comprehensive methodologies for the configuration of multiple such configurable components. Platforms featuring multiple opportunities of configuration, coupled with a holistic platform management methodology will provide significantly higher performance and energy efficiency compared to current schemes. In the next section, we introduce dynamic platform management, a methodology for such integrated platform customization.

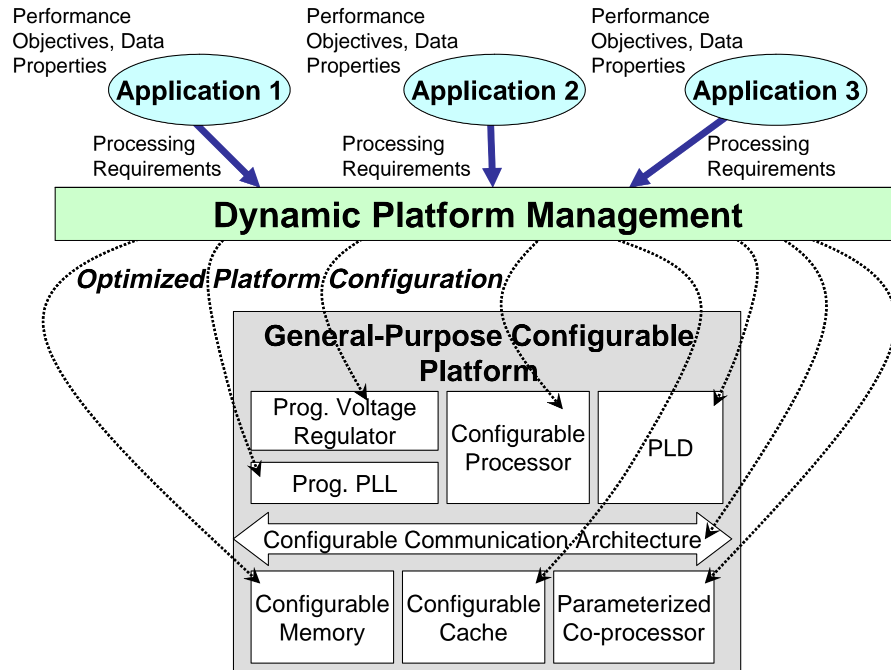


Figure II.1: Dynamically configurable platforms with dynamic platform management

II.C Dynamic Platform Management

This thesis considers SoC platforms featuring multiple, dynamically configurable components and architectural parameters (Figure II.1). Such platforms would be required to support multiple applications, possibly executing concurrently. Each application would have different characteristics, performance objectives and properties of the data being processed, which can change over time, thereby imposing time-varying processing requirements on the underlying platform architecture. To satisfy these requirements, we propose *dynamic platform management* for run-time platform customization (Figure II.1). Dynamic platform management monitors the requirements of the executing applications, and when appropriate, optimizes the configuration of the underlying platform to best suit these requirements. Since, the operation of different platform components may be interdependent, the configuration of one component may determine the selection of the optimized configuration of the other. For example, the configuration of the data placement in memory would determine the execution time of different ap-

plications, thereby affecting how much the frequency and voltage of the system can be scaled (described in Chapter IV). Therefore, dynamic platform management takes a holistic approach to on-line platform customization by taking into consideration the interaction between platform components and their configurations.

The dynamic platform management approach consists of two phases: (i) off-line characterization phase, and (ii) run-time platform configuration phase. During the off-line phase, we characterize the execution of the different applications on the platform by determining how platform resource usage (*e.g.*, CPU cycles, memory accesses) varies with application characteristics. This is performed using a combination of analysis and simulation. During the run-time phase, the platform management algorithms execute on the platform, and select and apply optimized platform configurations based on the off-line information. These steps are described in detail for two dynamically configurable platforms in Chapters IV and V.

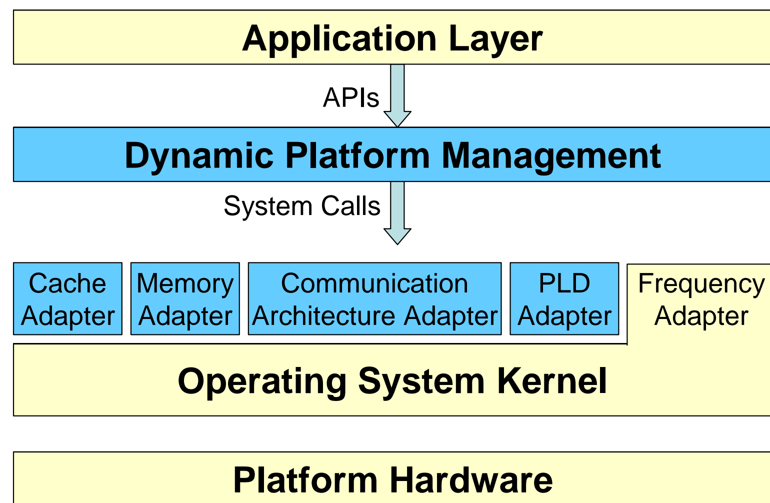


Figure II.2: Software architecture for dynamic platform management

Run-time platform management is implemented as a platform middle-ware layer that communicates with the applications, in order to identify run-time variations in their requirements, and with the underlying operating system kernel or platform hardware, in order to apply the selected configuration. Figure II.2 illustrates the software architecture for implementing dynamic platform management. The platform manage-

ment layer provides a set of APIs (application programming interface) through which the executing applications can provide information about their run-time characteristics (performance requirements, data properties, *etc.*). It then determines the optimized configuration for different platform components. The selected configuration is applied by invoking "adapters" for each configurable component using system calls. The adapters are device-drivers that interact with the hardware to perform the requested adaptation. The adapter implementation is specific to the platform and is tied closely to the operating system. For some configurable features, the adapters are available as part of the operating system kernel, such as support for frequency scaling in Linux [101]. For other configurable components, such as the on-chip communication architecture, the adapters need to be provided. Chapter IV describes how this framework is implemented for the Altera Excalibur SOPC (system-on-a-programmable-chip) [28].

Compared to traditional design approaches, designing systems based on configurable SoC platforms with dynamic platform management brings about several advantages:

- Modifying platform functionality only requires introducing new application software and upgrading the platform management middle-ware, instead of incurring an expensive redesign. Hence, such platforms, once deployed, can enjoy a long market life via software upgrades.
- The same platform can be targeted to a wide variety of applications, facilitating the amortization of non-recurring costs over larger markets.
- The platforms are designed using general-purpose, commercial, off-the-shelf components, which leads to reduced platform development costs, since large resources need not be deployed to identify and develop domain-/application-specific components.
- Since the process of application-specific customization is performed in software rather than hardware, design using dynamic platform management techniques result in shorter time to market, productivity gains, and reduced development costs.

II.D Conclusions

In this chapter, we surveyed several technologies that have been developed to enable the dynamic configuration of individual platform components. Many of these technologies have reached relative maturity, a few of them having made their appearance in commercial products. However, there is a need for providing more configurability in platform components, as well as developing platforms consisting of multiple configurability options, in order to better adapt platforms to application characteristics. We also surveyed techniques for the configuration of individual platform components, and pointed out the need for integrated techniques for the management of platforms featuring multiple configurability options. Finally, we described our dynamic platform management approach for such holistic configuration of SoC platforms, and described its potential benefits. In the remaining chapters of this thesis, we elaborate on the platform management concept, and describe how it is applied to specific configurable SoC platforms.

The text of this chapter, in part, is based on material that has been published in the International Conference on VLSI Design, 2004, and material submitted to the IEEE Transactions on Computer-Aided Design of Circuits and Systems. The dissertation author was the primary researcher and author, and the coauthors listed in these publications collaborated on, or supervised the research that forms the basis for this chapter.

III

Dynamically Configurable Bus Topologies for High-Performance On-Chip Communication

III.A Introduction

In the previous chapter, we presented several emerging technologies for providing dynamic configurability in platform components. We also discussed the need for more configurability options, so that the platform can be better adapted to applications' requirements. This chapter addresses the problem of providing such configurability in the on-chip communication architecture, which has emerged as a critical determinant of overall performance in complex SoCs. This is because, the integration of complex systems comprising numerous and diverse components onto a single chip is leading to a significant increase in the volume and diversity of system-level on-chip communication traffic. Unfortunately, in nanometer technologies, the global interconnect, which in large part constitutes the on-chip communication infrastructure, appears increasingly performance-limited compared to the components that are connected to it [102].

For high-performance designs, it is therefore crucial to ensure that the communication architecture is customized to best suit the characteristics of the traffic gen-

erated by the application. However, as shown in this chapter, communication traffic characteristics can exhibit significant *dynamic variation* depending on the specific application task being processed at a given time, the subset of SoC components that are involved in executing the task, and the run-time inputs. Furthermore, different applications may be mapped to the same SoC platform, leading to the execution of entirely different applications at different times, which in turn could lead to a wide variation in traffic characteristics. Therefore, configurable communication architectures that can be effectively customized to the application(s) requirements at run-time are desirable.

Most state-of-the-art communication architectures provide limited customization opportunities through a *static* (design time) configuration of architectural parameters, and as such, lack the flexibility to provide high performance in cases where the traffic characteristics exhibit dynamic variation. Provisioning for such dynamic flexibility and exploiting it is the focus of this chapter.

III.A.1 Chapter Overview

In this chapter, we describe FLEXBUS, a flexible, bus-based on-chip communication architecture featuring a *dynamically configurable topology*. FLEXBUS is designed to be able to detect run-time variations in system-wide communication traffic characteristics, and efficiently adapt the *logical connectivity* of the communication architecture and the components connected to it. This is achieved using two novel techniques, *bridge by-pass* and *component re-mapping*, which address configurability at the system and component levels, respectively. Bridge by-pass provides flexibility in dynamically choosing the number of bus segments that constitute the communication architecture, while component re-mapping provides flexibility in determining the manner in which components are connected to the communication architecture. These techniques provide opportunities for dynamically optimizing the communication architecture topology, a capability, which if properly exploited, can yield substantial performance gains. The FLEXBUS architecture is compatible with different existing bus standards. We describe the implementation of FLEXBUS based on AMBA AHB [22], a popular commercial on-

chip bus standard. We also present dynamic platform management policies for choosing optimized FLEXBUS configurations under time-varying traffic characteristics. We describe how the proposed techniques scale with increasing system complexity in terms of configurability of the topology, and the run-time configuration policies. We have conducted detailed experiments on FLEXBUS using a commercial design flow to analyze its area, timing, and performance under a wide variety of system-level traffic characteristics, and have compared it to conventional communication architectures. We have also analyzed its impact on the performance of two example SoC designs, (i) an IEEE 802.11 MAC processor and (ii) a UMTS Turbo decoder, and have compared the results with those obtained using conventional communication architectures. In our studies, we found that FLEXBUS provides up to 31.5% performance gains for the MAC processor, and up to 34.33% performance gains for the Turbo decoder compared to conventional architectures, with negligible hardware overhead.

The rest of this chapter is organized as follows. In Section III.A.2, we describe related work. In Section III.B, we define relevant terminology used in the context of bus-based architectures. In Section III.C, we illustrate using examples, the deficiencies of conventional architectures, and the advantages of the FLEXBUS approach. Section III.D presents details of the FLEXBUS architecture, and Section III.E discusses run-time techniques for exploiting the dynamic configurability provided by FLEXBUS. In Section III.F, we discuss how the concepts behind FLEXBUS can be scaled to arbitrarily complex system architectures. Finally, Section III.G reports on the experimental studies conducted to analyze the FLEXBUS architecture, and Section III.H concludes this chapter.

III.A.2 Related Work

The increasing importance of on-chip communication has resulted in numerous advances in communication architecture topology and protocol design, in both industry and academia. Numerous competing commercial communication architectures are in use today (*e.g.,* [23, 22, 103]). Recently, architectures based on more complex

topologies have been proposed (*e.g.*, [104, 105, 106, 107]). However, these architectures are based on fixed topologies, and therefore, not optimized for traffic with time-varying characteristics.

Dynamic configurability of the communication protocol is available in several commercial on-chip communication architectures, as described in Section II.B.4. Sophisticated protocols have also been proposed for improved sharing of on-chip communication bandwidth [108, 109, 110]. Techniques for customizing communication protocols, both statically and dynamically, to adapt to traffic characteristics, have also been studied [91, 111]. Protocol customization and topology customization are complementary, and hence, may be combined to yield large performance gains.

A number of automatic approaches have been proposed to statically optimize the communication architecture topology [112, 113, 114]. While many of these techniques aim at exploiting application characteristics, they do not adequately address dynamic variations in the communication traffic characteristics. In order to exploit such dynamic variations, adaptive routing protocols has been proposed in the context of network-on-chip based designs [115]. Our techniques are aimed at bus-based communication architectures, and focus on topology adaptation.

Transaction-level modeling and automated model refinement help raise the level of abstraction at which communication architectures are designed [116, 117]. Latency insensitive design techniques help guarantee correct system execution under variable communication delay, and hence, reduce verification effort for systems based on complex communication architectures [118]. Recent initiatives to standardize the interfaces of system components [119, 120] facilitate the customization of the communication architecture without requiring changes to the system components themselves.

III.B Background

Communication architecture *topologies* can range from a single shared bus, to which all the system components are connected, to a network of bus segments in-

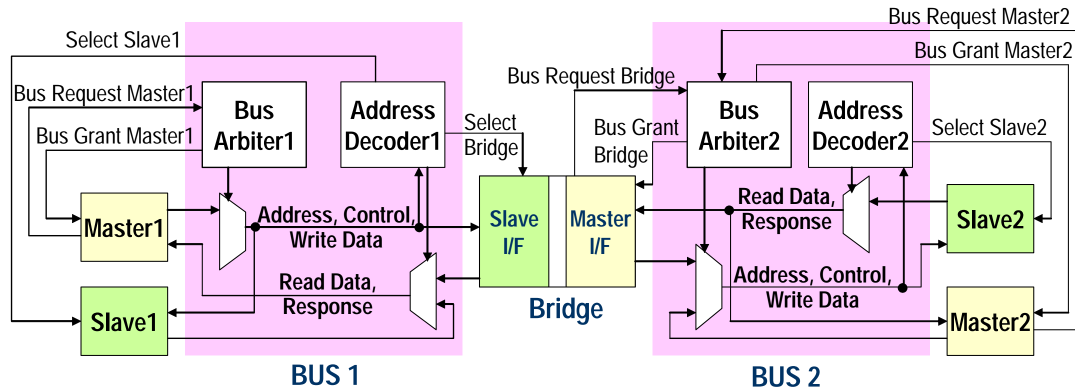


Figure III.1: Example bus-based communication architecture with two bus segments

terconnected by *bridges*. *Component mapping* refers to the association between system components and bus segments. Components mapped to the communication architecture can be either *masters* (*e.g.*, CPUs, DSPs), which can initiate communication transactions (reads/writes), or *slaves* (*e.g.,* memories, peripherals), which can only respond to transactions initiated by a master. Figure III.1 illustrates an example bus architecture consisting of two bus segments, each with one master and one slave, connected via a bridge. The internal logic of a bus segment typically comprises (i) one or more *multiplexers* for the proper routing of read and write data, and control signals between masters and slaves, (ii) an *address decoder* for selecting the slave that corresponds to a read/write transaction, and (iii) a *bus arbiter* for determining which master should be granted access to the bus and for how many cycles. *Communication protocols* specify conventions for the data transfer, such as arbitration policies, burst transfer modes, *etc*. *Bridges* are specialized components that facilitate transactions between masters and slaves located on different bus segments. "Cross-bridge" transactions execute as follows: the transaction request from the master, once granted by the first bus, is registered by the bridge's slave interface (Figure III.1). The bridge then forwards the transaction to it's master interface, which then requests access to the second bus. On being granted, the bridge executes the transaction with the destination slave and then returns the response to it's slave interface, which finally returns the response to the original master.

III.C Motivation

In this section, we analyze the shortcomings of conventional communication architectures in which the topology is configured statically, using an IEEE 802.11 MAC processor design as an example. We next describe the advantages of the FLEXBUS architecture, illustrating the importance of considering the configurability of the communication architecture topology at both the system and component levels.

III.C.1 Case Study: IEEE 802.11 MAC Processor

The functional specification of the IEEE 802.11 MAC processor system consists of a set of communicating tasks, shown in Figure III.2(a) (details are available in [121]). For outgoing frames, the LLC task receives frames from the Logical Link Control Layer, and stores them in the system memory. The Wired Equivalent Privacy (WEP) task encrypts frame data. The Integrity Checksum Vector (ICV) task works in conjunction with the WEP task in order to compute a checksum over the payload. The HDR task generates the MAC header. The Frame Check Sequence (FCS) task computes a CRC-32 checksum over the encrypted frame and header. MAC_CTRL implements the CSMA/CA algorithm, determines transmit times for frames, and signals the Physical Layer Interface (PLI) task to transmit the encrypted frames. The Temporal Key Integrity Protocol (TKIP), if enabled, generates a sequence of encryption keys dynamically. If TKIP is disabled, the key is statically configured by the network administrator.

Figure III.2(b) shows the set of components to which the above tasks are mapped in our design. An embedded processor (the ARM946E-S [122]) implements the MAC_CTRL, HDR, and TKIP tasks, while dedicated hardware units implement the LLC, WEP, FCS, and PLI tasks. Other components include frame buffers for storing MAC frames, a key buffer for secure key storage, and the communication architecture. In the following sub-sections, we consider in turn, the use of two conventional communication architectures, and the proposed FLEXBUS architecture, for this design. In order to simplify the discussion, we focus on only the tasks that result in the majority of

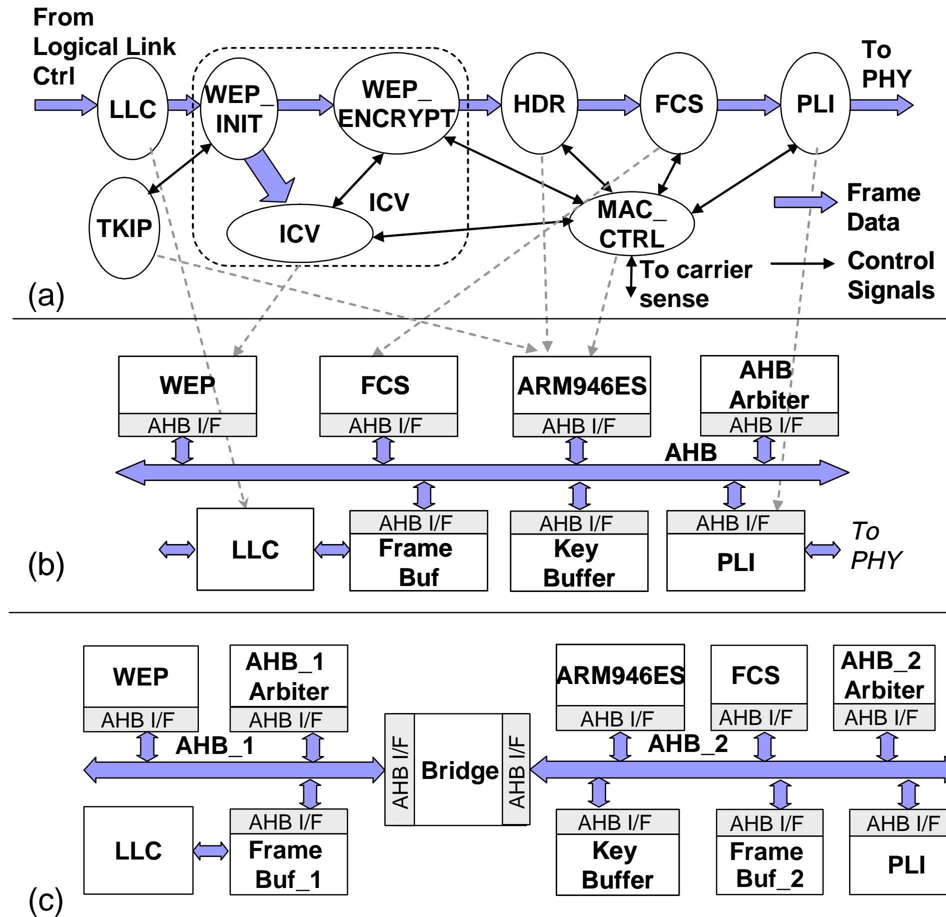


Figure III.2: IEEE 802.11 MAC processor: (a) functional specification, (b) mapping to a single shared bus, (c) mapping to a multiple bus architecture

the communication traffic: WEP, FCS, and TKIP. The system was implemented using an instruction set model for the processor and RTL Verilog for the remaining hardware, and simulated using ModelSim [123] (details of this framework are described in Section III.G.1).

III.C.2 Statically Configured Topologies

Example 1: The first architecture we consider is illustrated in Figure III.2(b), where a single AMBA AHB bus segment [22] integrates all the system components. The MAC frames are stored in a shared memory (`Frame_Buf`), and are processed in

a pipelined manner: the WEP component encrypts a MAC frame, and then signals the FCS component to start computing a checksum on the encrypted frame, while it starts encrypting the next frame. We first consider that the keys are statically configured (*i.e.*, the TKIP task is disabled). In such a scenario, the on-chip communication traffic is largely due to the WEP and FCS components. Figure III.3(a) presents a symbolic representation of an illustrative portion of the system execution traces. The figure reveals that under this architecture, at various times during system execution, simultaneous attempts by the WEP and FCS hardware to access the system bus lead to a large number of bus conflicts, resulting in significant performance loss. Experiments indicate that the maximum data rate that this architecture can support is 188 Mbps. When the TKIP task is enabled (for dynamic keys), additional traffic between the processor and the key buffer further degrades the data rate to 158 Mbps. ■

Clearly, the single shared bus topology fails to provide high performance when there are simultaneous access attempts from different masters. In particular, it fails to exploit parallelism in the communication transactions, a drawback that can be addressed by using an architecture that uses multiple bus segments.

Example 2: Figure III.2(c) presents a version of the MAC processor implemented using a topology consisting of two AHB bus segments connected by a bridge. The WEP component reads frame data from memory `Frame_Buf1`, encrypts it, and then transfers the encrypted frame into `Frame_Buf2`. The FCS component processes the frame from `Frame_Buf2`, while WEP starts processing the next frame from `Frame_Buf1`. Figure III.3(b) illustrates the execution trace under this architecture for statically configured keys. We observe that the parallelism offered by the multiple bus architecture enables the FCS and WEP tasks to process frame data stored in their local frame buffers concurrently. However, we also observe additional latencies in certain intervals (indicated by shading) where a majority of transactions need to go across the bridge, due to the complex nature of cross-bridge transactions (described in Section III.B). Experiments indicate that the data rate achieved by this architecture is

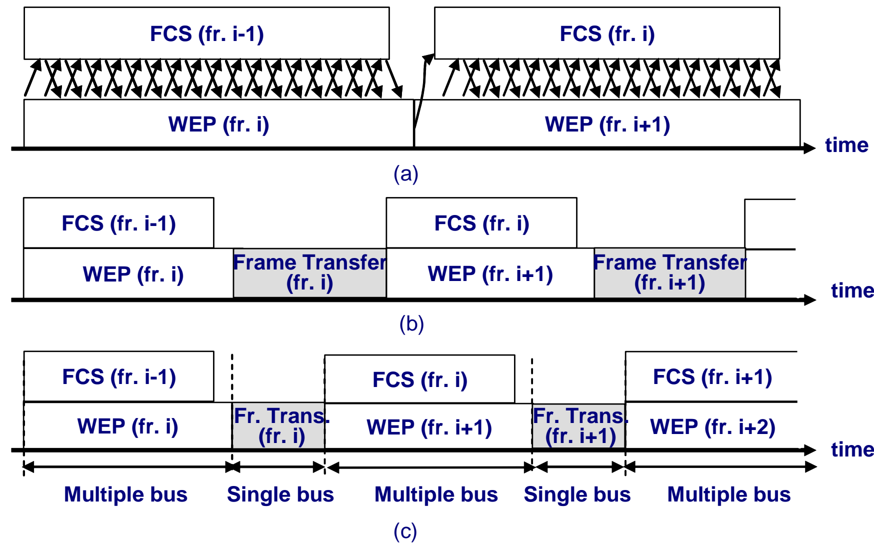


Figure III.3: Execution of the IEEE 802.11 MAC processor under (a) single shared bus, (b) multiple-bus, (c) FLEXBUS

201 Mbps, only a 7% improvement over the single bus. When the TKIP task is enabled, the achieved data rate is 176 Mbps, a 11% improvement over the single bus. ∎

This example illustrates that the advantage of a particular communication architecture topology depends on the characteristics of the communication traffic: when the proportion of cross-bridge traffic is low, the multiple bus architecture performs well, whereas at other times, the single shared bus architecture is superior. These examples also illustrate how during the course of execution of an application, the characteristics of the communication traffic can change significantly over time, based on the set of concurrently executing tasks, and their inter-component communication requirements. Further, communication architectures based on fixed topologies are not capable of detecting and adapting to such changes, and hence, often yield sub-optimal performance.

III.C.3 Dynamic Topology Configuration

We next consider the execution of the IEEE 802.11 MAC processor under two variants of the FLEXBUS architecture.

Example 3: We first consider the case where the encryption keys are statically configured (the TKIP task is disabled). The execution trace under FLEXBUS is illustrated in Figure III.3(c). The trace illustrates that the bus architecture operates in a multiple bus mode during intervals that exhibit localized communications, and hence enables concurrent processing of the WEP and FCS tasks. In intervals that require low latency communication between components located on different bus segments, a *dynamic bridge by-pass* mechanism is used. Under this technique, the two bus segments of the multiple bus topology are temporarily fused together into a single shared bus. The measured data rate under this architecture was found to be 248 Mbps, a 23% improvement over the best conventional architecture. ∎

The above example illustrates that by adapting the bus topology to traffic characteristics, the benefits of shared and multiple bus architectures can be combined. Note that, the bridge by-pass mechanism provides a technique to make *coarse-grained* (system-level) changes to the communication architecture topology. However, at times, the ability to make more *fine-grained* (component-level) changes to the topology is also important, as illustrated next.

Example 4: We consider the case in which the TKIP task is enabled. For the FLEXBUS architecture of Example 3, the resulting increase in bus traffic on AHB_2 (Figure III.2(c)) causes the achieved data rate to decrease to 208 Mbps, although it still out-performs the best conventional architecture by 18%. We next consider the execution of another version of the FLEXBUS architecture featuring a *dynamic component re-mapping* capability. Using this technique, while the overall architecture remains in the multiple bus configuration, the mapping of the slave `Frame_Buf2` is dynamically switched between the two buses at specific times. In particular, it is mapped to AHB_2 as long as the FCS and WEP components are processing frame data, and to AHB_1 at times when the most recently encrypted frame needs to be efficiently transferred from `Frame_Buf1` by the WEP task. By preserving the multiple bus topology, and thereby enabling concurrent operation of the frame transfer and the TKIP tasks, this architec-

ture achieves a data rate of 224 Mbps, a 27% improvement over the best conventional architecture. ■

This example shows that at times, exploiting local variations in traffic characteristics through component-level changes can provide additional performance benefits.

In summary, these illustrations establish that by recognizing dynamic variations in the spatial distribution of communication transactions, and correspondingly adapting the communication architecture topology (both at the system and component levels), large performance gains can be achieved. In the next three sections, we describe how these opportunities are exploited by the FLEXBUS architecture.

III.D FLEXBUS Architecture

In this section, we first provide a brief overview of the FLEXBUS architecture and its design goals. Next, we present a detailed description of the key techniques that underlie the FLEXBUS architecture in the context of a two-segment AMBA AHB based bus architecture. The extension of FLEXBUS to more complex communication architectures is discussed in Section III.F.

III.D.1 Overview

The FLEXBUS architecture features a dynamically configurable communication architecture topology. The techniques underlying FLEXBUS are independent of specific communication protocols, and hence can be applied to a variety of on-chip communication architectures. In our work, we demonstrate its application to the AMBA AHB [22], a popular commercial on-chip bus. FLEXBUS provides applications with opportunities for dynamic topology customization at the system level, using techniques that enable run-time fusing and splitting of bus segments. This is achieved using dynamic bridge by-pass, details of which are described in Section III.D.2. FLEXBUS also provides customization opportunities at the component level in order to exploit local variations in traffic characteristics, by using techniques that allow components to be dy-

namically switched from one bus segment to another. This is achieved using component re-mapping, details of which are described in Section III.D.3.

Numerous technical challenges need to be met in order to provide such configurability. The particular goals that were kept in mind during the design of FLEXBUS include the following:

- maintaining compatibility with existing on-chip bus standards for efficient deployment
- minimizing timing impact to enable high speed operation
- minimizing logic and wiring complexity (hardware overhead)
- providing low reconfiguration penalty to maximize the gains achieved through flexibility

The rest of this section provides details on how FLEXBUS provisions for dynamic configurability keeping the above goals in mind.

III.D.2 Coarse-Grained Topology Control: Bridge By-Pass Mechanism

Figure III.4 illustrates the hardware required to support dynamic bridge by-pass for an example system consisting of two AMBA AHB bus segments, connected by a bridge. AHB1, the primary bus, has two masters (M1 and M2) and one slave (S1), while AHB2, the secondary bus, has one master (M3) and one slave (S2). Each bus segment contains an Arbiter, an Address Decoder, and multiplexers for routing the granted master's address values, control signals and write data to the slaves, and for routing the selected slave's ready signal, response signals and read data back to the masters. The bridge enables transactions between masters on AHB1 and slaves on AHB2.

Hardware Enhancements

The system can be operated in (i) a multiple bus mode or (ii) a single shared bus mode, by disabling or enabling bridge by-pass, respectively, via the *config_select*

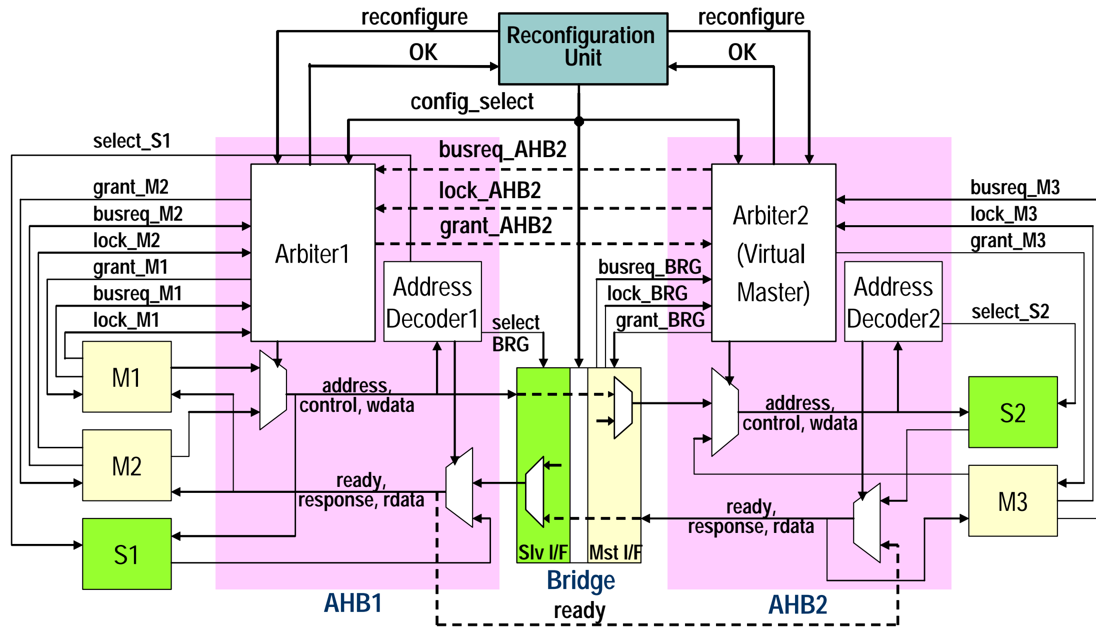


Figure III.4: Dynamic bridge by-pass capability in FLEXBUS

signal, which is an output of the "Reconfiguration Unit" module. In the multiple bus mode (*config_select* = *0*), the signals shown by the dotted arrows are inactive. The two bus segments operate concurrently, with each arbiter resolving conflicts among the masters in its own bus segment. Transactions between masters on AHB1 and slaves on AHB2 go through the bridge using the conventions described in Section III.B. In the single shared bus mode (*config_select* = *1*), the signals shown by the dotted arrow are active, and the bridge is "by-passed", thereby fusing the two bus segments together. We next describe the enhancements required to the basic hardware architecture of the two segment AMBA AHB bus to support these two operating modes.

Bridge: To by-pass the bridge in the single shared bus mode, the inputs to the bridge master and slave interfaces are directly routed to the outputs, by-passing the internal bridge logic (using multiplexers). This allows transaction requests from masters on AHB1 to slaves on AHB2 (and the corresponding slave responses) to reach within one clock cycle. Note that, while we illustrate bridge by-pass for a one-way bridge, the technique can be applied to two-way bridges as well.

Arbiters: In the single bus mode, only one master can be granted access to the FLEXBUS fabric at any given time, whereas in the multiple bus mode, more than one master may have transactions executing in parallel. Clearly, the arbitration policies of the multiple bus mode need to be adapted for the single bus mode. A naive solution would be to designate one of the arbiters as a centralized arbiter for the single bus mode. However, this would require the centralized arbiter to be connected to the *busreq*, *lock* and *grant* signals of all the system masters, resulting in large wiring overhead, and potentially large arbitration latencies. Instead, we opt for a *distributed arbitration mechanism*, in which one of the arbiters (Arbiter2 in Figure III.4) behaves as a *virtual master* that is regulated by the other arbiter (Arbiter1). On receiving one or more bus requests from masters on AHB2, Arbiter2 immediately sends a bus request to Arbiter1 using the *busreq_AHB2* and *lock_AHB2* signals, which are generated by a bitwise *OR* of the bus request and lock signals of all the masters on AHB2. Arbiter1 arbitrates among the received bus requests from AHB1 masters as well as the virtual master, which, in effect, represents all the masters on AHB2. In parallel, in order to reduce arbitration latency, Arbiter2 arbitrates among its received bus requests. However, Arbiter2 grants the bus to the selected master only when it receives a grant (via the *grant_AHB2* signal) from Arbiter1. The grants for masters on AHB2 are generated by a pairwise *AND* of the *grant_AHB2* signal with the grant signals of Arbiter2. This guarantees that only one master is granted access to FLEXBUS when in the single bus mode.

Note that, in the above distributed arbitration scheme, since Arbiter1 receives a single bus request (*busreq_AHB2*) on behalf of all the masters on AHB2, the granularity at which arbitration is performed could suffer. However, in practice, this is acceptable, since bus hierarchies impose similar restrictions (*i.e.,* bridges behave as "agents" for any of the masters on AHB1 requesting access to a remote slave on AHB2). Also, in the case of the AMBA AHB bus protocol, the *ready* bus signal indicates the state of the bus to all the components. In order to ensure correct operation of the system in the single bus mode, all the components should observe the same *ready* signal. This is achieved by routing the *ready* signal of AHB1 to AHB2. Note that, the combinational loop between

the multiplexers shown in Figure III.4 is a false loop, *i.e.*, it is never enabled during operation.

Address Decoders: The address decoders on the two bus segments require no change, since the address maps of the slaves and their mapping to the bus segments does not change under dynamic bridge by-pass.

Reconfiguration Unit: The Reconfiguration Unit (Figure III.4) is a new hardware component that is introduced to enable dynamic topology configuration. It is responsible for selecting the bus configuration at run-time, and for ensuring correctness of system operation while switching between the two configurations. It can either make configuration decisions automatically (using policies such as described in Section III.E), or be directed by a higher level policy, such as Dynamic Platform Management (described in Chapter V), to change the configuration. To apply a new configuration, it first asserts the *reconfigure* signal to the arbiters. On receiving this signal, the arbiters terminate the current transaction (unless the master has acquired a lock on the bus), deassert all grant signals, and assert the *OK* signal. On receiving the *OK* signal from both the arbiters, the Reconfiguration Unit toggles the *config_select* signal. The exact overhead of reconfiguration depends on the precise set of pending bus transactions. In our design, the worst case overhead of bus reconfiguration for the two AHB segment AMBA based system was observed to be 17 cycles (assuming the bus is not locked and all slaves have single cycle response). This overhead should be taken into account while making bus configuration decisions (described in Section III.E).

Delay Impact

The addition of logic and wiring for dynamic bridge by-pass results in a slight increase in the critical path delay of the bus. The true critical path in the multiple bus mode is shorter than that in the single bus mode, since many long paths of the single bus mode are false paths in the multiple bus mode (shown by the dotted arrows in Figure III.4). However, as borne out by experiments presented in Section III.G, the delay

penalty of each mode compared to the corresponding static architectures is small, and is more than compensated for by the performance improvements achieved by exploiting the flexibility of the architecture.

When the worst case delays of the single bus mode and the multiple bus mode are comparable, it is feasible to always operate FLEXBUS at a single frequency determined by the larger of the two delays. However, in some cases, the delay in the single bus mode maybe much larger than that in the multiple bus mode. Also, some multiple bus systems operate different bus segments at different frequencies in order to support high performance components such as CPUs on one bus, and low performance peripherals on a secondary bus. In such scenarios, the FLEXBUS frequency needs to be adapted, based on its current configuration. Using a programmable PLL to achieve this would lead to a high reconfiguration penalty (hundreds of μsecs). However, we observe that since we only need to switch between two clock frequencies, two PLLs in conjunction with a dynamic clock source switching circuit [124] could be used in such a scenario for efficient frequency adaptation.

III.D.3 Fine-Grained Topology Control: Component Re-Mapping Mechanism

Figure III.5 shows a two segment AMBA AHB bus architecture, which implements a dynamic re-mapping capability for master M2 and slave S2. Dynamic re-mapping allows the mapping of each of these components to be dynamically switched between AHB1 and AHB2.

Hardware Enhancements

The mapping of M2 and S2 is selected by the signals, *config_select_M2* and *config_select_S2*, respectively, which are generated by the Reconfiguration Unit. The hardware enhancements that are required to the AMBA AHB bus architecture to enable this are described next.

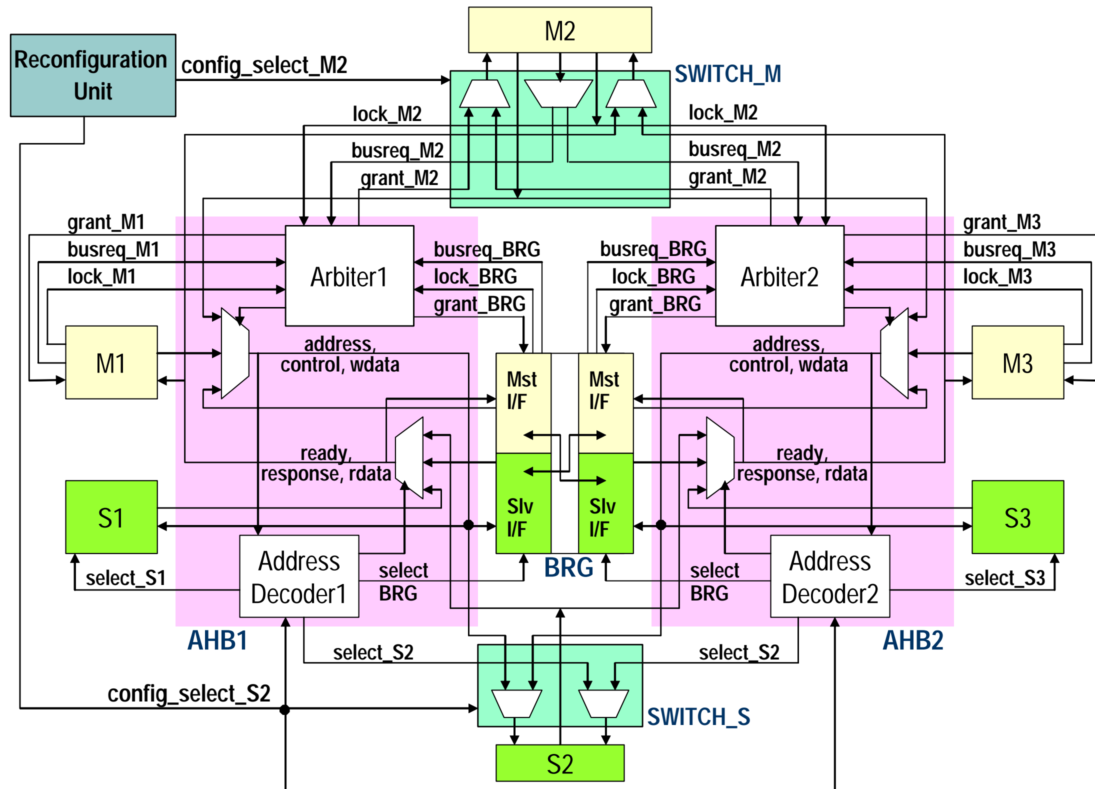


Figure III.5: Dynamic component re-mapping capability in FLEXBUS

Switches: The signals of a re-mappable master or slave are physically routed to both AHB1 and AHB2. However, the switch boxes, SWITCH_M and SWITCH_S, activate the signals to and from only one of the bus segments using multiplexers, depending on the configuration. Note that, only a subset of the master and slave signals require to be switched in the AMBA AHB protocol.

Bridge: The bridge does not require any changes to enable component re-mapping, since the multiple bus structure of the communication architecture is preserved. However, note that, component re-mapping can lead to error responses in the case of one-way bridges. For example, assuming that BRG in Figure III.5 is one-way (with AHB1 being the primary bus and AHB2 the secondary bus), and that M2 is mapped to AHB2, then if M2 generates a request for S1, it would receive an error response. Therefore, in the presence of masters that cannot handle error responses, it can be safely applied only in the case of two-way bridges.

Arbiters: The arbiters on the two bus segments need to be designed to arbitrate amongst all the masters that can potentially be mapped to their respective bus segment. No other changes are required, since master bus requests are only sent to the arbiter on the bus to which they are currently mapped.

Address Decoders: To enable dynamic slave re-mapping, reprogrammable address decoders are required. Depending on the mapping of the slaves, the address decoders on the two bus segments are reconfigured to generate the correct slave select signal. For example, when S2 is mapped to AHB1, on observing an address belonging to S2 on the address bus, Decoder1 asserts the *select_S2* signal, while Decoder2 asserts the *select_BRG* signal. The situation is reversed when S2 is mapped to AHB2. This is done by switching the mapping of the address space of S2 between BRG and S2 in the decoder, depending on the *config_select_S2* signal.

Reconfiguration Unit: The Reconfiguration Unit is responsible for selecting the mapping of the re-mappable masters and slaves (using policies such as described in Section III.E), and for generating the appropriate configuration select signals. It can also be directed by a higher level platform configuration policy (such as Dynamic Platform Management) to apply a specific mapping. In order to ensure correct operation of the system during re-mapping, the Reconfiguration Unit monitors the master's *busreq_M2* and slave's *select_S2* signals to determine if they are currently active on the bus, and if not, the corresponding *config_select* signal is toggled. The rest of the bus continues operating without interruption.

Delay Impact

The extra logic and wiring required to enable component re-mapping may lead to a slight increase in the critical path delay of the bus. However, this should be compensated for by the performance improvements achieved through adaptation. If the two bus segments operate at different clock frequencies, then dynamic component re-mapping can only be applied to masters and slaves which are capable of operating at

both clock frequencies. In this case, re-mapping also involves changing the clock source of the re-mapped component using techniques such as described in [124].

III.E Dynamic Configuration Policies for FLEXBUS

In this section, we describe run-time policies for adapting the FLEXBUS configuration based on changes in the characteristics of the on-chip communication traffic. We discuss these policies in the context of two segment buses, considering in turn, bridge by-pass and component re-mapping. Extension of these policies to more general communication architecture topologies is discussed in the next section.

The problem of dynamically choosing the optimum configuration of the FLEXBUS architecture based on an observation of the characteristics of the communication traffic can be addressed using several approaches. In the past, stochastic control policies have been proposed to address a similar problem in the domain of dynamic power management, where the optimum power state of components (active, idle) needs to be selected for dynamically varying system workloads [125]. Numerous heuristic approaches that attempt to predict future behavior based on an observed history of the workload have also been proposed to address the same problem [126]. In our work, we examine history-based techniques in further detail.

Let us consider a FLEXBUS system featuring dynamic bridge by-pass between two bus segments, *BUS1* and *BUS2*. Let N_{BUS1}, N_{BUS2} and N_{BRG} represent the number of local transactions on *BUS1*, number of local transactions on *BUS2* and number of transactions between the two bus segments, respectively, during an observation interval. A transaction refers to a single bus access (*e.g.*, a burst of 5 beats constitutes 5 transactions). The time taken to process this traffic under the single bus mode, T_{Single}, is given by:

$$T_{Single} = (N_{BUS1} + N_{BUS2} + N_{BRG}) \times C_L \times t_{SB} \qquad \text{(III.1)}$$

where C_L is the average number of cycles for a local bus transaction, and t_{SB} is the clock

period in the single bus mode, since all transactions are on the same bus. Similarly, the time taken under the multiple bus mode, $T_{Multiple}$, is approximated by:

$$T_{Multiple} = max(N_{BUS1}, N_{BUS2}) \times C_L \times t_{MB} + N_{BRG} \times C_B \times t_{MB} \qquad \text{(III.2)}$$

where C_B is the average number of cycles for a cross-bridge transaction, and t_{MB} is the clock time period in the multiple bus mode. If $T_{Single} < T_{Multiple}$, then the single bus mode is preferred, else the multiple bus mode is better. Each bus segment is enhanced with extra logic to observe and record the number of bus transactions of each type at run-time over a time period TP. At the end of the time period, the reconfiguration unit reads these values and selects the new configuration based on the above criterion.

The choice of an appropriate configuration time period, TP, is crucial. Smaller time periods enable the policy to be more responsive to variations in the traffic characteristics. However, if the traffic characteristics change rapidly, this might lead to excessive oscillations between the configurations, thus potentially degrading the performance due to the reconfiguration overhead. Therefore, in our policy we use an adaptive time period, TP, which is selected as follows. Let C denote the number of times the bus was reconfigured over the last τ clock cycles. If $C/\tau > \lambda_1$, then the time period, TP, is doubled, if $C/\tau < \lambda_2$, then TP is halved, else it is unchanged. λ_1 and λ_2 represent two thresholds, and depend on the reconfiguration overhead. In our experiments with an example eight master and eight slave system (described in Section III.G.1), we observed an average reconfiguration overhead of 10 cycles, for which $\tau = 250$ cycles, $\lambda_1 = 0.0025$, and $\lambda_2 = 0.001$ proved effective. We conclude that, in general, these parameters should be carefully set, based on an analysis of the traffic characteristics of the application.

Let us next consider a FLEXBUS architecture consisting of two bus segments, *BUS1* and *BUS2*, connected by a bridge, and with some re-mappable master and slave components. The problem of dynamic component re-mapping is to select at run-time the mapping of the re-mappable components to either *BUS1* or *BUS2*. For this, we propose a history-based policy, in which for each re-mappable master (or slave), the

number of transactions to (or from) components on either bus segment is monitored over an observation time period, based on which the configuration for the next time period is selected. The optimal mapping of components to buses is one that minimizes the number of transactions across the bridge (to reduce bridge overhead), while balancing the number of transactions on the two bus segments (to have concurrent operation). This problem maps to the graph bisection problem, which is NP-complete [127]. Therefore, we make use of the following simple strategy. For each re-mappable component, the difference between its number of cross-bridge and local transactions is monitored. The component for which this difference is positive and maximum is selected to be re-mapped. The time period over which transactions are monitored is adapted as in the bridge by-pass approach.

III.F Scalability of the FLEXBUS Approach

In this section, we discuss how the architectural mechanisms underlying FLEXBUS, for both bridge by-pass and component re-mapping, can be extended to complex communication architectures consisting of arbitrary organizations of bus segments and bridges. We then formulate the problem of how to optimize the deployment of FLEXBUS and select optimized FLEXBUS configurations at run-time.

III.F.1 Dynamic Multi-Bridge By-Pass

Complex communication architectures may consist of numerous bus segments connected by multiple bridges. It may be necessary, at certain times, to fuse multiple (more than two) bus segments into a single shared bus, depending on the traffic characteristics. In order to achieve this, all the bridges that integrate these bus segments must be made by-passable as described in Section III.D.2. In doing so, the architecture needs to ensure that only one master gets access to the fused shared bus at any point of time. The distributed arbitration mechanism described for one bridge (Section III.D.2) can be extended to multiple bridges as follows. When the architecture is in the single

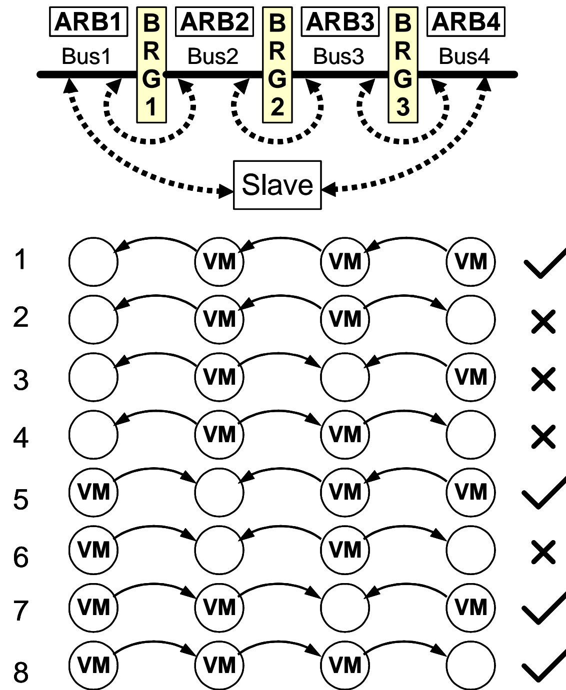


Figure III.6: Example system with four bus segments, and possible assignment of arbiters as virtual masters

bus mode, the arbiters co-ordinate by acting as virtual masters and forwarding their bus requests to adjacent arbiters. The decision of which arbiters to designate as virtual masters and where to forward their bus requests is performed statically, and is crucial to ensure correct operation of the system. Consider the example system in Figure III.6 that consists of four bus segments and three bridges, where as many as all three bridges can be by-passed simultaneously at run-time. The figure also shows different assignments of arbiters as virtual masters (circles marked as *VM*) and their dependency due to bus request signals on their adjacent arbiters (directed arrows). An arbiter behaves as a virtual master and forwards bus requests to an adjacent arbiter only when its bus is fused with a neighboring bus. The figure illustrates the feasibility of different mappings of virtual masters to arbiters. For a N segment bus, $N-1$ arbiters must be designated as virtual masters, each with a dependency on another arbiter, in order to prevent independent arbitration decisions, which can cause erroneous bus operation. Numerous feasible solutions are possible (*e.g.,* 1,5,8) differing in terms of the arbiters selected as virtual

masters and their dependency edges. The best choice among the feasible solutions is one that results in the minimum wiring overhead and delay penalty. An additional criterion is that there should be no cycles in the dependencies of the arbiters to prevent deadlock. The same conclusions hold for any arbitrary connection of the bus segments (*e.g.*, hierarchical bus).

III.F.2 Scalability of Component Re-Mapping

In complex communication architectures it may be necessary to provide support for dynamically re-mapping a component, either a master or a slave, to multiple (greater than two) bus segments. In order to provide this support, the basic approach described in Section III.D.3 is extended as follows. The signals of the re-mappable component are physically connected to all the bus segments to which it can be potentially mapped. An appropriate switch is used to ensure that only one set of signals is active at any point of time. For slave re-mapping, the address decoders on each bus segment are configured to generate the correct slave select signals by reprogramming the address maps. This needs to be done not only for the address decoders on the bus segments to which the slave can be mapped, but also for intermediate address decoders. For example, in Figure III.6, suppose a slave is re-mappable to either *Bus1* or *Bus4*. Now, when it is mapped to *Bus1*, on observing an address belonging to the slave, the address decoders on *Bus1*, *Bus2*, *Bus3* and *Bus4* should select the slave, *BRG1*, *BRG2* and *BRG3*, respectively, while when it is mapped to *Bus4*, the address decoders should select *BRG1*, *BRG2*, *BRG3* and the slave, respectively.

The two FLEXBUS mechanisms, bridge by-pass and component re-mapping, can be applied together to provide maximum performance benefits through adaptation.

III.F.3 Applying FLEXBUS in Complex Communication Architectures

In order to provide maximum flexibility during run-time configuration, all the bridges in the system can be made by-passable, and all the master and slave components can be made re-mappable to all bus segments using the techniques described

in Sections III.F.1 and III.F.2. However, in practice, this would lead to unacceptably high overheads, potentially negating any performance improvements through adaptation. This is because as more and more bus segments are fused together, the associated delay penalty due to extra logic and wiring overhead increases. Similarly, as a component is made re-mappable to more and more bus segments, its performance may decrease due to the overhead of the switch and additional wiring. Also, a bus segment with many re-mappable components may incur large delay penalty due to the extra wiring and logic overhead in the bus required to provision for the additional masters and slaves. We expect that provisioning for and properly managing such flexibility in *judiciously selected* parts of the communication architecture will provide sufficient performance improvements through adaptation with acceptable overheads. Hence, we envision a two step methodology for using FLEXBUS in complex communication architectures. In the first step, the designer must select (either manually or through automatic tools) the set of desired configurability options, namely, which bridges should be augmented with by-pass support, and which components should be augmented with re-mapping support. In the second step, run-time policies must be designed to exploit the provided configurability.

Selection of Configurable Components

The goal of this step is to select the bridges and the components that should be made by-passable and re-mappable, respectively, such that the performance improvements achievable through adaptation are maximized, while not exceeding specified area and wiring constraints. We assume the availability of a statically customized communication architecture topology optimized for a partitioned and mapped system, and consider the different possible configurability options for improving the flexibility of the given architecture.

We note that both the bridge by-pass and component re-mapping techniques address the latency of transactions between components on different bus segments incurred due to the bridge overhead. For such cross-bridge transactions between any two components, the performance improvement through component re-mapping is superior

to by-passing the intermediate bridges, since (i) in component re-mapping, the multiple bus segments are preserved enabling concurrent operation, and (ii) the clock frequency of an individual bus segment would be higher than that of a long single shared bus made by fusing multiple bus segments together. However, bridge by-pass provides performance improvements for *all* transactions across the bridge, and not just for transactions between a particular pair of components. To provide such improvements using component re-mapping, more and more components would need to be made re-mappable, leading to significant hardware overhead, and large delay penalty. This trade-off should be kept in mind while selecting the by-passable bridges and re-mappable components. For example, components that have a large volume of cross-bridge transactions can be made re-mappable, while bridge by-pass can be used for providing performance improvements for other cross-bridge transactions. Also, note that, component re-mapping may not be applicable in the case of one-way bridges as it can lead to error responses.

With growing communication architecture complexity, it is clear that the number of solutions and the underlying trade-offs will render manual approaches impractical. Using fast analysis approaches such as [128, 114], automatic techniques can be developed that methodically search the space of different configurability options to choose an optimized subset that maximizes performance gains under specified area and wiring constraints.

Dynamic System-Level Configuration Policies for Complex Communication Architectures

Given an architecture with support for by-passing selected bridges and re-mapping selected components, the dynamic policy needs to optimally select the configuration of the communication architecture depending on run-time traffic characteristics. We next briefly discuss how the history-based policies described in Section III.E can be extended to more complex communication architecture topologies. In this approach, on each bus segment, the number of transactions of different types are monitored by the bus logic over a time period and stored in addressable registers. This is then read by

the reconfiguration unit (implemented as a bus master), based on which it selects the configuration for the next time period. The goal of the policy is to map the re-mappable components to the bus segment on which they have the most number of transactions, and to by-pass each bridge when the number of transactions across the bridge significantly exceeds the local traffic on each bus segment (excluding the number of transactions due to the re-mappable components). For small systems consisting of a few bus segments, the bus configuration decisions can be taken globally in a centralized manner based on the transaction histories of all the bus segments. The policy can be implemented on a centralized reconfiguration unit which communicates with all the bus segments and configurable components. However, for larger systems, this might be infeasible due to overheads in collecting transaction statistics from all the bus segments, and sending configuration signals to all the configurable components. However, assuming that the initial communication architecture topology and component mapping is one that is conscious of spatial locality, we can safely expect that there will be few transactions between components that are separated from each other by numerous bus segments. Therefore, the system can be partitioned into smaller clusters of bus segments, where each cluster has its own reconfiguration unit making localized bus configuration decisions for the cluster. More global configuration decisions can be made through co-operation between such reconfiguration units.

III.G Experimental Results

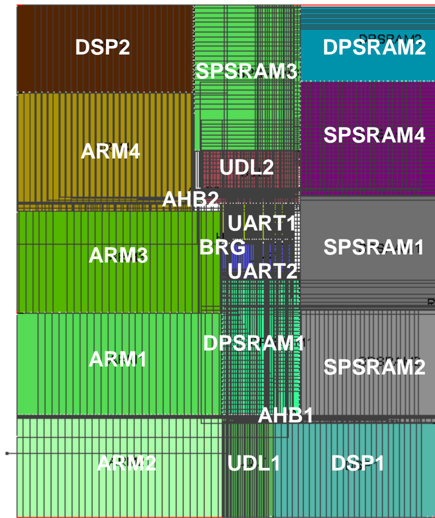
In this section, we present experimental studies that evaluate the FLEXBUS architecture. We present hardware implementation results, followed by an analysis of the FLEXBUS architecture using synthetic traffic profiles to systematically evaluate its performance under different types of traffic. We also apply FLEXBUS to the design of two system-on-chips: (a) an IEEE 802.11 MAC processor, and (b) a UMTS Turbo decoder, to evaluate its performance in the context of real applications.

III.G.1 Experimental Methodology

For the experiments with synthetic workloads, we used a system consisting of two bus segments connected by a bridge with four masters and four slaves on each bus segment, equipped with programmable VERA bus-functional models [129] for traffic generation. The IEEE 802.11 MAC processor (described in Section III.C.1) was implemented using an instruction set model for the ARM processor, and Verilog for the remaining hardware. The UMTS Turbo decoder (described in Section III.G.5) was implemented using Verilog. Reference AMBA AHB RT-level implementations of the conventional communication architectures for each system were generated using the CoreConsultant tool of the Synopsys Designware AMBA tool suite [129]. FLEXBUS was implemented by enhancing the reference AMBA AHB implementations as described in Section III.D. The Reconfiguration Unit incorporating the run-time policies described in Section III.E was implemented in Verilog. Performance analysis results were obtained through simulations using ModelSim [123]. For accurate chip-level area and delay comparison, we generated floorplans of all the systems [130]. For this, area estimates for the different components were obtained from datasheets [131] in some cases, and from synthesis using Synopsys Design Compiler [132] in cases where RTL descriptions were available, for the NEC 0.13μm technology [133]. The floorplanner was modified to report wirelengths of the global wires that constitute the different communication architectures. Global wire delay was calculated assuming delay optimal buffer insertion [134] and Metal 6 wiring. The designs were annotated with these wire delays and Synopsys Design Compiler [132] was used for delay estimation.

III.G.2 Hardware Implementation Results

The area and timing analysis methodology described above was applied to the example eight master and eight slave system under (i) FLEXBUS with dynamic bridge by-pass, (ii) single shared bus, and (iii) multiple bus architectures. Figure III.7(a) shows the floorplan of the system under the FLEXBUS architecture. The results of these studies are shown in Figure III.7(b). From the figure, we observe that the total chip area of



(a) Floorplan under the FLEXBUS architecture

Bus Architecture	Total Area (sq. mm)	Delay (ns)	Frequency (MHz)
Single Shared Bus	82.12	4.59	218
Multiple Bus	84.27	3.79	264
FLEXBUS (single bus mode)	82.66	4.72	212
FLEXBUS (multiple bus mode)		3.93	254

(b) Total chip area, bus delay and bus clock frequency under different communication architectures

Figure III.7: Hardware implementation results for example eight master and eight slave system

the system with the multiple bus architecture is 2% larger than that for FLEXBUS, since the floorplanner achieves more optimized wirelengths for the multiple bus architecture by incurring a small area penalty. The figure also presents delay results under different bus architectures. For the FLEXBUS, the critical path delay in the multiple bus mode is smaller than that in the single bus mode since many long paths of the single bus mode are false paths in the multiple bus mode. This corresponds with the observation that the static multiple bus architecture can operate at a higher frequency than the static single shared bus due to shorter wirelengths and lesser bus loading. FLEXBUS incurs a 3.2% delay penalty on average compared to the statically configured architectures, due to extra wiring and logic delay. However, this timing penalty is more than compensated for by the performance benefits of adapting the bus to changing traffic characteristics, as borne out by the following subsections.

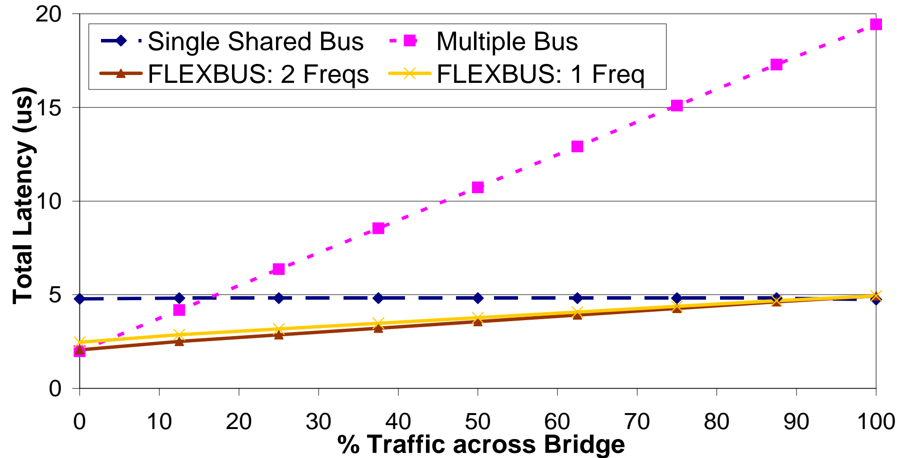


Figure III.8: Performance comparison under varying volume of cross-bridge traffic

III.G.3 Performance Under Synthetic Traffic Profiles

We performed experiments to analyze the performance of the FLEXBUS architecture compared to the statically configured architectures under a wide range of communication traffic.

Deterministic Traffic Profiles

For this experiment, we consider simple traffic profiles that consist of two phases. In the first phase, all the traffic is local *i.e.*, between masters and slaves on the same bus segment (no traffic across the bridge), while in the second phase, all the traffic is between masters on AHB1 and slaves on AHB2 (all traffic across the bridge). Different such profiles were generated by varying the relative volume of traffic in each phase. The conventional architectures were operated at the clock frequencies shown in Figure III.7(b). For FLEXBUS, we considered two cases: (i) it is always operated at 212 MHz, and (ii) the clock frequency is switched between 212 MHz and 254 MHz depending on the configuration. Figure III.8 shows the total latency (Y-axis) for the different traffic profiles (X-axis) under the different bus architectures. For the single bus, the latency remains constant since all the components are connected to the same bus. The multiple bus performs much better than the single bus when there is less traffic

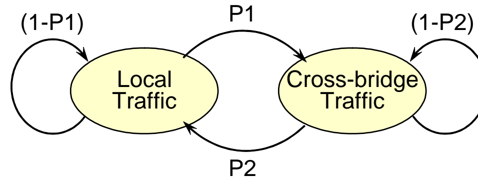
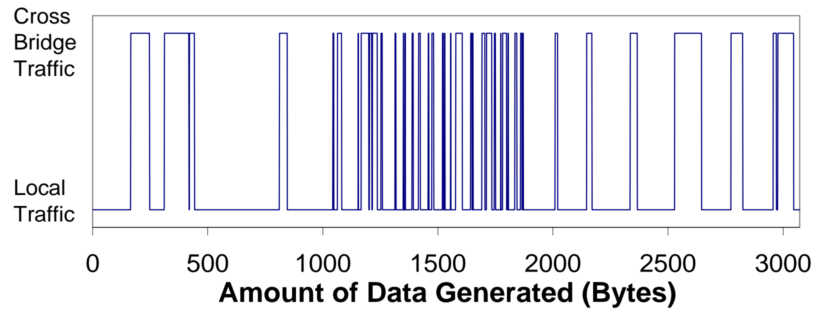


Figure III.9: Two state Markov model to systematically generate varying traffic profiles

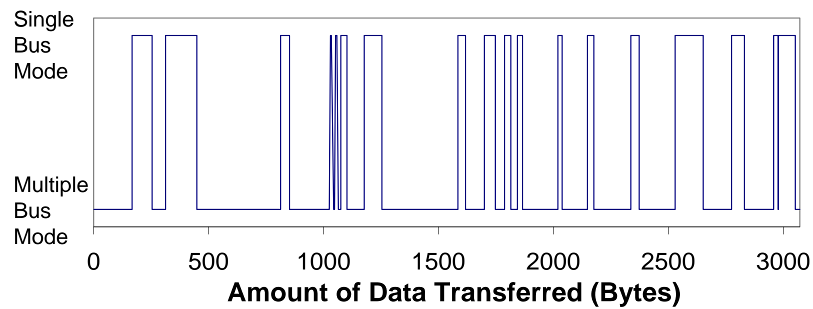
across the bridge, but its performance rapidly deteriorates with increasing cross-bridge traffic due to the large delay penalty of the bridge. Both variants of FLEXBUS achieve substantial performance gains over the conventional architectures across most of the traffic space. As expected, for purely local traffic, FLEXBUS performance is almost identical to a multiple bus architecture, while for heavy cross-bridge traffic, it is similar to the single shared bus.

Random Traffic Profiles

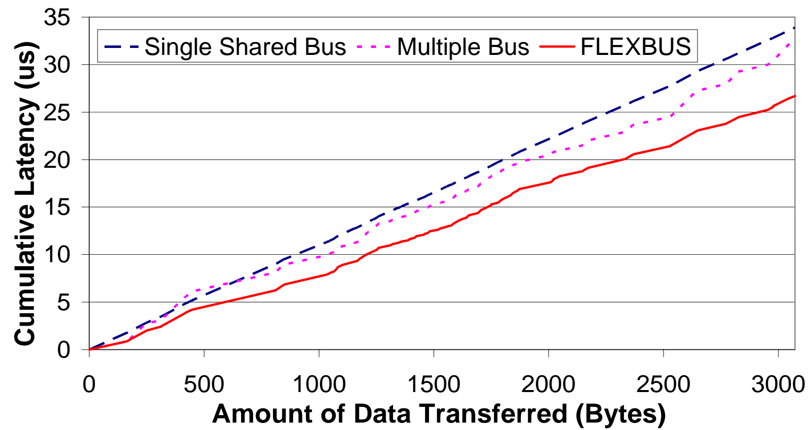
In the next experiment, the effectiveness of FLEXBUS under run-time variations in traffic characteristics is evaluated. The traffic profiles were generated using a two-state Markov model, where each state corresponds to either local traffic or cross-bridge traffic, as illustrated in Figure III.9. Varying the transition probabilities of the edges (P1 and P2) allowed us to vary the granularity with which the two traffic types are interleaved in the profile. Figure III.10(a) shows a representative traffic profile consisting of a mix of local and cross-bridge traffic. Figure III.10(b) shows the run-time configuration decisions taken by the policy described in Section III.E. Figure III.10(c) plots the cumulative latency of the different architectures for this traffic profile. We observe that the policy successfully adapts FLEXBUS to changes in the traffic characteristics, achieving significant performance benefits over the static single shared bus (21.3%) and multiple bus (17.5%) architectures. Also, the policy achieves performance benefits even under frequent variations in the traffic profile (between 1050 seconds and 1870 seconds), by increasing the configuration time period, TP, thus configuring the bus at larger time granularities.



(a) Example traffic profile



(b) Bus configurations selected by the policy



(c) Cumulative latency under different bus architectures

Figure III.10: Performance of FLEXBUS with run-time configuration policy

III.G.4 Application to an IEEE 802.11 MAC Processor

Table III.1: Performance of the IEEE 802.11 MAC processor under different communication architectures

Bus Architecture	Computation Time (ns)	Data Transfer Time (ns)	Total Time (ns)
Single Shared Bus	42480	-	42480
Multiple Bus	26905	12800	39705
FLEXBUS (Dynamic Bridge By-pass)	27025	5290	32315
FLEXBUS (Dynamic Component Re-mapping)	27010	5270	32280
Ideally Reconfigurable Bus	26905	5120	32025

Next, we examine the performance of FLEXBUS and conventional communication architectures for the IEEE 802.11 MAC processor described in Section III.C.1. We considered two variants of the FLEXBUS architecture, (i) featuring dynamic bridge by-pass, and (ii) featuring dynamic component re-mapping (`Frame_Buf2` being a re-mappable slave). For the first variant, the bus configuration policy described in Section III.E selects when to disable or enable the bridge at run-time. For the second variant, since `Frame_Buf2` is the only re-mappable component, the re-mapping policy dynamically maps it to the bus segment from which it receives the most number of transactions. All the buses were operated at 200 MHz. Table III.1 shows the average time taken to process a single frame (of size 1 KB) under the different bus architectures. From the table, we see that the times required by both variants of the FLEXBUS architecture are significantly smaller compared to the conventional architectures. The data rate increase due to FLEXBUS over the single shared bus is 31.5% and over the multiple bus is 23%. The table also shows the upper bound on performance, obtained using an ideal reconfigurable bus (zero reconfiguration overhead) with an ideal reconfiguration policy (complete knowledge of future bus traffic). We observe that for this system, FLEXBUS and its associated policies perform close (data rate within 1%) to the ideal case.

III.G.5 Application to a UMTS Turbo Decoder Design

Finally, we apply FLEXBUS with dynamic bridge by-pass to the design of a Turbo decoder for the Universal Mobile Telecommunications System (UMTS) specification, and evaluate its performance compared to conventional static bus architectures.

Turbo coding has received considerable attention in recent years due to its near Shannon capacity performance [135], and has been included in the specifications for both the WCDMA (UMTS) [136] and cdma2000 [137] third-generation cellular standards. Figure III.11(a) and (b) show the functional blocks in a turbo encoder and decoder, respectively (details are available in [138, 139]). The turbo encoder consists of two recursive systematic convolutional (RSC) encoders connected in parallel with an interleaver between them. For an input frame X, the outputs of the turbo encoder are the systematic bits (X), and the parity bits from the two RSC encoders (Z_1 and Z_2). The turbo decoder consists of two RSC component decoders linked together by an interleaver and a de-interleaver. The inputs to the turbo decoder are the noise-contaminated received frame systematic bits (X') and parity bits (Z_1' and Z_2'). As indicated by the feedback path, the turbo decoder operates in an iterative manner. In each iteration, the first component decoder generates soft outputs (L_{e1}) about the likely values of the bits to be decoded in terms of the log-likelihood ratios (LLR - logarithm of the ratio of the probability of the bit being 1 to the bit being 0), which are then interleaved and input to the second decoder. The second decoder then generates another set of LLR values (L_{e2}), which are fed back to the first decoder after de-interleaving them. After a number of such iterations (typically 8 for low bit error rate), a hard (0 or 1) decision is taken about the value of each bit.

Figure III.11(c) shows the mapping of the Turbo decoder functional blocks to hardware components in our design. The two RSC decoders are based on the *log-MAP* algorithm [140] and are implemented by custom hardware components, DCDR1 and DCDR2, respectively, using the sliding window approach [141]. The interleaver and de-interleaver functionality is implemented by the INT/DE-INT hardware component. The system processes two input frames simultaneously. While DCDR1 processes

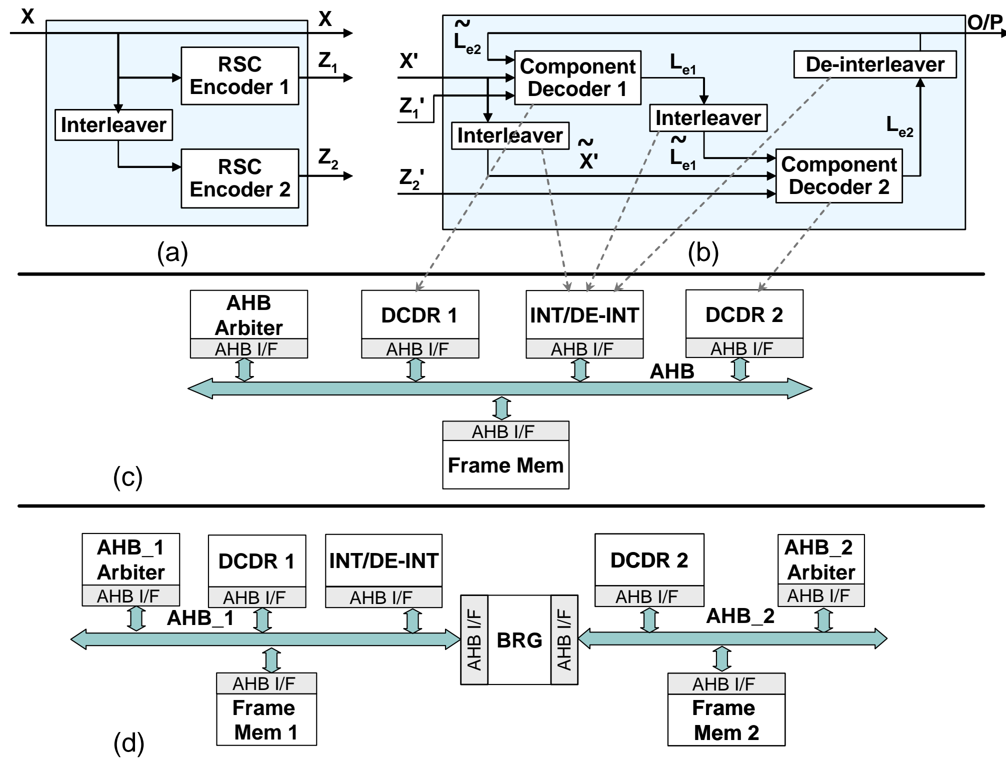


Figure III.11: UMTS Turbo encoder and decoder: (a) encoder functional blocks, (b) decoder functional blocks, (c) mapping to a single shared bus architecture, (d) mapping to a multiple bus architecture

the first frame, DCDR2 processes the second frame. On completion, they signal the INT/DE-INT, which then interleaves the output of DCDR1 and de-interleaves the output of DCDR2. The processing of the frames is now swapped with DCDR1 processing the second frame and DCDR2 processing the first frame. After eight such iterations, the decoded bits are read out and the system starts processing a new pair of frames.

Figure III.11(c) shows the design under a single shared bus architecture with `Frame_Mem` storing all the input and intermediate frame data. Figure III.11(d) shows the design implemented using a multiple bus architecture, with DCDR1, INT/DE-INT and `Frame_Mem1` mapped to AHB1, and DCDR2 and `Frame_Mem2` mapped to AHB2. The input and output frame data of DCDR1 and DCDR2 are stored in `Frame_Mem1` and `Frame_Mem2`, respectively. The INT/DE-INT block processes the output of DCDR1

from Frame_Mem1 and stores the interleaved data in Frame_Mem2, and de-interleaves the output of DCDR2 from Frame_Mem2 and stores it in Frame_Mem1. The system is also operated under FLEXBUS with dynamic bridge by-pass, using the bus configuration policy described in Section III.E. All the buses were operated at 200 MHz.

Table III.2 shows the performance of the system while processing two frames each of size 1 KByte, in terms of the time taken for the decoding phase, and interleaving and de-interleaving phase for each half-iteration, the total time to process both the frames, and the corresponding data rate, under the different bus architectures. From the table, we see that the multiple bus performs better compared to the single shared bus during the decoding phase by avoiding bus conflicts between DCDR1 and DCDR2, but suffers during the interleaving/de-interleaving phase due to the bridge overhead. By enabling dynamic by-pass of the bridge, FLEXBUS performs much better than either static architecture, achieving a data rate improvement of 34.33% over the single shared bus and 30.27% over the multiple bus architecture.

Table III.2: Performance of the UMTS Turbo decoder under different communication architectures

Bus Architecture	Decoding Phase (ns)	Interleaving/ De-interleaving Phase (ns)	Total Time (ns)	Data Rate (Mbps)
Single Shared Bus	20620	16080	587200	3.4877
Multiple Bus	10910	24680	569440	3.5965
FLEXBUS (Dynamic Bridge Bypass)	11095	16225	437120	4.6852

III.H Conclusions

In this chapter, we illustrated that significant performance benefits can be achieved by configuring the on-chip communication architecture topology in response to variations in traffic characteristics. We presented FLEXBUS, a novel dynamically configurable communication architecture, featuring two different configurability options: (i) dynamic bridge by-pass, and (ii) dynamic component re-mapping, and described

techniques for efficiently adapting FLEXBUS at run-time. Extensive experiments on FLEXBUS using a commercial design flow to analyze its area and performance under a wide variety of traffic profiles, and its application to the design of an IEEE 802.11 MAC processor and a UMTS Turbo decoder, demonstrate its superiority over conventional communication architectures.

The text of this chapter, in part, is based on material that has been published in the Design Automation Conference, 2005, and material submitted to the IEEE Transactions on VLSI Systems. The dissertation author was the primary researcher and author, and the coauthors listed in these publications collaborated on, or supervised the research that forms the basis for this chapter.

IV

Dynamic Management of SoC Platforms with Configurable Processors and Dynamic Data Relocation

IV.A Introduction

In Chapter II, we described several dynamic configurability features emerging in SoC platform components, and motivated the need for platforms with multiple such options, so that they can be better adapted to the applications' requirements. We also presented the concept of dynamic platform management for the holistic, run-time configuration of such platforms. In this chapter, we propose SoC platforms featuring two different configurability options, (i) configurable processors that support frequency and voltage scaling, and (ii) flexible data relocation in memory, and explain how dynamic platform management can be applied for such platforms.

Frequency and voltage scalable processors allow the operating clock frequency and supply voltage to be dynamically changed resulting in a significant impact on the processor energy consumption (Section II.B.1). This is because the dominant

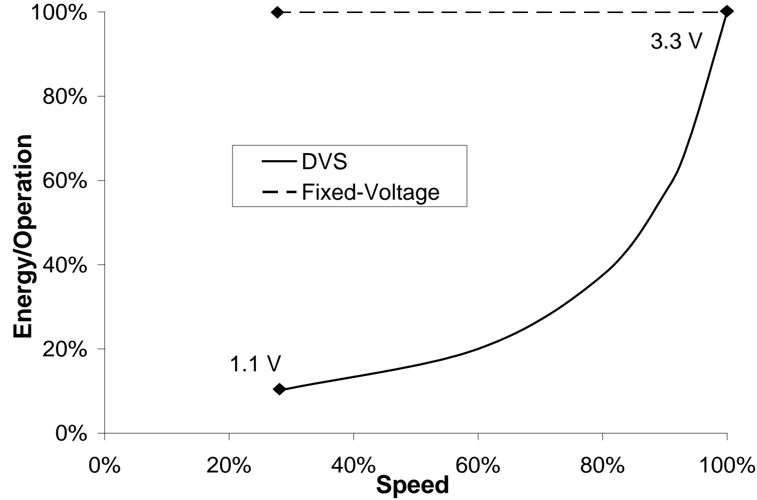


Figure IV.1: Energy versus speed trade-off [1]

source of power dissipation in digital CMOS circuits is the switching component of power, which is given by $P = p_t C_L V_{dd}^2 f_{clk}$ [142], where p_t is the switching activity in the circuit, C_L is the load capacitance, V_{dd} is the supply voltage, and f_{clk} is the clock frequency. The energy per transition is therefore given by [142]

$$energy\ per\ transition = \frac{P}{f_{clk}} = p_t C_L V_{dd}^2 \qquad \text{(IV.1)}$$

which implies that a reduction in the supply voltage leads to a quadratic reduction in the energy consumed. Reducing the supply voltage, however, increases the gate delay, which can be approximated by

$$delay = \frac{C_L \times V_{dd}}{\mu C_{ox}(W/L)(V_{dd} - V_t)^2} \qquad \text{(IV.2)}$$

where μ is the carrier mobility, C_{ox} is the oxide capacitance, W and L are the width and length of the gate, respectively, and V_t is the threshold voltage [142]. Therefore, the maximum speed at which a circuit can be clocked monotonically decreases as the voltage is reduced. These two relationships expose the trade-off between energy and speed which can be exploited by varying the voltage, graphically depicted in Figure IV.1 [1]. Several frequency and voltage scalable processors are commercially available today, such as XScale [49] and Crusoe [48].

Flexible data relocation allows the data objects accessed by the executing applications to be relocated among the system memories at run-time. Correct data relocation requires that accesses to the relocated objects should find them at their new location. Several techniques to enable this are discussed in Section II.B.3. These techniques provide the capability of dynamically partitioning the application data among the different memories. Since different data objects are accessed by different applications, and the data objects accessed by an individual application may also change over time, there might be a significant variation in the system memory access profile at run-time. Hence, such data relocation, if applied intelligently, can provide significant benefits to the cache and memory system performance, by adapting to run-time application behavior.

The reason for incorporating these two configurable features on the same platform is that there is a dependence between the operating frequency and voltage, and the placement of data in memory, as illustrated in this chapter. Therefore, integrated dynamic platform management techniques, which take into account such dependencies are necessary for the optimized configuration of such platforms.

IV.A.1 Chapter Overview

In this chapter, we propose dynamically configurable SoC platforms featuring fine-grained frequency and voltage scaling, and flexible relocation of application data between on-chip and external memory. We demonstrate the dependence between these two configurability options, and motivate the need for their integrated configuration. We illustrate the application of dynamic platform management to such platforms, and describe the methodology in detail. The methodology enables optimized usage of available CPU and on-chip memory resources. As described in Section II.C, the methodology consists of two parts. In the first part, we develop detailed (off-line) characterizations of how platform resource usage (*e.g.*, CPU cycles, memory accesses) varies with the characteristics of the set of executing applications. The second part consists of run-time platform management algorithms that customize the platform for time-varying application requirements. We demonstrate how our techniques can achieve significant gains in

performance and energy efficiency by applying it to the design of a dual-access UMTS and WLAN wireless security processing system. The system was implemented on two different configurable platforms: (i) a StrongARM based platform, and (ii) the Altera Excalibur development board. Experiments demonstrate that, compared to a conventionally optimized design (on the same general-purpose platform), the proposed techniques enable up to 160% improvements in security processing throughput, and achieve 59% energy savings (on average).

The rest of this chapter is organized as follows. In Section IV.A.2, we describe related work. In Section IV.B, we describe the configurable platform architecture that we consider in this chapter. In Section IV.C, we illustrate, using the UMTS/WLAN security processing system as an example, the execution of the system based on dynamic platform management, and the advantages it provides. In Section IV.D, we present details of the methodology, considering the key steps, and algorithms employed. We present experimental results in Section IV.E that evaluate the performance and energy-efficiency of the developed security processing system, and compare it to a conventionally optimized design. Finally, we conclude in Section IV.F.

IV.A.2 Related Work

A significant body of work has emerged relating to dynamic scaling of processor frequency and voltage (DVS) to take advantage of available slack in processing workloads. These techniques are described in Section II.B.1. Techniques for relocating application data at run-time have also been proposed as described in Section II.B.3. Most data relocation techniques are applicable only for single applications, and are compile-time, relying on the compiler to analyze the behavior of the application and insert appropriate code to perform data relocation. Therefore, such techniques would perform poorly in the presence of multiple time-varying applications. In contrast, our dynamic platform management methodology analyzes the requirements of multiple applications and globally optimizes the placement of data in memory. Also, previous works consider the configuration of the frequency and voltage, and data placement in

isolation. In this work, we analyze the interaction between these two configurability options and present integrated platform optimization techniques.

IV.B Configurable Platform Architecture

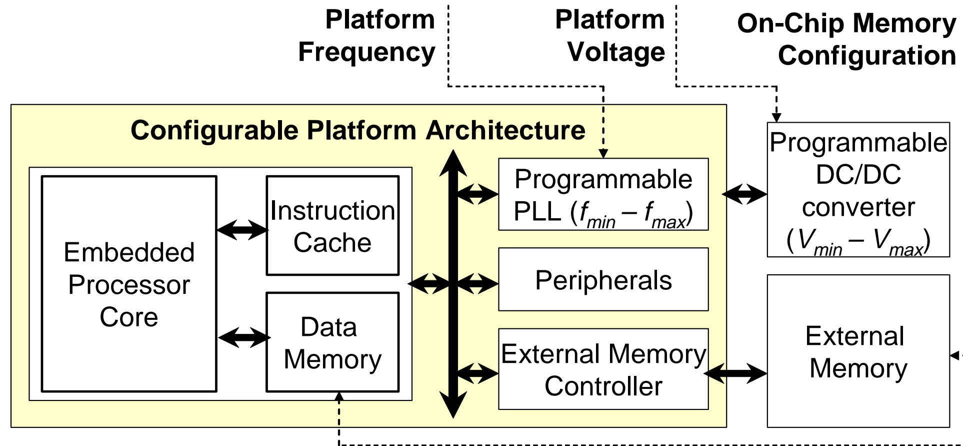


Figure IV.2: A general-purpose configurable platform architecture, featuring frequency and voltage scaling, and flexible data relocation

The configurable platform architecture proposed in this work is shown in Figure IV.2. It consists of an embedded processor core, a small and fast on-chip data memory (SRAM), and an instruction cache. Such dedicated on-chip memories are often used in embedded systems in place of data caches, to reduce power consumption, exploit application characteristics, and obtain more predictable execution [143, 144]. The platform is also connected to a larger and slower external memory through an external memory controller. The platform may contain other hardware peripherals such as UARTs, timers, interrupt controllers, *etc*.

The platform features two dynamic configurability options. First, the platform can be operated at a set of discrete frequency and voltage levels, which can be changed dynamically. This is enabled through the use of programmable clock generators (PLLs) and programmable, variable voltage DC/DC converters (Figure IV.2). Second, it features flexible data relocation that supports the dynamic partitioning of the data objects

used by the various software tasks between the on-chip data memory and the external memory. This provides the flexibility of selecting at run-time, an optimized set of data objects for storage in the on-chip memory depending on the applications' requirements. Data relocation can be enabled using various techniques such as through the virtual-to-physical address mapping, base pointer registers, *etc* (Section II.B.3). In this work, data relocation is provided using the "cache locking" feature available in many data caches [145, 146]. Cache locking allows specified cache lines to be locked so that the data in them is not replaced by a linefill. Locking all the lines of the cache, therefore, enables it to be used as dedicated on-chip data memory. The cache locking mechanism is controlled through programmable cache control registers [146]. To relocate a data object to the on-chip memory, first cache locking is disabled, the data object is accessed so that it is loaded into the cache, and then the cache is again locked down. To relocate a data object back to the external memory, cache locking is disabled, the relevant cache lines are cleaned and marked as invalid, after which the cache is locked again. On data relocation, future references to the relocated data objects find them at their correct location using the normal cache addressing mechanism: if the data object is in the on-chip memory, it results in a cache hit, whereas if it is in external memory, it is accessed (but not loaded into the on-chip memory if it is locked) after a cache miss.

The space of platform configurations is defined by two dimensions. The first dimension consists of different (f, v) pairs, which define clock frequencies and their associated supply voltage values. The second dimension consists of the different ways in which the set of data items (accessed by the tasks) can be partitioned between on-chip and external memory. For a set of data items $D = \{D_1, D_2, ..., D_n\}$, a configuration of the on-chip memory is defined by a set $D_{on} \subseteq D$, such that the total size of items in D_{on} does not exceed the capacity of the on-chip memory. A candidate platform configuration is defined by $\langle f, v, D_{on} \rangle$. A configuration is successful in meeting performance requirements, if the associated CPU utilization is no more than 100%.

IV.C Demonstrating Platform Management for Security Processing

In this section, we illustrate the operation and advantages of dynamic platform management for the platform described in the previous section, using a dual-access (UMTS and WLAN) security processing system as an example. We first describe the security processing tasks, and their mapping to the platform architecture. Next, we highlight the space of available platform configurations. We then illustrate the problems associated with configuring the platform statically, and finally, illustrate how dynamic platform management chooses optimized platform configurations at run-time, and thereby achieves desired security processing throughput, and improvements in energy-efficiency. The experiments were performed using cycle-accurate, instruction-set simulation of a StrongARM based platform architecture, details of which are presented in Section IV.E.1.

IV.C.1 Case Study: UMTS and WLAN Security Processing

The system implements Layer 2 security protocols of two wireless standards: the Universal Mobile Telecommunications System (UMTS) for third generation cellular networks [147], and the IEEE 802.11b standard for wireless LANs [121]. Our design is motivated by the emergence of converged handsets, capable of simultaneous communication over multiple wireless interfaces [148, 149]. The need to support upgrades (due to the evolving nature of security protocols), while achieving high security processing throughput and energy-efficiency, makes a general-purpose, programmable and configurable platform a suitable implementation choice (Figure IV.2).

The dual-access security processing system executes two tasks. The UMTS task is responsible for ciphering and integrity of UMTS frames [147]. Each UMTS frame may be ciphered and integrity checked, or only ciphered, depending on the frame type. All frames are ciphered using the *f8* algorithm. In addition, signalling frames are also integrity checked using a 32-bit Message Authentication Code, computed using the *f9* algorithm. Both *f8* and *f9* are are based on the *KASUMI* block cipher [147].

The WLAN task (defined in [121]) is responsible for encrypting Layer 2 frames (if encryption is enabled) using the Wired Equivalent Privacy (WEP) protocol, which is based on a 64 bit symmetric key stream cipher, and computing a Frame Check Sequence (a 32 bit CRC) for data integrity. While processing a UMTS (or WLAN) frame, the UMTS (or WLAN) task accesses a set of data objects (*e.g.*, CRC tables, substitution tables, the program stack, frame data *etc*).

The security processing tasks (UMTS and WLAN) are mapped to the configurable platform described in the previous section. The specific instance of the configurable platform architecture that we consider in this example is based on a StrongARM processor core [150], described in detail in Section IV.E.1. The size of both the instruction cache and the on-chip data memory is 1.5 KBytes. The UMTS and WLAN tasks execute on the StrongARM core, and the various task data objects are partitioned between the on-chip and external memory (since all the items cannot be accommodated in the on-chip memory). The tasks individually exhibit significant dynamic variation in their CPU requirements and memory access patterns, due to variable data rates, frame types and frame size distributions.

IV.C.2 Platform Configuration Space

Figure IV.3 depicts three workload scenarios, **Case 1**, **Case 2** and **Case 3**. For all cases, both the UMTS and WLAN tasks are active, and are processing respective frames. In this example, the frame characteristics are held constant (as shown in Figure IV.3), while the data rate of each task varies as shown. Below each case, we show a pair of tables, one for each of the two tasks. The UMTS table for **Case 1** depicts a space of possible platform configurations $\langle f, V, D_{UMTS} \rangle$, with the CPU% consumed by the UMTS task under the given workload. For example, row 1 of Table UMTS-Case1 indicates that if the platform is operated at 206 Mhz, 1.5 V, and if the `Stack`, `Key Schedule (KS)`, and `S7` data items are stored in the on-chip memory, then while processing 5114 bit signalling frames at 1.8 Mbps, the UMTS task consumes 57.8% of the CPU. A similar table is presented for WLAN-Case1, and for the other cases.

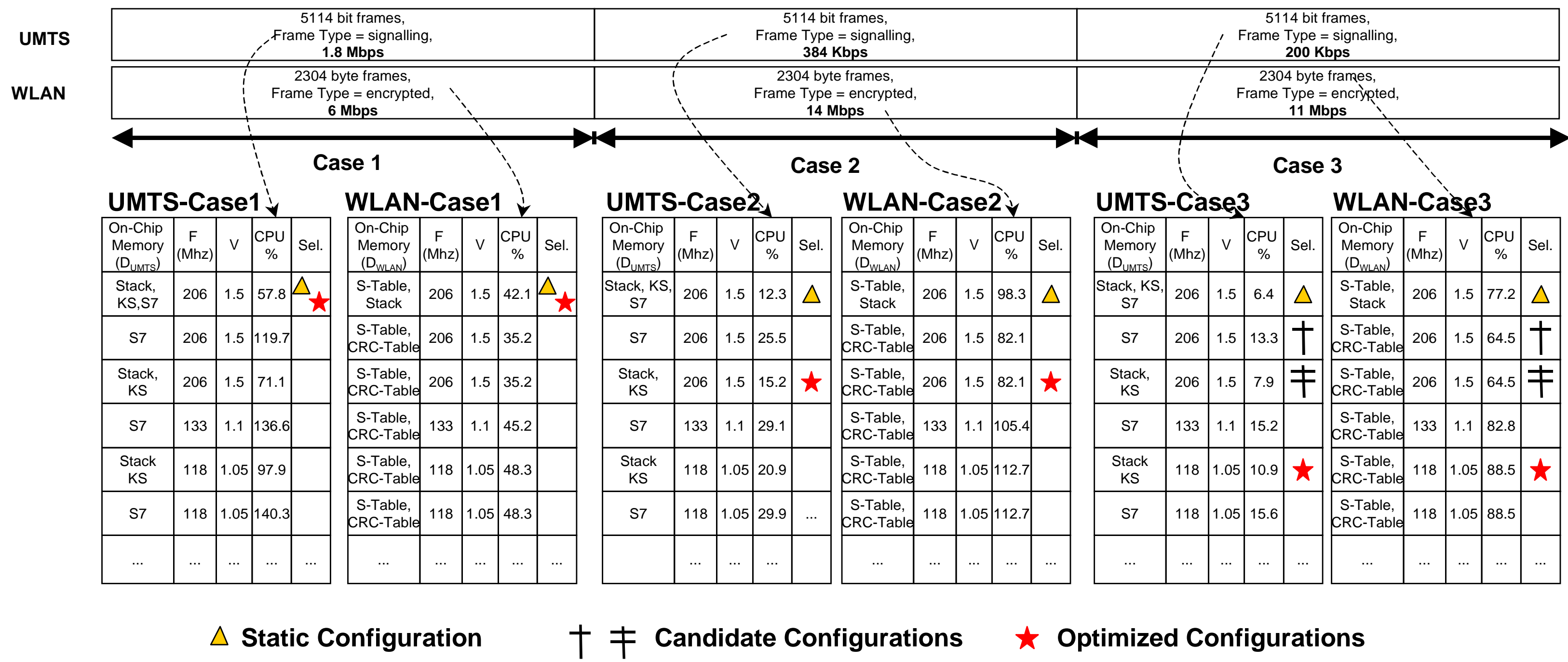


Figure IV.3: Dynamic selection of optimized platform configurations for UMTS and WLAN security processing

At any given time, a candidate platform configuration is defined by a pair of rows, one from each table, such that the following two conditions are met: (i) the frequency and voltage in the two selected rows are identical, (ii) the total size of the data items stored in the on-chip memory is less than its capacity. A platform configuration that satisfies these conditions, and achieves no more than 100% CPU, will satisfy performance (data rate) requirements.

IV.C.3 Security Processing: Static Configuration

We first consider the execution of the platform for each of the three cases depicted in Figure IV.3, when it is statically configured. The particular static configuration that we choose is one that is optimized for a large space of requirements (details in Section IV.E.1). Figure IV.3 illustrates the sequence of platform configurations, showing how, in this case, the pair of "selected rows" remain fixed over time (indicated by $\triangle$). The static configuration is defined by $\langle f, V, D_{on} \rangle$, where $f = 206Mhz$, $V = 1.5V$, $D_{on} = \{\texttt{Stack}_{\texttt{UMTS}}, \texttt{KS}_{\texttt{UMTS}}, \texttt{S7}_{\texttt{UMTS}}, \texttt{S} - \texttt{Table}_{\texttt{WLAN}}, \texttt{Stack}_{\texttt{WLAN}}\}$. Inspection of the CPU% figures for this architecture indicates the following problem. While for **Case 1**, the total CPU% is $57.76 + 42.11 = 99.87\%$, in **Case 2**, the rows corresponding to the static architecture ($\triangle$) have CPU% values that add up to 110.58%, indicating that the static architecture is not capable of satisfying the processing requirements imposed by **Case 2**.

IV.C.4 Security Processing: Dynamic Platform Management

We next illustrate the execution of the security processing tasks with the proposed dynamic platform management technique, for the three cases considered in the previous example. The platform management technique considers (i) time-varying requirements imposed by the individual tasks (by examining workload parameters, such as frame sizes, types, and data rate requirements), and (ii) pre-determined characteristics of each task, to choose an optimized configuration of the platform. For example, under **Case 1**, the platform management technique chooses a platform configuration that

is identical to the static configuration used in the previous example. However, as the requirements imposed by the applications change, the selected platform configuration may change, as illustrated next.

When the requirements change from **Case 1** to **Case 2**, the platform management techniques consider the space of possible platform configurations available to further optimize the system for **Case 2** (defined in Tables UMTS-Case2 and WLAN-Case2). Recall that a pair of rows from these uniquely defines a platform configuration. Clearly the pair of rows labeled with $\triangle$ define a platform configuration that cannot meet the requirements imposed by **Case 2** (total CPU exceeds 100%). A possible configuration is the one defined by the pair of rows labeled with a $\star$. This configuration differs from the previous one in terms of the set of data items stored in the on-chip memory: the items $\texttt{S7}_{\texttt{UMTS}}$ and $\texttt{Stack}_{\texttt{WLAN}}$ are replaced by $\texttt{CRC} - \texttt{Table}_{\texttt{WLAN}}$. Under the new configuration, the CPU% consumed by the two tasks are 15.17% and 82.12%, resulting in a total of 97.29%, which is less than 100%. Hence, this is determined to be a platform configuration that satisfies **Case 2**.

However, the platform management technique does more than just select an alternate set of data objects to be stored in memory. It does this in a manner so as to minimize wasted CPU cycles, and hence increase CPU availability. To illustrate this, consider **Case 3**, where in the static case, (denoted by $\triangle$), the total CPU% is $6.42 + 77.20 = 83.62\%$. Even though this configuration meets performance requirements comfortably, the platform management technique considers the space of possible configurations that might result in more efficient use of the CPU. For example, to process the workload of **Case 3**, the configuration defined by the pair of rows labeled with † results in 77.82% CPU utilization, whereas the configuration defined by the pair of rows labeled with ‡ result in 72.42% CPU utilization. The platform management technique selects ‡ over †, since ‡ achieves 13% savings in CPU cycles, while † achieves only 7% savings. Reducing CPU utilization can enable (i) accommodation of other processing tasks (if they exist), or (ii) in our case, reductions in power consumption via frequency and voltage scaling. In the example, the platform management technique se-

lects a configuration defined by the pair of rows labeled with ⋆. Exploiting the resultant CPU slack enables operating the platform at a lower frequency (118 Mhz) and voltage (1.05 V), leading to energy savings. Note that, if the memory configuration defined in † had been used, then the platform would have had to be operated at 133 Mhz and 1.1 V, which would have led to higher power consumption. On the other hand, if the platform was operated at 118 Mhz (the optimum speed for ‡), it would have led to a total CPU utilization of 104.1%, resulting in failure to meet performance requirements.

From this example, we draw a few important conclusions:

- Tasks can exhibit significant dynamic variation in their workload characteristics, resulting in a wide range of processing requirements imposed on a platform. While we considered data rate as a variable parameter, the platform resource usage profiles could vary significantly due to several other factors (*e.g.*, the exact set of tasks currently active, and frame properties, such as frame types, sizes, *etc*).

- Dynamically configuring the platform while exploiting (i) an accurate characterization of application tasks, and (ii) a detailed knowledge of the platform architecture can help improve platform resource utilization (CPU, memory), resulting in performance improvements, and large energy savings.

- Platform management techniques should be based on tightly coupled algorithms for dynamically optimizing different components and parameters for maximizing performance and energy-efficiency. The example illustrated how usage profiles of different platform components are interdependent, and demonstrated how intelligent use of an on-chip memory helps free up CPU cycles (by reducing the number of slow external memory accesses), enabling larger CPU headroom, or potential power savings.

IV.D Dynamic Platform Management Methodology

In this section, we describe the details of the dynamic platform management methodology for the proposed configurable SoC platform (Section IV.B). We first define terminology, and then go on to describe (i) the off-line task characterization step, and (ii) the dynamic platform management algorithms that optimize the platform configuration at run-time, so as to meet performance requirements, while minimizing energy consumption.

IV.D.1 Terminology

We consider tasks with periodic arrivals, having soft real-time requirements. At a given time, let the set of currently executing tasks be denoted by $T = \{T_1, T_2, ..., T_M\}$, where $M \leq MAX$, the total number of tasks in the system. Associated with each task T_i, where $1 \leq i \leq MAX$, is a set of data objects, $D_i = \{d_{i,1}, \ldots, d_{i,m_i}\}$. A data object refers to a logical data structure, or data block, that can be addressed contiguously by the task. For example, the CRC table used for computing the 32 bit checksum in the WLAN task is a data object. Each data object, $d_{i,j}$, has an associated maximum size $s_{i,j}$. Each instance of a task T_i, has a time-interval P_i, within which it has to finish executing. Let $N_i = \{n_{i,1}, n_{i,2}, ..., n_{i,m_i}\}$ denote a set consisting of the number of accesses T_i makes to each data object in D_i in time-interval P_i (*i.e.*, there is a one-one mapping between D_i and N_i). Let C_i denote the number of processing cycles (excluding data memory access cycles) required by each instance of the task T_i in the time-interval P_i. Given D_{on}, the set of data objects in on-chip memory, the execution time ET_i for each instance of a task T_i is estimated using the following:

$$ET_i = (C_i + nC_{on-chip} + (N - n)(\lceil \frac{T_{ext}}{1/f} \rceil)) * \frac{1}{f} \quad \text{(IV.3)}$$

where C_i is the number of processing cycles, $C_{on-chip}$ is the number of cycles to access on-chip data memory, T_{ext} is the time to access external memory, $N = \sum N_i$ is the total number of data accesses, $n = \sum_{d_{i,j} \in D_{on}} n_{i,j}$ is the number of data accesses to on-chip data memory, and f is the operating frequency. The non-linear effects due to external

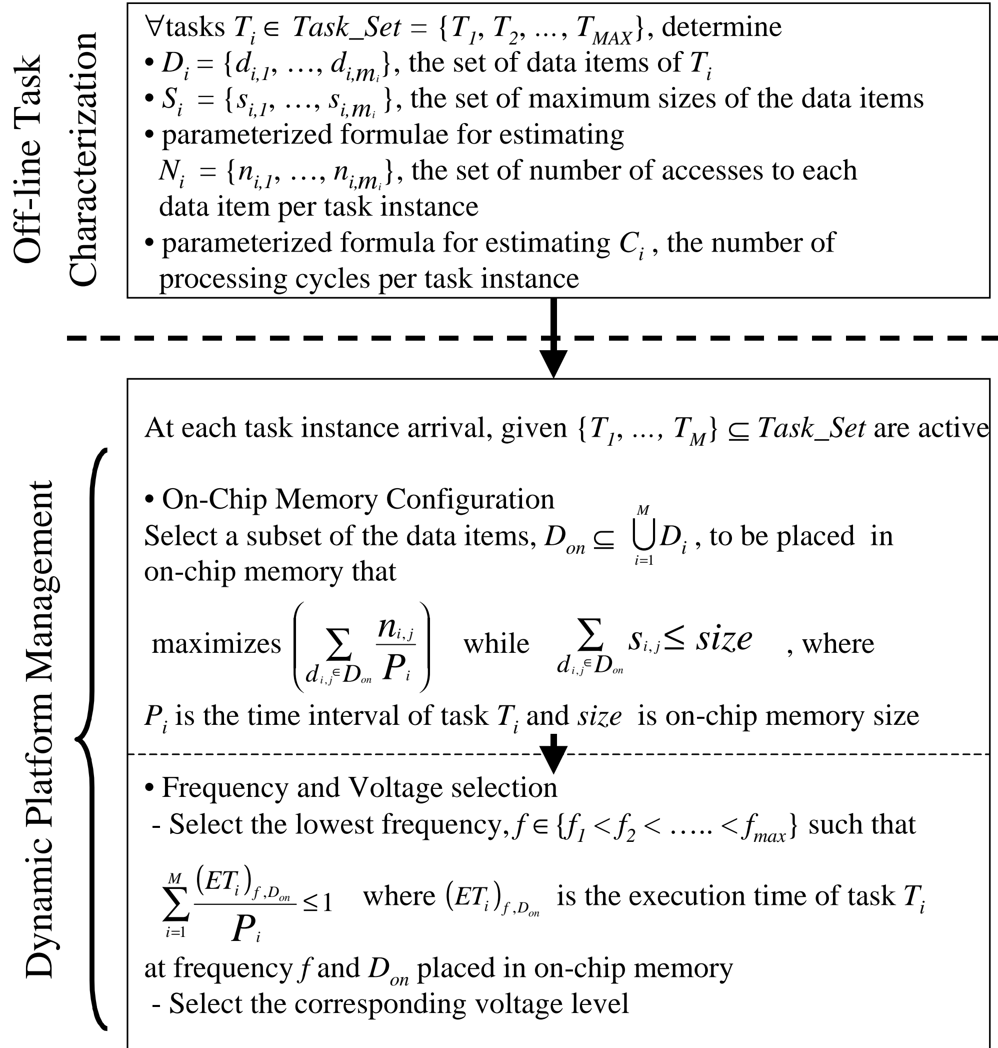


Figure IV.4: Dynamic platform management methodology

memory are accounted for by the $\lceil \frac{T_{ext}}{1/f} \rceil$ term. The values of $C_{on-chip}$ and T_{ext} depend on the platform.

In order to dynamically determine the optimized platform configuration, the platform management algorithms are provided with certain characteristics of the tasks which execute on the platform. This task specific information is obtained by characterizing each task off-line, using a procedure we describe next.

IV.D.2 Off-line Task Characterization

The steps performed in this off-line phase are illustrated by the first box in Figure IV.4 and the resulting characterization tables are illustrated by Table IV.1. For each task, $T_i, 1 \leq i \leq MAX$, which potentially executes on the platform, all the data addressed by the task is first divided into a set of logical data objects, D_i. Each data object $d_{i,j} \in D_i$ should be contiguously addressable by the task. Next, the maximum sizes of these data objects ($s_{i,j}$ values) are determined through a combination of analysis and simulation. Each data object $d_{i,j}$ is then characterized by the number of times, $n_{i,j}$, it is accessed by an instance of its associated task T_i. In general, the value of $n_{i,j}$ varies, depending on dynamically variable parameters of the task ($p_{i,k}$), and properties of the data being processed. For example, in a security processing system, the parameters may include the encryption key lengths, frame lengths and frame types. Hence, the result of this step is a set of formulae/models for estimating the number of accesses to each data object at run-time ($fn_{i,j}$). In addition, similar models for estimating the number of processing cycles, C_i, for each task instance are developed. Note that, we base our estimation models on parameters whose values can be obtained at task arrival. If such parameters are not available, accurate off-line estimation may prove difficult. While considering such tasks is beyond the scope of this work, we believe that our methodology can be extended to incorporate predictive strategies for estimating task characteristics at run-time ([151]).

Table IV.1: Task characterization tables

Task	Data Items ($d_{i,j}$)	Max Size ($s_{i,j}$)	Estimated Number of Accesses ($n_{i,j}$)	Proc. Cycles (C_i)
T_1	$d_{1,1}$	$s_{1,1}$	$n_{1,1}=fn_{1,1}(p_{1,1}, p_{1,2}, ...)$	$C_1 = fn_1(p_{1,1}, p_{1,2}, ...)$
	$d_{1,2}$	$s_{1,2}$	$n_{1,2}=fn_{1,2}(p_{1,1}, p_{1,2}, ...)$	
	...	...	...	
	d_{1,m_1}	s_{1,m_1}	$n_{1,m_1}=fn_{1,m_1}(p_{1,1}, p_{1,2}, ...)$	
...	...	...	...	...
	...	...	...	

Once these characterization tables are generated, they are incorporated into the dynamic platform management algorithms (Figure IV.4), which use them to optimize the platform configuration at run-time. Note that, in order to incorporate a new task into the platform, or to target the platform to a new set of application tasks, the platform management implementation remains the same. Only this off-line characterization step needs to be performed to generate the required tables, and provided to the platform management algorithms. In the next subsection, we describe how these characterization tables are used to select optimized platform configurations at run-time.

IV.D.3 Dynamic Platform Management Algorithms

At the arrival of each task instance, the dynamic platform management techniques choose an optimized configuration by (i) deciding on the on-chip memory configuration, and (ii) calculating a "memory-aware" frequency and voltage setting (Figure IV.4). We next describe how the memory, and frequency and voltage decisions are taken. Finally, we discuss the implications of the overhead imposed by the platform management algorithms.

On-Chip Memory Configuration

The task of configuring the on-chip memory consists of dynamically selecting the optimal set of data objects, D_{on}, to be placed in on-chip memory given a set of executing tasks. Given two different on-chip memory configurations, D_{on1} and D_{on2}, the one which reduces the CPU stall cycles as it waits for external memory is better, since it results in more efficient (lower) CPU utilization. Since the total CPU utilization is given by $\sum_{i=1}^{M} \frac{ET_i}{P_i}$, therefore, D_{on1} is preferable to D_{on2}, if :

$$\sum_{i=1}^{M} \frac{ET_{i,D_{on1}}}{P_i} < \sum_{i=1}^{M} \frac{ET_{i,D_{on2}}}{P_i} \tag{IV.4}$$

where M is the number of currently executing tasks, ET_i is the execution time of task T_i and P_i is its time-interval. Using Equation IV.3 and solving the above inequality, we get:

$$\sum_{d_{i,j} \in D_{on1}} \frac{n_{i,j}}{P_i} < \sum_{d_{i,j} \in D_{on2}} \frac{n_{i,j}}{P_i} \qquad \text{(IV.5)}$$

Equation IV.5 holds, provided on-chip memory access time is less than external memory access time (which is true). Hence, that on-chip memory configuration, which maximizes the rate of on-chip memory accesses, is optimal. This is subject to the constraint that the set of selected data objects fit within the limited on-chip memory, *i.e.,* $\sum_{d_{i,j} \in D_{on}} s_{i,j}$ should not exceed the size of the on-chip memory (Figure IV.4). The problem of optimizing the on-chip memory can be formulated in terms of the *Knapsack* problem (which is *NP*-complete [127]). Hence, we use a greedy strategy to dynamically choose the set D_{on} by using $\frac{n_{i,j}/P_i}{s_{i,j}}$, the ratio of the rate of memory accesses to the size of the data object, as the cost function. Since this involves sorting the data objects according to the cost function, the complexity of the memory configuration decision is $O(nlog(n))$, where n is the total number of data objects belonging to the active tasks.

As optimizing the on-chip memory configuration frees up CPU cycles, this enables the possibility of operating the CPU at a lower frequency and voltage setting.

Frequency and Voltage Setting

After optimizing the on-chip memory configuration, the dynamic platform management layer selects the frequency and voltage at which to operate the platform (Figure IV.4). Accurate off-line characterizations of the tasks, and knowledge of the currently selected memory configuration, enables the platform management layer to aggressively scale the operating frequency and voltage. A pre-emptive EDF scheduler is used for scheduling the arriving tasks, and the schedulability test for EDF [152] is used to determine the lowest frequency f (among a set of discrete frequencies of the platform,

$f_1 < f_2 < ... < f_{max}$) at which the set of active tasks can still meet their performance requirements. The frequency is determined from the following:

$$\sum_{i=1}^{M} \frac{(ET_i)_{f,D_{on}}}{P_i} = 1; \qquad \text{(IV.6)}$$

where $(ET_i)_{f,D_{on}}$ is the execution time of the currently active instance of task T_i, under frequency f and with the set of data objects D_{on} in on-chip memory (from Equation IV.3), and P_i is the execution time-interval of T_i. The voltage level is selected corresponding to the selected frequency setting. The complexity of frequency and voltage selection is $O(n)$, where n is the number of active tasks in the system.

Platform Management Overhead

The overhead associated with a platform configuration decision involves (i) the time taken to select the new platform configuration, (ii) the time taken to re-program the platform frequency and voltage, and (iii) the time taken to reconfigure the on-chip memory. The time for selecting the new configuration depends on the number of active tasks in the system and the corresponding total number of data objects. For example, for the security processing system described in Section IV.C, the time taken to select the new memory, frequency and voltage configuration, when both the UMTS and WLAN tasks are active, is approximately 10 μs. Many commercial platforms feature fine-grained frequency and voltage scaling, where the frequency and voltage can be changed with very little overhead. For example, the time taken to change the frequency of the StrongARM processor is approximately 150 μs [153]. The worst-case time for reconfiguring the on-chip memory is approximately 200 μs, which occurs when the entire contents of the on-chip memory are re-organized (assuming a clock frequency of 206 Mhz, single cycle on-chip memory access and $50\ ns$ external memory access time).

Platform management decisions are potentially taken at the arrival of each task instance. If the times between successive task arrivals are large, ($> 10\ ms$), then platform configuration decisions can be made at each task arrival with insignificant overhead. However, if the task time-periods are small (say hundreds of μs) then the overhead

of platform configuration may out-weigh its benefits. In such cases, platform configuration decisions need to be taken at coarser time-scales. For example, the platform management decisions may be taken at the arrival of the first task following the expiration of a fixed time interval. Since workload characteristics of future task instances are unknown, history-based prediction strategies can be used to estimate future task characteristics. In our work, we found that using a $10\ ms$ interval between platform management decisions proved to be effective.

IV.E Experimental Results

In this section, we present experimental results that evaluate the effectiveness of applying the proposed platform management techniques to the dual-access UMTS and WLAN security processing system described in Section IV.C.1. Two sets of experiments were performed, in which the system was mapped to two different instances of the proposed configurable SoC platform (Section IV.B): (i) a StrongARM [150] based platform, and (ii) the Altera Excalibur system-on-a-programmable-chip (SOPC) [28]. We next describe each of these sets of experiments in detail.

IV.E.1 Application to StrongARM based platform

Platform Description

The configurable platform features the StrongARM embedded processor core [150], which can be operated at a range of discrete frequencies from 59 Mhz to 206 Mhz, with corresponding voltage levels from 0.83 V to 1.5 V [154]. The overhead of frequency and voltage scaling is assumed to be constant at 150 μs [153]. The processor can access an instruction cache and a data cache, both of size 1.5 KBytes. The data cache is used as a dedicated on-chip data memory, and application data objects can be relocated between this on-chip memory and an external memory, using the cache locking mechanism (described in Section IV.B). Accesses to the on-chip memory and cache

are single cycle, and external memory access time is $50\ ns$. The overhead of relocating data objects depends on the size of the data and the memory access times.

Experimental Methodology

Optimized C implementations of the UMTS and WLAN security algorithms were compiled using the ARM C compiler *armcc* [155] (with maximum optimization) and were targeted to the platform described above. The experiments consider two variants of the example system. The first variant consists of a statically optimized configuration of the platform, where the platform is always operated at 206 Mhz and 1.5 V, with the on-chip memory configured as shown in row 1 of the tables in Figure IV.3. The second variant incorporates the dynamic platform management techniques described in Section IV.D, where the on-chip memory contents, and operating frequency and voltage are determined at run-time. The overhead of platform configuration was taken into account. The performance of the system under a given workload was measured using cycle-accurate instruction-level simulation of the platform architecture using *ARMulator* [155]. The power consumption was estimated using a cycle-accurate, software energy profiling tool, *JouleTrack* [154].

Off-line Task Characterization of UMTS and WLAN

Table IV.2 shows the results of performing the off-line task characterization step (Section IV.D.2) for the security processing tasks: UMTS ciphering and integrity, and WLAN encryption and checksum, for the platform described above. The data objects accessed by the UMTS task are its Stack (`Stack`), the Key Schedule (`KS`), the `S7` and `S9` lookup tables, and the UMTS frame (`Frame`). The WLAN data objects include its Stack (`Stack`), the State Table (`S-Table`), the CRC Table (`CRC-Table`), and the WLAN frame (`Frame`). The number of accesses per data item and the processing cycles requirement for UMTS depend on both the frame size (l) and the type of frame (for user frames, only the first parenthesized term is used, while for signalling frames, both the terms are used). The corresponding estimation formulae for WLAN depend only

Table IV.2: Characterization of the security processing tasks for the StrongARM based platform

Task	Data Items ($d_{i,j}$)	Max Size ($s_{i,j}$)	Estimated Number of Accesses ($n_{i,j}$)	Proc. Cycles (C_i)
UMTS	Stack	120	$(19.5\lceil l/8 \rceil+520.5)+(18.5\lceil l/8 \rceil+726.5)$	$(146.15\lceil l/8 \rceil +3176.71)+(131.95\lceil l/8 \rceil +4796.72)$
	KS	128	$(64\lfloor l\text{-}1/64 \rfloor+256)+(64\lfloor l+1/64 \rfloor+320)$	
	S7	256	$(48\lfloor l\text{-}1/64 \rfloor+96)+(48\lfloor l+1/64 \rfloor+144)$	
	S9	1024	$(48\lfloor l\text{-}1/64 \rfloor+96)+(48\lfloor l+1/64 \rfloor+144)$	
	Frame	1340	$(2\lceil l/8 \rceil+35)+(\lceil l/8 \rceil+40)$	
WLAN	Stack	96	272	$36l+5807$
	S-Table	256	$10l+1322$	
	CRC-Table	1024	$2l+40$	
	Frame	4654	$4l+54$	

on the frame size. These characterization tables are used by the platform management algorithms to select optimized platform configurations at run-time.

Impact of Dynamic Platform Management on Performance

This experiment evaluates improvements in security processing throughput made possible by the proposed techniques. For this experiment, we considered the "space" of possible data rate requirements of the UMTS and WLAN security processing tasks. We keep other characteristics, such as frame sizes and types constant. For each architecture, we measured the maximum pairwise achievable data rates (those achieved when the platform is maximally utilized). Figure IV.5 presents these results for the considered data rate space. The region on the left of each 100% CPU contour indicates data rate pairs that the corresponding architecture can satisfy.

From the figure, we observe that dynamic platform management enables the security processing system to satisfy a larger space of data rate combinations, compared to the statically configured system, facilitating large improvements in performance. For example, (in the absence of WLAN traffic) while the static configuration can achieve 3.1 Mbps UMTS throughput, the dynamic case can sustain a maximum data rate of

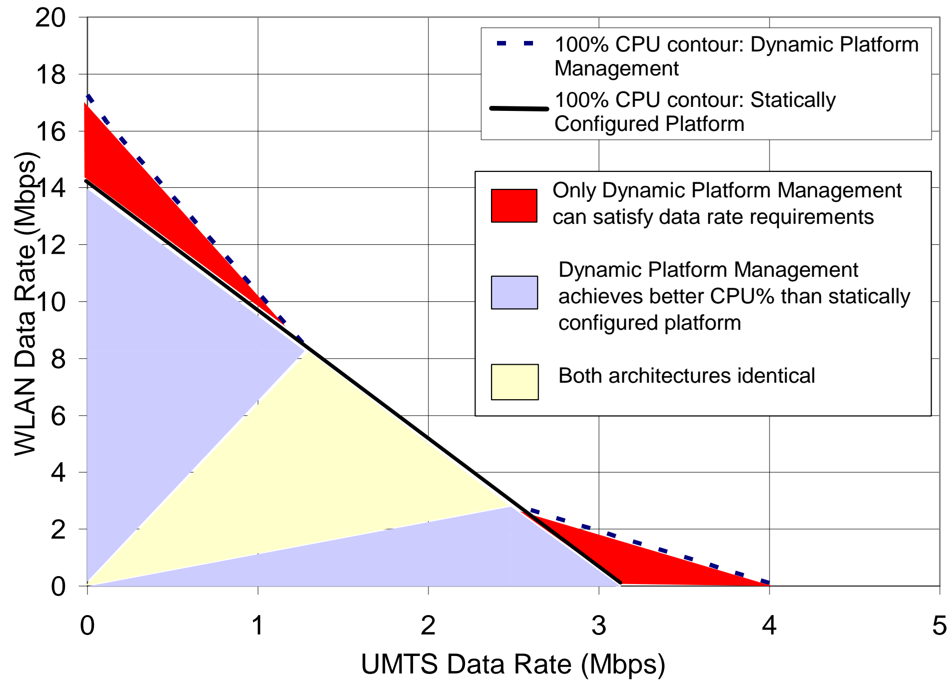


Figure IV.5: Dynamic platform management for the StrongARM based platform: space of achievable data rates and CPU efficiency

4.1 Mbps, a 33% improvement. Similarly, upto 21% improvements can be achieved for WLAN data rates. We exhaustively examined all possible static configurations of the platform, and found this configuration to be the one that meets the largest space of data rate requirements. By outperforming this static configuration, the dynamic platform management technique demonstrates that it can achieve significant performance gains over any static configuration.

Impact of Dynamic Platform Management on CPU Load

In practice, situations may often arise where data rate requirements are far lower than the maximum achievable values. We next demonstrate that even in cases where performance requirements can be met by the static configuration, it is still advantageous to use the dynamic platform management techniques. Figure IV.5 indicates portions of the data rate space, where both architectures can meet performance require-

ments (area to the left of the 100% contour for the static configuration). However, for a large fraction of this space (shaded grey), dynamic platform management results in fewer cycles being expended, hence improving CPU availability.

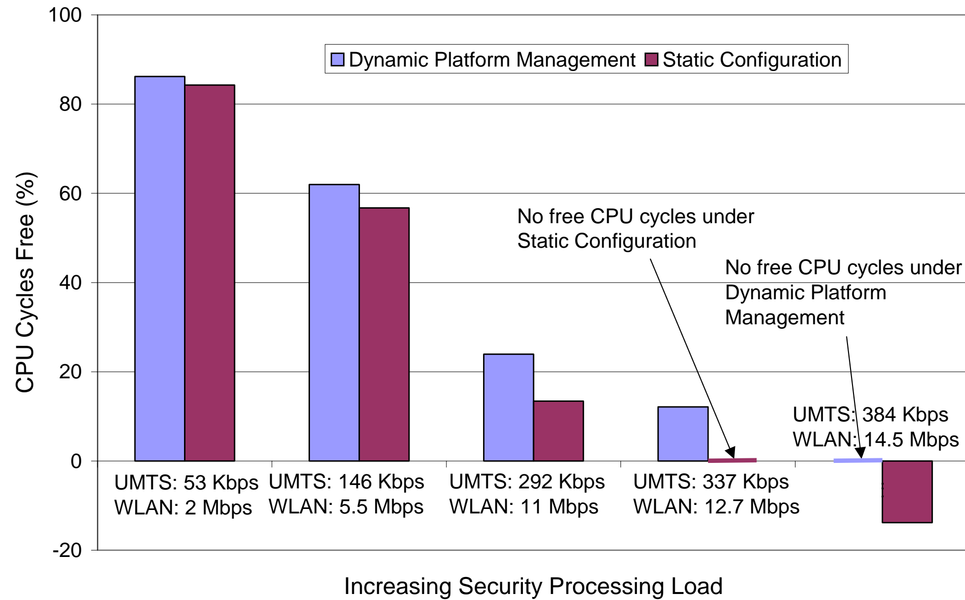


Figure IV.6: CPU fraction left-over from security processing under static configuration and dynamic platform management

To quantify this advantage, we performed an experiment where a set of discrete data rate pairs were considered. For each pair, we measured the fraction of CPU cycles left over from security processing, under the static configuration and with dynamic platform management. The results of these experiments, along with the data rate pairs used (in increasing order of imposed load), are presented in Figure IV.6. The figure shows that for all cases, the dynamic architecture makes more CPU cycles available than the static architecture. The effect is more significant at higher CPU loads. For example, in the case $\langle 337Kbps, 12.7Mbps \rangle$, while the static case "just" meets the requirements (CPU availability is 0%), in the dynamic case, the same requirements are met, with 12.14% of the CPU left over. The increased availability of the CPU (free cycles) can be

exploited to process other tasks, or reduce the frequency and supply voltage, and hence reduce power consumption.

The case $\langle 384Kbps, 14.5Mbps\rangle$ is of special interest. The results of Figure IV.6 show that the static architecture needs 14% more of the CPU than is available, hence cannot meet the requirements imposed by the data rates. However, for this case, dynamic platform management chooses an optimized platform configuration, which enables satisfying the imposed requirements.

In summary, Figures IV.5 and IV.6 demonstrate that the proposed dynamic platform management techniques (i) increase the space of maximum achievable performance of a configurable platform, and (ii) result in more efficient use of the CPU.

Impact of Dynamic Platform Management on Energy

In this experiment, we evaluated the power savings made possible with the proposed dynamic platform management techniques for the StrongARM based platform. For this experiment, we considered a dynamically varying workload consisting of varying tasks, data rate requirements, randomly varying frame size and types (Figure IV.7(a)). We compared the total energy consumed by the static configuration with that consumed with dynamic platform management while processing this workload. Figure IV.7(b) illustrates how the two platform parameters (frequency and on-chip memory) vary with time in the dynamic case, and Figure IV.7(c) plots a time profile of the total energy consumption. From Figure IV.7(c), we observe that the dynamically configured architecture achieves 59% energy savings compared to the static case. In both cases, all performance requirements were met.

It should be noted that the savings in energy consumption are due, in large part, to careful exploitation of the interdependence between on-chip memory configuration, CPU slack, and voltage scaling. To evaluate the benefit of our integrated approach, we measured the energy savings via traditional dynamic voltage scaling (DVS) [156], while keeping the on-chip memory configuration constant. From the cumulative energy plot corresponding to DVS in Figure IV.7(c) we observe that dynamic platform management

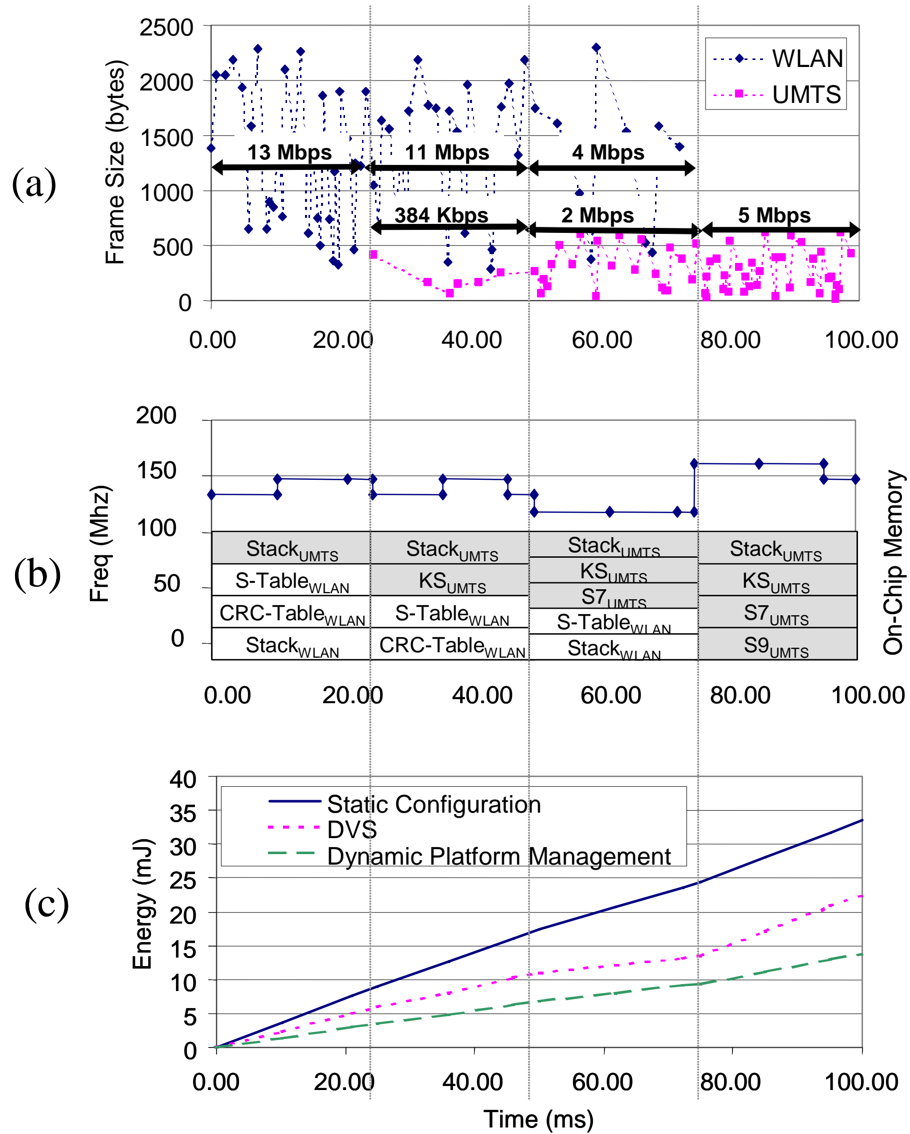


Figure IV.7: Energy savings using dynamic platform management: (a) varying UMTS and WLAN workload; (b) platform configuration sequence; and (c) cumulative energy profile

achieves 39% energy savings over DVS. These results demonstrate that the described platform management approach can be used to enhance system energy-efficiency over and above conventional techniques.

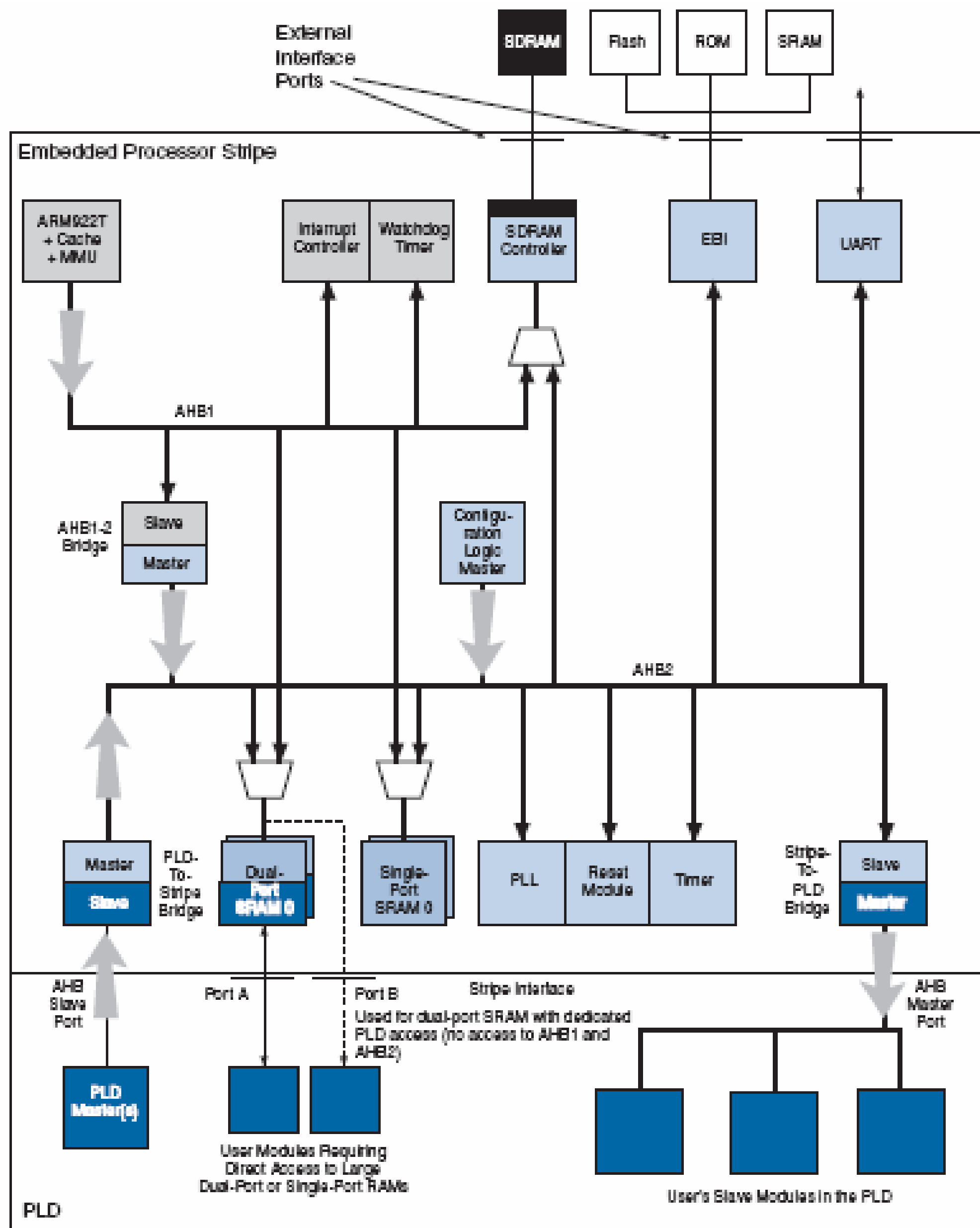
IV.E.2 Application to Altera Excalibur SOPC

Platform Description

The Excalibur chips from Altera [28] combine a processor, memory, logic and FPGA, enabling designers to integrate entire systems on a single device. Figure IV.8 shows a block diagram of the Excalibur platform [157]. It consists of two regions, (i) the stripe region, which contains hard-coded logic and memory, and (ii) the PLD region, which contains an FPGA. The stripe region features an ARM922T [146] embedded processor, with separate 8 KByte instruction and data caches, and memory management unit (MMU) support. The caches support locking, thereby enabling the data cache to be used as a dedicated data memory. It also contains a single port SRAM and a dual port SRAM. External interface ports are provided in the stripe to interface the platform with external memory (SDRAM, Flash, *etc*). Data relocation can be performed between external memory and the on-chip data memory using the cache locking mechanism, as described in Section IV.B. The platform can be operated at a maximum clock frequency of 200 Mhz, which can be changed at run-time via the phase lock loop block (PLL) using programmable registers. The bus architecture of the platform consists of two buses, AHB1 and AHB2, which conform to the AMBA high performance bus (AHB) specifications. The stripe also contains other peripherals, such as UART, timers, interrupt controller, bridges, *etc*. The PLD region contains programmable logic which can be used to implement upto a million gates.

Experimental Methodology

The UMTS and WLAN security processing system was implemented on the Excalibur EPXA1 development board from Altera [158]. Figure IV.9 shows a photo-



Source: Altera Corp.

Figure IV.8: Block diagram of the Altera Excalibur SOPC

graph of the experimental setup. The Excalibur chip on the board is connected to a Flash memory, which is used during system bootup, and an SDRAM, which is used during program execution. The board is interfaced with a host computer to control system execution and display execution statistics, using a serial connection via the on-chip UART. The platform's supply voltage is fixed, and therefore does not allow voltage scaling. [1] The instruction and data caches of the ARM922T processor are large enough to contain all the code and data of the security processing programs. Therefore, in order to illustrate dynamic data relocation for this system, only 1.5 KBytes of each cache was used, the rest remaining unused. The data cache was used as a 1.5 KByte on-chip data memory using cache locking. The PLD region of the platform was not utilized for this experiment. We next describe how the dynamic platform management framework, described in Section II.C, was implemented for this system.

Figure IV.9: Experimental setup for demonstrating dynamic platform management using the Altera Excalibur development board

System boot code and an operating system (OS) were implemented for providing an execution environment for the applications. The OS is responsible for scheduling the executing tasks (in this case, UMTS and WLAN security tasks) on the processor,

[1] Due to this reason, we could not demonstrate energy savings under dynamic platform management for this platform.

using the earliest deadline first (EDF) algorithm. It is also responsible for handling interrupts and for communicating with the host computer. The OS code and system data structures (process tables, page tables, *etc.*) are mapped to the on-chip single and dual port SRAMs. “Adapters” are provided by the OS for configuring the platform frequency, and for performing data relocation between the external SDRAM (where the applications’ data objects are mapped) and the on-chip data memory. The adapters can be invoked via system calls. The frequency adapter changes the platform frequency by writing to a set of registers in the PLL block. The data relocation adapter relocates application data using the data cache locking mechanism via cache control registers, as described in Section IV.B. The UMTS and WLAN security algorithms were implemented in C. The applications’ code and data was mapped to the external SDRAM. The UMTS and WLAN tasks process frames from respective frame buffers, and change their data rate requirements periodically. The programs were instrumented to provide information about run-time frame characteristics and data rate requirements to the dynamic platform management layer using APIs implemented as system calls. The platform management layer is tightly integrated with the OS. On being invoked, it uses the information provided by the UMTS and WLAN applications to (i) select the placement of the UMTS and WLAN data objects between the external SDRAM and the on-chip data memory, and (ii) select the operating frequency for the platform, using the algorithms described in Section IV.D. It then applies the selected configuration by invoking the adapters via system calls. The system execution statistics (data rate achieved for each application, number of missed deadlines, *etc.*) are collected by the OS, and periodically displayed on the host computer.

Impact of Dynamic Platform Management on Performance and CPU Load

We evaluated the space of achievable data rates for the UMTS and WLAN security processing tasks, executing on the Altera Excalibur platform described above, under two variants of the system: (i) a statically optimized platform configuration, where the clock frequency is set at 200 MHz, and the on-chip memory is configured as shown

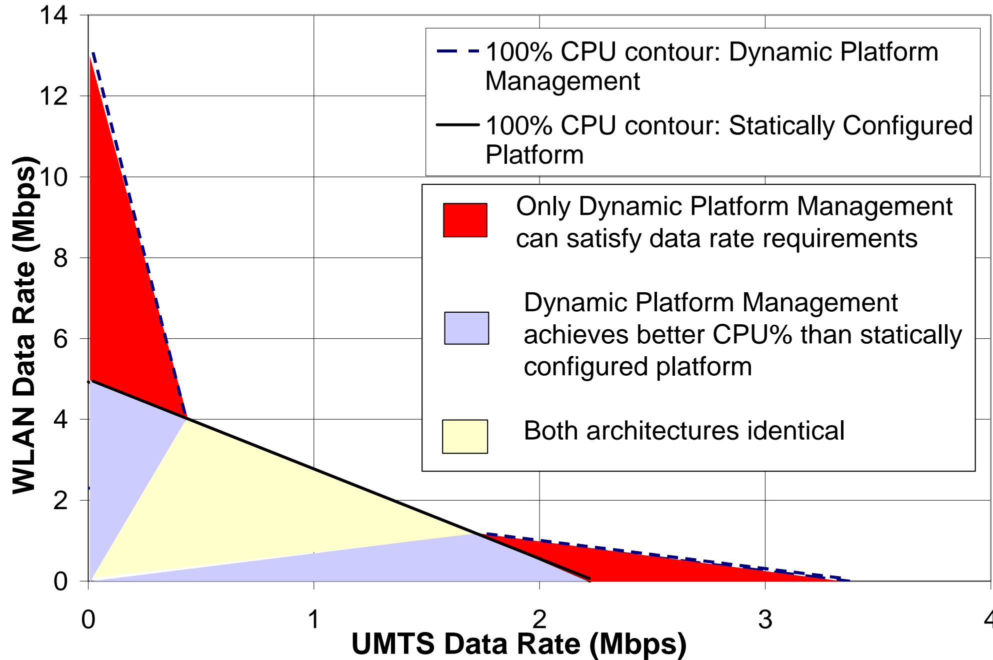


Figure IV.10: Dynamic platform management for Altera Excalibur: space of achievable data rates and CPU efficiency

in row 1 of the tables in Figure IV.3 (this was evaluated to be the best static configuration since it satisfies the largest space of data rate requirements among all static configurations), and (ii) the platform operating under dynamic platform management. Figure IV.10 shows the results of this study. From the figure, we observe that the platform operating under dynamic platform management can satisfy a much larger set of data rate requirements compared to the static case. For example, when WLAN executes alone, the dynamically configured platform can achieve a data rate of 12.9 Mbps compared to 4.95 Mbps for the static case, a 160% improvement, while when UMTS executes alone, platform management enables data rate improvements of up to 51%.

Figure IV.3 also shows regions of the data rate space (shaded gray) where although the static configuration can satisfy the data rate requirements, it is still beneficial to operate the platform under dynamic platform management. This is because in these regions, dynamic platform management achieves better CPU efficiency (fewer CPU cy-

cles) compared to the static case. This CPU slack can be used to either support more applications, or aggressively reduce the platform frequency and voltage for large energy savings.

IV.F Conclusions

In this chapter, we proposed SoC platforms featuring multiple dynamic configurability options, namely, fine-grained frequency and voltage scaling, and flexible relocation of application data. We illustrated the importance of considering the interaction between these configurability options, and presented integrated dynamic platform management techniques for the run-time, application-specific configuration of such platforms. Using a dual-access security processing system as an example, we evaluated the benefits of the proposed approach by implementing it for two different configurable platforms, a StrongARM based platform, and the Altera Excalibur development board. Experiments demonstrate that the proposed platforms coupled with dynamic platform management enable significant improvements in application performance, CPU utilization, and energy efficiency.

The text of this chapter, in part, is based on material that has been published in the International Conference on Computer-Aided Design, 2003, and material submitted to the IEEE Transactions on Computer-Aided Design of Circuits and Systems. The dissertation author was the primary researcher and author, and the coauthors listed in these publications collaborated on, or supervised the research that forms the basis for this chapter.

V

Dynamic Management of SoC Platforms with Dynamic Data Relocation and Reconfigurable Bus Architectures

V.A Introduction

The design of increasingly complex SoC platforms is being driven by the convergence of multiple, diverse applications onto a single device (*e.g.*, wireless handsets). The characteristics of the processing workload imposed on such platforms may exhibit large dynamic variation, depending on which functions are being exercised at any given time, and the variations in their individual performance requirements. Consequently, as demonstrated in this thesis, platform architectures that are statically customized for average or worst case requirements often fail to meet desired system-level design goals in terms of performance and/or energy efficiency, motivating the need for dynamically configurable platforms. In the previous chapter, we illustrated the benefits of integrated run-time management for platforms featuring frequency and voltage scalable processors and dynamic data relocation in memory. This chapter focuses on integrated dynamic

management for platforms featuring dynamic data relocation in memory and configurability in the on-chip communication architecture.

V.A.1 Chapter Overview

In this chapter, we propose dynamically configurable SoC platforms featuring configurability in the on-chip communication and memory architecture, two key subsystems that significantly influence overall system performance and energy efficiency [102, 159]. We motivate the importance of integrated management for such platforms by analyzing a hybrid Viterbi-Turbo decoding system that integrates decoders corresponding to the UMTS (3G cellular) and IEEE 802.11a (wireless LAN) standards for converged handsets [160]. We show that approaches in which the mapping of application data objects to regions of the memory address space, and configuration of the on-chip communication architecture, are performed statically, can lead to substantial performance loss. In addition, we show that the mapping of data to memory directly influences the on-chip communication traffic profile, which in turn affects the choice of communication architecture configuration. We illustrate how this interdependence affects design approaches in which the data placement is customized independently of the communication architecture (or vice versa). We propose a dynamic platform management methodology that addresses the configuration of these subsystems in an integrated fashion. The methodology consists of two phases, off-line (static) characterization phase, and run-time platform management phase. In the off-line phase, the application data rate space is partitioned into different regions, each of which is mapped to an appropriate bus and memory configuration. In the run-time phase, this off-line information is used to select optimized platform configurations depending on current application requirements. The proposed approach was evaluated on a mixed HW/SW implementation of the Viterbi-Turbo system. We observed performance gains of up to 32% compared to the best statically optimized design, with negligible hardware overhead.

The rest of this chapter is organized as follows. In Section V.A.2, we describe related work. In Section V.B, we describe the hardware support required to provision for dynamic data relocation and bus reconfiguration in SoC platforms. In Section V.C, we present illustrative examples that motivate our work. In Section V.D, we describe the proposed dynamic platform management methodology. In Section V.E, we present experimental results that evaluate the application of the proposed approach to the design of an integrated Viterbi-Turbo Decoder, and we conclude in Section V.F.

V.A.2 Related Work

The increasing importance of on-chip communication has in recent years led to the development of system-level techniques for customizing the communication architecture to application traffic characteristics as discussed in Chapter III [91, 113, 114, 161]. However, these techniques largely assume that the on-chip traffic characteristics are given, *i.e.*, they do not explore how the on-chip traffic is influenced by the placement of data across the various on-chip memories. Correspondingly, most techniques that optimize data placement and/or the memory organization, do so without assuming any configurability inherent in the communication architecture [159, 80, 81]. The interdependence between communication architectures and memory architectures has been recently studied in [162, 163, 164]. All of these techniques perform simultaneous exploration of the joint design space. However, they focus on statically optimizing the system architecture to specific application characteristics, and do not address the need for dynamic configuration.

The interdependence between bus protocols and other system-level design problems, such as hardware/software co-design was studied in [165], although they too focus on static optimization. To the best of our knowledge this is the first work that attempts to perform integrated, dynamic management of data placement and communication architecture configuration. While in this work, we consider reconfigurable bus-based architectures, our ideas could be extended to network-on-chips that provide support for dynamic configurability in the network topology and/or protocols [115]. Fi-

nally, we note that many of the static optimization techniques mentioned above can be used within our proposed flow to derive optimized system configurations for different application requirements. Thus, our platform management methodology is complementary to these techniques.

V.B Configurable Platform Architecture

In this section, we describe the two dynamic configurability options that we study in this chapter, namely data relocation and bus reconfiguration, and describe the hardware support required to provision for them.

Dynamic data relocation (described in Section II.B.3) refers to the ability to change the location or placement of application data structures or objects among the platform memories after they have been allocated (*i.e.*, at run-time). Fast data relocation can be performed using direct memory access (DMA). After relocating a data object, all future references to it must find it at its new location. For references from processors, this is ensured by changing the virtual to physical address mapping of the relocated data in the processor's page table, thereby enabling data relocation at the granularity of a page. For references from platform hardware components, "base-plus-offset" addressing modes must be used for each relocatable data object. On data relocation, the corresponding base pointer values are updated to point to the new base address. Since data relocation is handled differently for processors and other hardware components, an integral number of pages should be allocated to relocatable data objects that are accessed by both. The overhead of data relocation depends on the relocated data size, memory access times, bus characteristics (*e.g.*, burst modes), and bus speeds.

Dynamic bus reconfiguration (described in Section II.B.4) refers to the ability to configure the underlying bus architecture of the platform to system-level changes in communication traffic characteristics. In general, different aspects of the bus architecture can be reconfigured, such as the bus protocol (*e.g.*, arbitration priorities) or the topology (*e.g.*, mapping of components to bus segments). In this work, we focus on

bus topology reconfiguration through *dynamic bridge by-pass*, which was described in Chapter III. Briefly, bridge by-pass enables the internal logic of bridges that connect different bus segments to be "by-passed" at run-time, thereby fusing two or more bus segments into a single shared bus. A Bus Reconfiguration Unit is responsible for enabling or disabling by-pass for different bridges when so instructed by system software, by sending appropriate control signals to the respective bus arbiters and bridges. The exact penalty of bus reconfiguration depends on the number of pending bus transactions, but is typically on the order of tens of bus cycles. Details of the bridge by-pass mechanism are described in Chapter III.

V.C Motivational Example

In this section, we illustrate, using an integrated IEEE 802.11a Viterbi and UMTS Turbo decoder design, that the problems of placing data in memory and configuring the on-chip bus are interdependent, and motivate the need for joint dynamic data relocation and bus reconfiguration.

V.C.1 Case Study: Integrated Viterbi-Turbo Decoder Design

Forward-error correction (FEC), also known as channel coding, is used to improve the capacity of a channel by adding redundant information to the data being transmitted. Viterbi coding [166] is a popular FEC technique used in a wide variety of wireless standards (*e.g.*, IEEE 802.11a [167], IEEE 802.16 [168], WCDMA (UMTS) [136]). Figure V.1(a) illustrates the tasks that constitute Viterbi decoding [169]. The decoder receives quantized soft bits (multiple bits to represent the confidence in a bit being 0 or 1), IN_V, corresponding to the noise-contaminated received signal. The "de-puncture" task inserts dummy zero values in place of the bits that were "punctured" at the transmitter (puncturing is a process of omitting some encoded bits in the transmitter to increase the coding rate). The Viterbi decode task then processes these bits using the Viterbi algorithm to generate the output decoded bits, OUT_V.

Turbo coding is another FEC technique that has received considerable attention in recent years due to its near Shannon capacity performance [135], and has been included in the specifications for third-generation cellular standards (*e.g.*, WCDMA (UMTS) [136], cdma2000 [137]). Figure V.1(b) shows the tasks involved in Turbo decoding [169]. The decoder receives quantized soft bits, X_T, $Z_{1,T}$ and $Z_{2,T}$, corresponding to the noise-contaminated received signal. Turbo decoding consists of two identical recursive systematic convolutional (RSC) decoding tasks linked together by an interleaving and a de-interleaving task. As indicated by the feedback path, Turbo decoding operates in an iterative manner. In each iteration, the first convolutional decoding task generates soft outputs ($L_{e1,T}$) about the likely values of the bits to be decoded in terms of the log-likelihood ratios (LLR - logarithm of the ratio of the probability of the bit being 1 to the bit being 0). These values are then interleaved and input to the second decoding task ($L'_{e1,T}$), which then generates another set of LLR values ($L_{e2,T}$), which are fed back to the first decoding task after de-interleaving them ($L'_{e2,T}$). After a number of such iterations (typically 8 for low bit error rate), a hard (0 or 1) decision about the value of each bit is output (OUT_T).

The integrated Viterbi and Turbo decoder design is motivated by the emergence of converged handsets capable of operating over multiple air interfaces simultaneously [148, 160]. The Viterbi decoder implements the specifications for IEEE 802.11a [167], while the Turbo decoder implements the specifications for UMTS (WCDMA) [136]. The data rate requirements for Viterbi and Turbo decoding may vary over time depending on application data rate, signal strength, number of users, *etc*. Figure V.1(c) shows the mapping of the Viterbi and Turbo tasks to components in the design. The Viterbi de-puncture task and the Turbo interleave and de-interleave tasks are implemented in software on the embedded processor (ARM946E-S [122]). The Viterbi decode task and the Turbo convolutional decode task are implemented as dedicated hardware (Viterbi Unit and Turbo Unit, respectively). The Viterbi coding rate is set to 2/3, the number of Turbo iterations is set to 8, and the block size for both Viterbi and Turbo decoding is set to 1024 bits. The bus architecture is based on the AMBA AHB

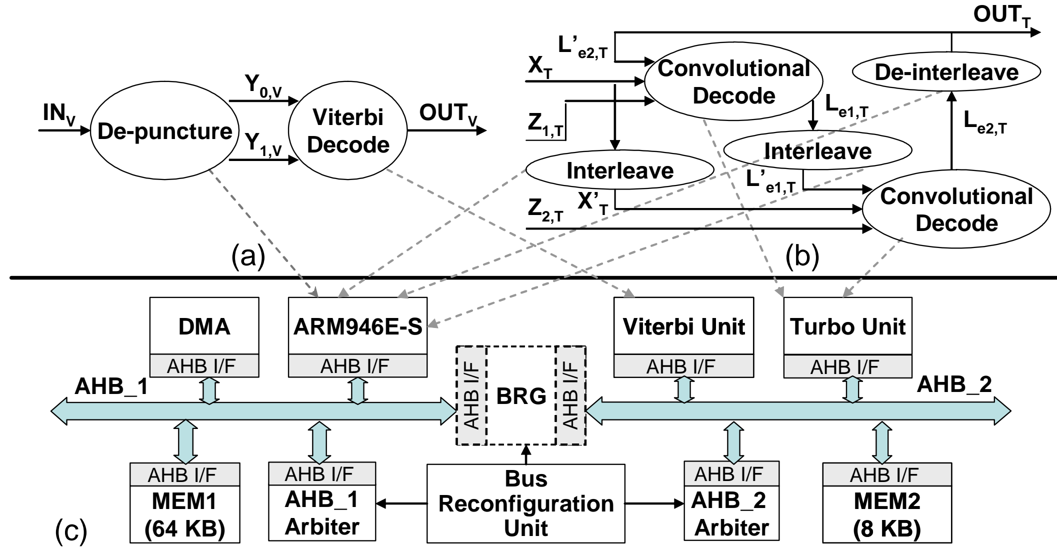


Figure V.1: Integrated Viterbi-Turbo decoder design: functional specification of (a) Viterbi decoder, (b) Turbo decoder, and (c) mapping of functional blocks to integrated decoder design

bus standard [22] and consists of two bus segments, AHB1 and AHB2, connected by a two-way bridge, BRG. The platform also has two SRAM memory components; MEM1, of size 64 KBytes, is connected to AHB1, and MEM2, of size 8 KBytes, is connected to AHB2. Table V.1 shows the different data objects used by the Viterbi and Turbo applications and their sizes. For example, $STable_V$ is an internal data structure used by the Viterbi decode task, and requires 5120 Bytes.

The design was enhanced with the two dynamic configurability options described in Section V.B: (i) dynamic data relocation, which enables Viterbi and Turbo data objects to be relocated between MEM1 and MEM2, and (ii) dynamic bus reconfiguration, which allows BRG to be by-passed at times, thereby enabling switching between a multiple bus architecture and a single shared bus architecture. The Viterbi-Turbo decoder was optimized using dynamic platform management, details of which are provided in Section V.D.

Table V.1: Viterbi and Turbo decoding data objects

App.	Data Objects	Purpose	Size
Viterbi	IN_V	Soft Input bits	1536 bytes
	$Y_{0,V}$	De-punctured bits 1	1024 bytes
	$Y_{1,V}$	De-punctured bits 2	1024 bytes
	OUT_V	Decoded Output bits	512 bytes
	$STable_V$	State History bits	5120 bytes
Turbo	X_T	Soft Input Systematic bits	1024 bytes
	X'_T	Interleaved Systematic bits	1024 bytes
	$Z_{1,T}$	Soft Input Parity bits 1	1024 bytes
	$Z_{2,T}$	Soft Input Parity bits 2	1024 bytes
	$L_{e1,T}$	LLR1 bits	1024 bytes
	$L'_{e1,T}$	Interleaved LLR1 bits	1024 bytes
	$L_{e2,T}$	LLR2 bits	1024 bytes
	$L'_{e2,T}$	De-interleaved LLR2 bits	1024 bytes
	OUT_T	Decoded Output bits	512 bytes

V.C.2 Illustrative Examples

In the following examples, we analyze the Viterbi-Turbo decoder design to demonstrate the benefit of the proposed dynamic platform management techniques in terms of system performance. System performance was evaluated using cycle-accurate co-simulation (details of the experimental methodology are described in Section V.E.1).

Example 1

We first consider the application scenario where only the Viterbi decoder is executing on the architecture shown in Figure V.1(c). Table V.2 shows the maximum data rates achieved under different combinations of data placement and bus configuration. From the table, we observe that, under a single shared bus (*i.e.*, when BRG is bypassed), all data placements give the same performance (since both MEM1 and MEM2 are on the same bus, it doesn't matter which data objects each contains). However, under a multiple bus architecture, data placement 1 shown in Table V.2 achieves the highest data rate. This clearly shows that the optimal placement of data in memory depends on

the underlying bus topology. Also, if data placement 1 is used, the multiple bus architecture performs the best, while if data placement 4 is used, the single shared bus gives a higher data rate. This shows that for best performance, the topology of the underlying bus architecture should take into account the placement of data in memory. ■

Table V.2: Viterbi decoding data rates under different data placement and bus configurations

Data Placement			Bus Topology	
#	**MEM1**	**MEM2**	**Single Shared Bus**	**Multiple Bus**
1.	IN_V	$Y_{0,V}$, $Y_{1,V}$, $STable_V$, OUT_V	30.1591 Mbps	52.4926 Mbps
2.	IN_V , OUT_V , $Y_{0,V}$	$Y_{1,V}$, $STable_V$	30.1591 Mbps	46.3901 Mbps
3.	IN_V , OUT_V , $Y_{0,V}$, $Y_{1,V}$	$STable_V$	30.1591 Mbps	40.7515 Mbps
4.	IN_V , OUT_V ,$Y_{0,V}$, $Y_{1,V}$, $STable_V$	-	30.1591 Mbps	9.9071 Mbps

The above example, therefore, illustrates that the placement of data in memory and the bus configuration are interdependent, and should be jointly optimized. We next make the case for this joint optimization to be performed dynamically.

Example 2

We consider the simultaneous execution of both Viterbi and Turbo decoding on the architecture of Figure V.1(c). Figure V.2 illustrates the Viterbi and Turbo decoding "data rate space", which consists of different combinations of data rate requirements for each of the two applications. Each point in this data rate space consists of a specific data rate for Turbo decoding that must be supported, along with the data rate to be supported for concurrent Viterbi decoding. We examined six candidate configurations of the bus architecture and data placement in memory (see Table V.3 in Section V.E.2) and identified the regions in the data rate space that are achievable for each of them. For some sample points in the data rate space, Figure V.2 shows the platform configurations under which they are achievable. For example, when the decoding requirement for Viterbi is 10 Mbps and for Turbo is 1 Mbps, multiple platform configurations (C_1, C_3,

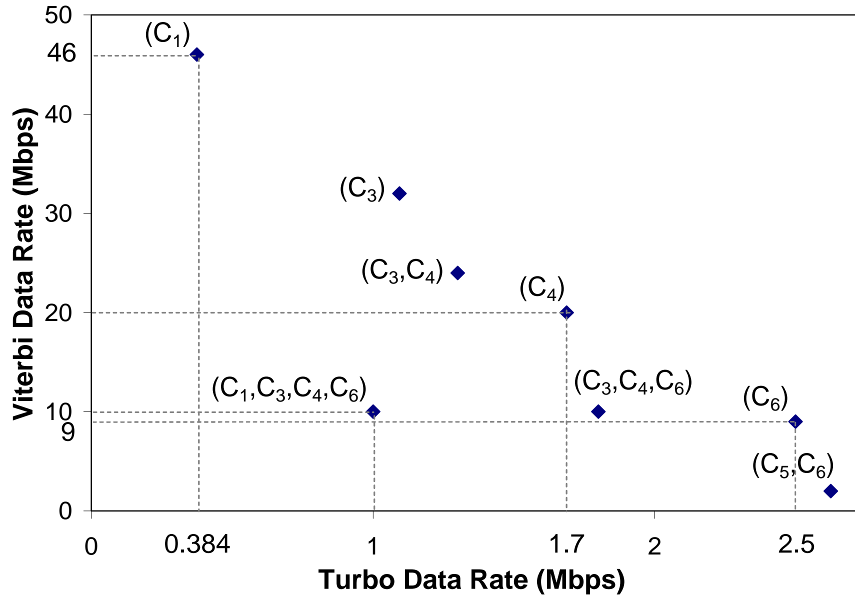


Figure V.2: Viterbi and Turbo decoding data rate requirements and platform configurations that can satisfy them

C_4 and C_6) can satisfy it. However, *only* configuration C_1 can achieve a Viterbi data rate of 46 Mbps and a Turbo data rate of 384 Kbps, while only configuration C_4 can achieve a Viterbi data rate of 20 Mbps and a Turbo data rate of 1.7 Mbps. Configurations C_1 and C_4 differ in the way data is placed in memory (Table V.3), indicating that dynamic data relocation can help satisfy a larger space of performance requirements for the design. We also observe that when the decoding requirement for Viterbi is 9 Mbps and for Turbo is 2.5 Mbps, only configuration C_6, which employs a single shared bus, can satisfy it. Finally, if the application requirements change at run-time from (384 Kbps, 46 Mbps) to (1.7 Mbps, 20 Mbps), the only way to satisfy both requirements is to dynamically change the configuration from C_1 to C_4. ∎

The above example illustrates that dynamic data relocation and bus reconfiguration enable the design to satisfy a larger set of performance objectives. This is because data relocation enables the placement of data in memory to be optimized to best suit the current requirements of the executing applications, while adapting the bus configuration enables it to be better matched to the resulting on-chip communication traffic profile.

In summary, the above examples motivate the need for integrated, dynamic configuration of the on-chip communication and memory architectures. We next describe our dynamic platform management methodology based on such an approach.

V.D Dynamic Platform Management Methodology

In this section, we first describe the problem of run-time selection of optimized data placement and bus configuration for SoC platforms, and present an overview of our dynamic platform management methodology to address it. We then describe the steps in the methodology in detail.

V.D.1 Problem Description and Methodology Overview

We consider a partitioned and mapped SoC platform architecture whose components are mapped to a set of bus segments interconnected by bridges. The platform executes a set of applications, $A_1, A_2, ..., A_N$, with corresponding time-varying data rate requirements, $DR_1, DR_2, ..., DR_N$. Each application A_i has a set of relocatable data objects. The platform is enhanced to support the relocation of data objects in memory, and reconfiguration of the bus architecture through dynamic by-pass of some or all of the bridges it contains. The problem of dynamic platform management is to select the optimized placement of the application data objects in memory, and the bus configuration (*i.e.*, which bridges are to be by-passed) at run-time, such that the data rate requirements of all the applications can be satisfied.

The proposed methodology for addressing this problem consists of two phases: (i) off-line characterization phase, during which the applications' data rate space is partitioned and an optimized platform configuration for each partition is determined, and (ii) run-time platform configuration phase, during which optimized platform configurations are selected and applied depending on the current data rate requirements of the executing applications, based on the off-line generated information. We next describe each of these phases in further detail.

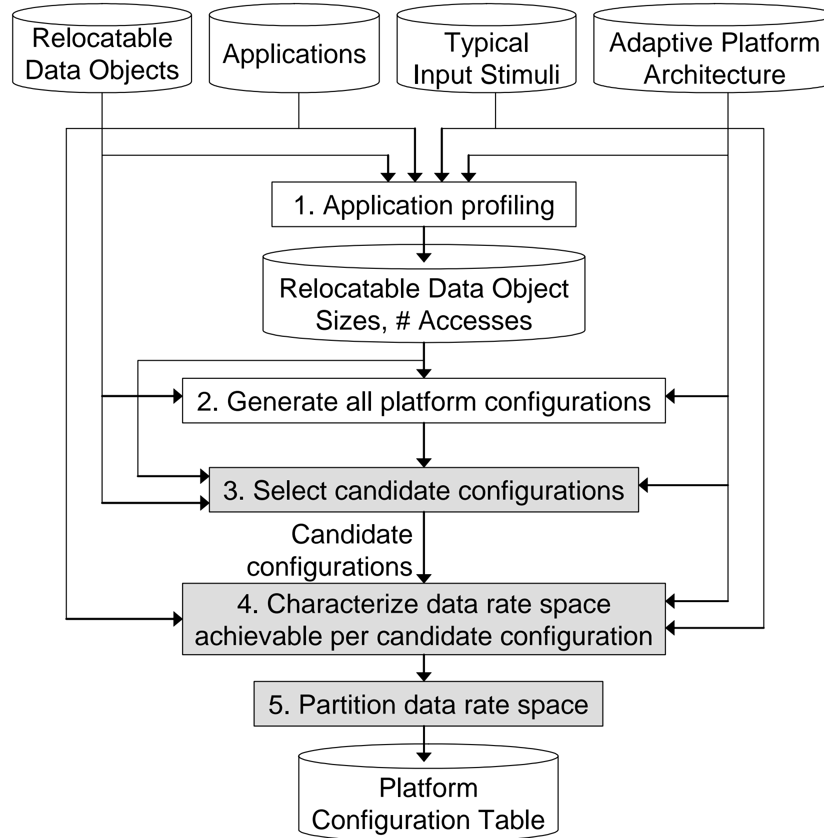


Figure V.3: Off-line Characterization Phase

V.D.2 Off-line Characterization Phase

Figure V.3 shows the steps in the methodology in the off-line phase. Each application A_i is associated with a set of relocatable data objects $D_i = \{d_{i,1}, d_{i,2}, ..., d_{i,M}\}$. We first determine their sizes $S_i = \{s_{i,1}, s_{i,2}, ..., s_{i,M}\}$. We also estimate, for each $d_{i,j}$, the average number of accesses to it $n_{i,j,k}$, from each platform component P_k, through simulation using typical input stimuli (step **1**). We then generate an exhaustive list of all possible platform configurations $C_l = \langle M_m, B_n \rangle$, where M_m represents the data placement configuration, and B_n the bus configuration, with the only constraint that, for each memory, the set of data objects allocated to it should not exceed its size (step **2**). This configuration space is then pruned to a subset of candidate platform configurations, such that each can potentially cover a unique region of the data rate space of the applications (step **3**). Next, each candidate configuration is analyzed to estimate the set of points in

the data rate space that it can achieve (step **4**). Finally, the application data rate space is partitioned among the candidate configurations (step **5**). The result of the off-line phase is a Platform Configuration Table, which lists, for each data rate partition, the associated optimized platform configuration. This table is used to perform run-time platform configuration as described in Section V.D.3. We next describe the highlighted steps in the off-line phase in further detail.

Selection of Candidate Configurations

In this step (step **3** in Figure V.3), we prune the exhaustive platform configuration space to a subset of candidate configurations that can potentially cover a unique region of the application data rate space. This is performed as follows. For each application A_i, under each platform configuration $\langle M_m, B_n \rangle$, we compute the total number of "cross-bridge" accesses $T_{i,m,n}$, to the application's relocatable data objects D_i. A cross-bridge access refers to an access by a platform component to a data object across a bridge. This is given by

$$T_{i,m,n} = \sum_{j=1}^{M} \sum_{P_k} n_{i,j,k} \times BRG_{i,j,k} \tag{V.1}$$

where $BRG_{i,j,k}$ is the number of intermediate bridges between component P_k, and data object $d_{i,j}$, and depends on the platform configuration. Next, for each configuration $\langle M_m, B_n \rangle$, the total number of cross-bridge accesses $T_{i,m,n}$, for each application A_i, are compared to the corresponding number of cross-bridge accesses $T_{i,r,n}$, under all other data placement configurations M_r, but for the *same* bus configuration B_n. If there exists a configuration for which $T_{i,r,n} \leq T_{i,m,n}$ for *all* applications, then the configuration $\langle M_m, B_n \rangle$ is discarded; otherwise, it is chosen as a candidate configuration. This is because for a given bus configuration, the number of cross-bridge accesses has a significant impact on the performance. Therefore, the data rate space achieved under data placements that result in a larger number of cross-bridge accesses for all applications, would be contained within the data rate space achieved by those that result in

fewer cross-bridge accesses. Note that, we do not prune configurations across different bus configurations at this stage. This is because comparing different bus configurations would require a detailed control flow analysis of the applications, since the bus configuration affects the system concurrency. Note that, redundant candidate configurations will ultimately be discarded by the methodology as described later.

Characterization of Data Rate Space Achievable under each Candidate Configuration

Next, we determine the data rate space achievable under each candidate configuration (step **4** in Figure V.3). To illustrate this, let us consider an example system executing two applications, A_1 and A_2. Under a particular platform configuration C_1, let the data rate achieved by A_1 when it alone executes be DR_{A_1,C_1}, and by A_2 when it alone executes be DR_{A_2,C_1} (Figure V.4(a)). If we assume that only one application can execute on the system at any point of time (*i.e.*, A_1 and A_2 cannot execute concurrently), then the maximum data rate combinations that can be achieved for A_1 and A_2 under configuration C_1 is given by the dark dotted line in Figure V.4(a), and the data rate space achieved is given by the triangle formed by this line and the axes. Now, if we assume that both A_1 and A_2 can execute fully concurrently on the system without any resource contention (bus conflicts, hardware resource conflicts, *etc.*), then the data rate space under C_1 is given by the rectangle formed by the dark dashed lines and the axes, since both applications can execute without any interference from the other application. In reality, however, the above two cases give the worst possible and best possible performance of the system, respectively, and the actual data rate space achieved will lie somewhere between these two extremes, shown by the region between the dark solid curve and the axes (Figure V.4(a)). This line lies closer to the dark dotted line if there is not much concurrency available in the system, and closer to the dark broken lines if the system has more concurrency.

In order to exactly characterize this data rate space would require exhaustive performance analysis of the system under each candidate configuration and for all pos-

sible combinations of application data rate requirements. This is because whether a particular data rate combination can be achieved for a set of applications under a given configuration of the system, depends on the fine-grained control flow and data access profile of the individual applications. Clearly, such exhaustive simulation would be infeasible for most systems. However, the objective of this step is to enable a good partitioning of the overall application data rate space (step **5** in Figure V.3), for which even coarse-grained approximation of the data rate space under each candidate configuration is sufficient. Therefore, we propose a technique to approximate the data rate space achievable by each candidate configuration based on a limited number of detailed simulations.

To illustrate approximate data rate space computation, let us again consider the above two application example system. Figure V.4(a) shows the exact data rate space achieved under candidate configuration C_1 (the region between the dark solid curve and the axes). To approximate this curve, we obtain only a limited number of points on the exact curve through detailed simulation, and fit a spline curve that passes through the obtained points. For this, the platform is simulated under configuration C_1 using typical input stimuli, for (i) each application executing alone, resulting in points (DR_{A_1,C_1},0) and (0,DR_{A_2,C_1}) in Figure V.4(a), and (ii) with both applications executing concurrently processing as fast as possible, resulting in point (DR'_{A_1,C_1}, DR'_{A_2,C_1}) in Figure V.4(a). Next, we use a quadratic parametric spline curve fitted to these three points, resulting in the light solid curve for C_1. This parametric curve is represented by equations of the form $x(t) = a_2t^2 + a_1t + a_0$ and $y(t) = b_2t^2 + b_1t + b_0$, where $x(t)$ and $y(t)$ are points on the curve, the co-efficients define the shape of the curve, and t is the parameter. For the points on the Y-axis and X-axis, the parameter value is set to 0 and 1, respectively, while for the third point it is set to $d_y/(d_x + d_y)$, where d_x and d_y are the Euclidean distance of the third point from the point on the X-axis and Y-axis, respectively. In our experiments, we found that such curves can approximate the actual data rate space well. Note that, more accurate characterization can be performed using more simulations of the platform, resulting in a larger number of points to fit the curve.

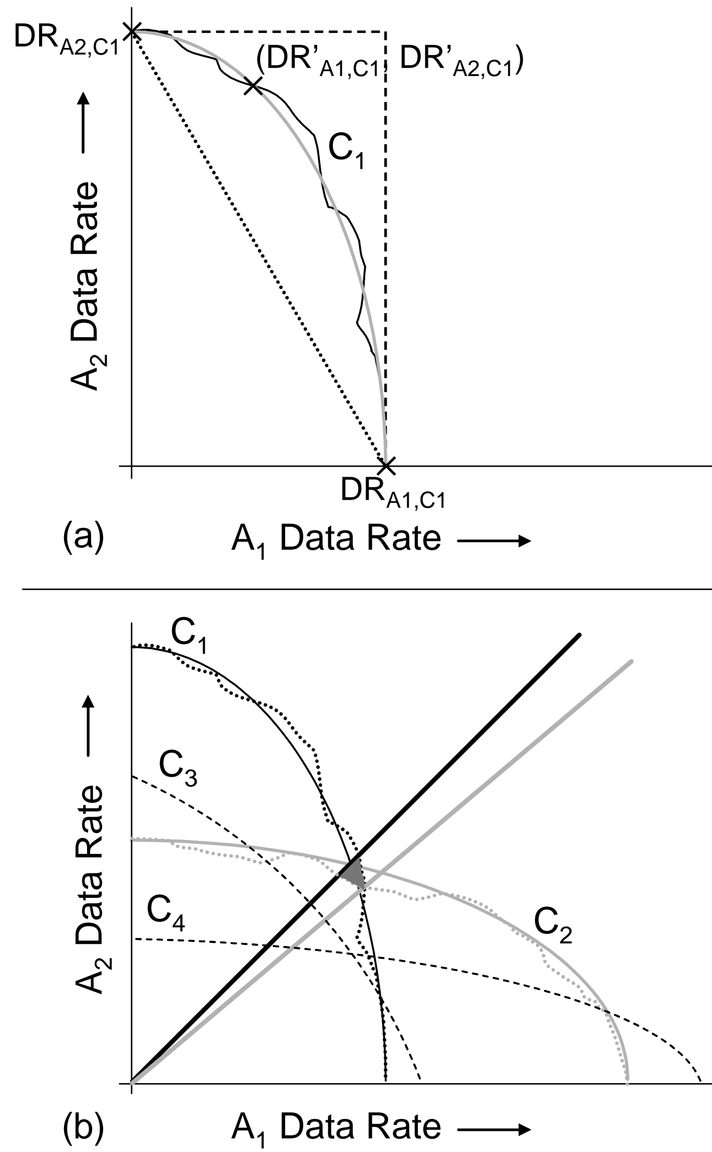


Figure V.4: For two applications: (a) data rate space characterization, and (b) overall application data rate space partitioning

In that sense, the proposed approximation technique is scalable. Other performance analysis techniques [163, 164] can also be used in this step to speedup simulation.

The above procedure is repeated for all candidate configurations, to characterize their respective achievable data rate spaces.

Partitioning of the Data Rate Space

Finally, the overall application data rate space is partitioned among the candidate configurations, such that in each partition, the associated platform configuration is best able to satisfy the applications' data rate requirements. To illustrate this, we consider Figure V.4(b), which shows the exact data rate space (dark dotted curve) and our approximated data rate space (dark solid curve) achieved under platform configuration C_1 for the above example system (from Figure V.4(a)). Figure V.4(b) also shows the exact (light dotted curve) and our approximated (light solid curve) data rate space achieved under another candidate configuration C_2. The line joining the intersection of the approximated curves for C_1 and C_2, and the origin (dark solid line) partitions the data rate space, such that for points that lie above this line, it is preferable to choose C_1, while for points below this line, it is preferable to choose C_2. Figure V.4(b) also shows the ideal partitioning of the data rate space derived based on the exact data rate curves (light solid line). The shaded area in the figure indicates the data rate region that cannot be achieved due to the inaccuracy introduced by our approximation. The size of this region is system dependent. In our experiments, this size was found to be quite small, as shown in the results.

Figure V.4(b) also shows the estimated data rate space under two other candidate configurations C_3 and C_4 (dashed curves). Redundant configurations such as C_3 should be pruned, since their data rate space is completely subsumed by other configurations, while configurations such as C_4 should not, as they can achieve a unique region of the data rate space. Therefore, the methodology partitions the data rate space only among configurations whose individual data rate spaces form a part of the *convex hull* of the total achieved data rate space (in this case, C_1, C_2 and C_4). This is performed as

follows. We start with the candidate configuration C_i, whose data rate curve intersects the Y-axis at the highest point. The intersection points of this curve with the curves under all other candidate configurations, $I_i = \{(X_{C_i,C_1}, Y_{C_i,C_1}), (X_{C_i,C_2}, Y_{C_i,C_2}), ...\}$, are then computed. The first chosen intersection point is one with the highest Y value, say $(X_{C_i,C_j}, Y_{C_i,C_j})$, and this forms the first partition of the data rate space with the associated platform configuration being C_i. Next, the intersection points of the curve for C_j with those for the other configurations is computed. The intersection point selected among them is the one with the highest Y value less than Y_{C_i,C_j}, and this forms the second partition with C_j as the associated configuration. This process is continued until no more intersection points can be selected. These partitions and the associated configurations form the Platform Configuration Table, which is used for run-time platform optimization as described in the Section V.D.3.

Extension to an Arbitrary Number of Applications

To apply the off-line characterization phase to more than 2 applications, for each candidate configuration, the system is simulated for all possible combinations of the applications executing together, to determine different points in the data rate space under this configuration. Therefore, for N applications and C candidate configurations, this requires $C * (2^{|N|} - 1)$ simulations of the system. In practice, we expect N to be fairly small. The data rate space under each configuration is then approximated by fitting these data rate points on an N-dimensional surface. Next, the application data rate space is partitioned among the configurations whose individual data rate spaces lie on the convex hull, by determining the intersection of their data rate spaces, as in the two application case.

V.D.3 Run-time Platform Configuration Phase

The run-time platform configuration phase is responsible for selecting and applying optimized data placement and bus configurations depending on the current data rate requirements of the executing applications. The steps in this phase are illustrated

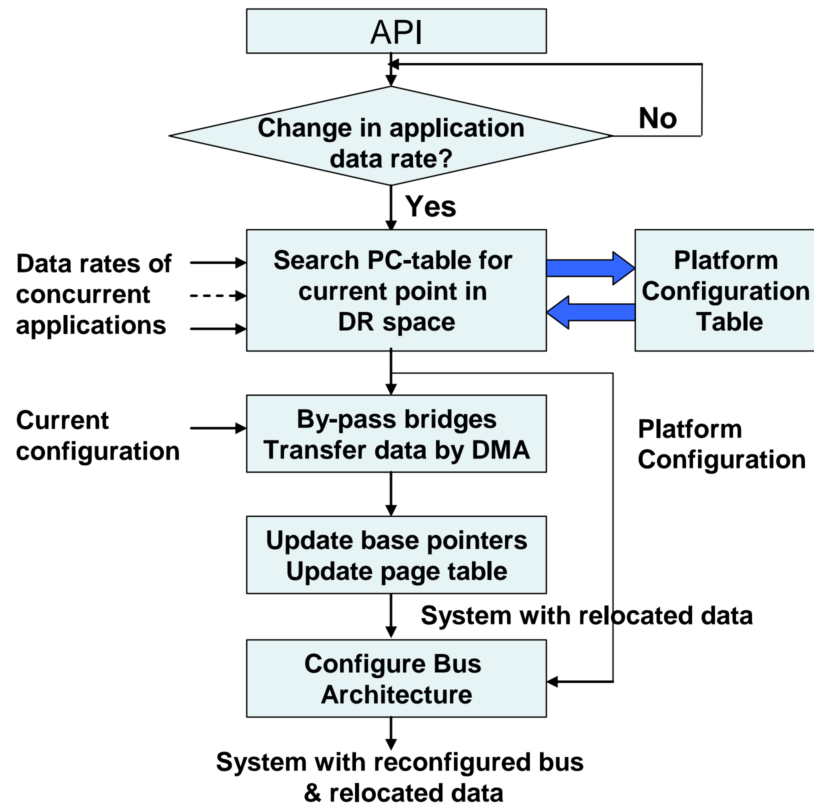


Figure V.5: Run-time Platform Configuration Phase

in Figure V.5. This functionality is implemented as a software middle-ware layer on the embedded processor(s) in the platform (Section II.C). When an application's data rate requirement changes, it indicates this to the platform configuration software through an API (application programming interface).The run-time flow searches the Platform Configuration Table using the new and existing data rate requirements to identify the current point in the data rate space and the pre-computed optimized data placement and bus configuration. The selected data placement is then applied by first by-passing all the bridges using the Bus Reconfiguration Unit (to reduce data transfer time), and then relocating the selected data objects using DMA mode of transfer. Next, the base pointer registers in the hardware components and processor page-tables are updated. Finally, the selected bus configuration is applied by instructing the Bus Reconfiguration Unit to enable or disable the by-pass of the selected bridges according to the identified platform configuration.

V.E Experimental Results

In this section, we present experimental results that evaluate the performance benefits of applying the proposed dynamic platform management methodology to the integrated Viterbi-Turbo decoder design.

V.E.1 Experimental Methodology

The integrated Viterbi-Turbo decoder design was implemented using an instruction set model for the ARM processor, and programmable VERA master bus-functional models [129] for traffic generation for the Viterbi Unit and Turbo Unit hardware. The memories were implemented using programmable VERA slave models [129] with zero wait-state. The page size of the processor is 1 KByte, and hence, OUT_V and OUT_T are mapped to the same page, together constituting one relocatable data structure. The configurable bus architecture with bridge by-pass was implemented by enhancing reference RT-level implementation of the AMBA AHB bus, generated using the CoreConsultant tool of the Synopsys DesignWare AMBA toolsuite [129], using the techniques described in Chapter III. The design was operated at a clock frequency of 400 MHz. Performance analysis results were obtained through cycle-accurate simulations using ModelSim [123]. The data placement and bus configuration of the platform was determined at run-time using the proposed dynamic platform management methodology. The overhead of reconfiguration was taken into account in evaluating the system performance.

V.E.2 Application to the Viterbi-Turbo Decoder Design

The proposed methodology was applied to the integrated 802.11a Viterbi and UMTS Turbo decoder design. The relocatable data objects of the applications and their sizes are shown in Table V.1. The average number of accesses to each data object from each platform component was also estimated. The total number of possible platform configurations are **7422**, out of which the off-line pruning step only selects **6** candi-

Table V.3: Candidate data placement and bus configurations

Config.	Data Placement		Bus Config.
	MEM1	MEM2	
C_1	IN_V, X_T, X'_T, $Z_{1,T}$, $Z_{2,T}$, $L_{e1,T}$, $L'_{e1,T}$, $L_{e2,T}$, $L'_{e2,T}$	$Y_{0,V}$, $Y_{1,V}$, $STable_V$, OUT_V, OUT_T	Multiple Bus
C_2	IN_V, OUT_V, X'_T, $Z_{1,T}$, $Z_{2,T}$, $L_{e1,T}$, $L'_{e1,T}$, $L_{e2,T}$, $L'_{e2,T}$, OUT_T	$Y_{0,V}$, $Y_{1,V}$, $STable_V$, X_T	Multiple Bus
C_3	IN_V, OUT_V, $Y_{0,V}$, $Z_{1,T}$, $Z_{2,T}$, $L_{e1,T}$, $L'_{e1,T}$, $L_{e2,T}$, $L'_{e2,T}$, OUT_T	$Y_{1,V}$, $STable_V$, X_T, X'_T	Multiple Bus
C_4	IN_V, OUT_V, $Y_{0,V}$, $Y_{1,V}$, $Z_{2,T}$, $L_{e1,T}$, $L'_{e1,T}$, $L_{e2,T}$, $L'_{e2,T}$, OUT_T	$STable_V$, X_T, X'_T, $Z_{1,T}$	Multiple Bus
C_5	IN_V, OUT_V, $Y_{0,V}$, $Y_{1,V}$, $STable_V$, OUT_T	X_T, X'_T, $Z_{1,T}$, $Z_{2,T}$, $L_{e1,T}$, $L'_{e1,T}$, $L_{e2,T}$, $L'_{e2,T}$	Multiple Bus
C_6	any	any	Single Shared Bus

date configurations, shown in Table V.3. The data rate space of the applications was then partitioned by approximating the individual data rate space under each candidate configuration, as described in Section V.D.2. We also accurately characterized the data rate space under each candidate configuration by performing detailed simulation of all the configurations. The size of the data rate space that cannot be achieved due to the inaccuracy introduced by our approximation was found to be less than **2%** of the total data rate space that can be achieved. Table V.4 shows the resulting Platform Configuration Table for the design. The first column shows the different partitions of the data rate space for the applications, where DR_V represents Viterbi data rate requirement and DR_T represents Turbo data rate requirement, and the second column shows the corresponding optimized platform configuration. The candidate configurations C_2 and C_5 were redundant, and hence, do not appear in the table.

V.E.3 Platform Configuration Overhead

The overhead of run-time platform configuration includes the time required to relocate data objects between memories (copy and update page tables and base pointers), as well as the time required to enable or disable bridge by-pass in the bus. The worst case

Table V.4: Platform Configuration Table

Data Rate Requirement	Config.
$DR_V \geq 40.8\ DR_T$	C_1
$DR_V < 40.8\ DR_T$ and $DR_V \geq 21.46\ DR_T$	C_3
$DR_V < 21.46\ DR_T$ and $DR_V \geq 8.03\ DR_T$	C_4
$DR_V < 8.03\ DR_T$	C_6

overhead for data relocation is while switching between platform configurations C_1 and C_4, and was measured to be approximately 10 μs (using a DMA unit, with single cycle memory access and with BRG by-passed). The average bus reconfiguration overhead was measured to be approximately 10 cycles. This shows that the platform adaptation overhead is negligible compared to the granularity at which the data rate requirements are expected to change (tens of milliseconds [170]).

V.E.4 Impact of Dynamic Platform Management on Performance

We next evaluate the performance improvements achieved through joint data relocation and bus reconfiguration for the design, and compare it to the performance under three other cases: (i) best static data placement and bus configuration, (ii) only dynamic data relocation (bus configuration fixed), and (iii) only dynamic bus reconfiguration (data placement fixed).

Performance under Best Statically Configured Design: Figure V.6(a) shows the data rate space achieved for the Viterbi and Turbo decoding applications when the platform configuration is statically fixed to configuration C_4 from Table V.3 (light shaded area). This is the best static configuration, since it can satisfy the largest space of data rates among all candidate configurations.

Performance under Dynamic Data Relocation: In this experiment, the performance of the design under only dynamic data relocation was evaluated, while keeping the bus configuration fixed as a multiple bus. This corresponds to selecting the platform configuration at run-time from among C_1 to C_5 from Table V.3. Figure V.6(b) shows the

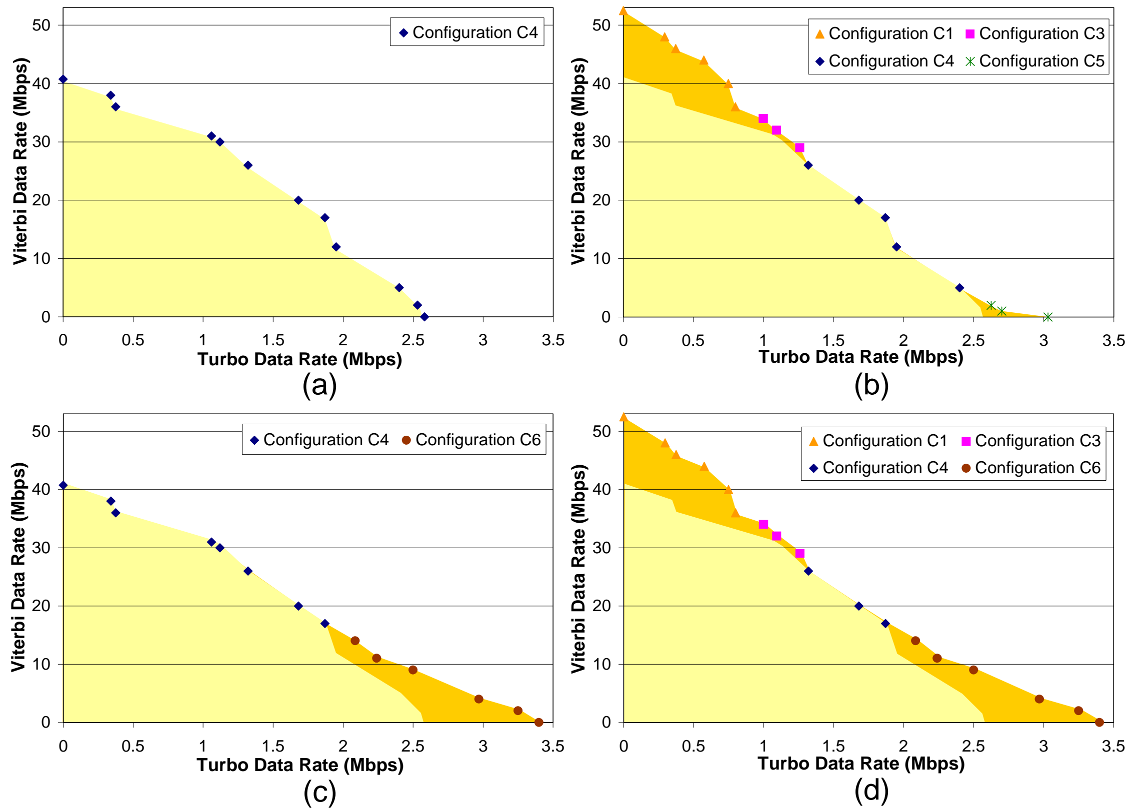


Figure V.6: Data rate space achieved under different platform adaptation schemes

data rate space achieved under this technique (complete shaded area), and the platform configurations selected for each data rate combination simulated. It also shows the data rate space under the best static configuration (light shaded area). Comparison of the two indicates that dynamic data relocation can help satisfy a larger space of performance objectives for the design.

Performance under Dynamic Bus Reconfiguration: Figure V.6(c) shows the data rate space achieved under only dynamic bus reconfiguration, while keeping the data placement fixed as in the best static configuration (complete shaded area). This corresponds to selecting the platform configuration between C_4 and C_6 from Table V.3. It also shows the configuration selected for each data rate combination simulated. Comparison of this data rate space with that under the best static configuration (light shaded

area) indicates that dynamic bus reconfiguration can also provide significant performance improvements.

Performance under Joint Data Relocation and Bus Reconfiguration: Finally, the performance of the design under the proposed dynamic platform management methodology was evaluated. Figure V.6(d) shows the data rate space achieved under this scheme (complete shaded area), and the configuration selected for each data rate combination simulated. It also shows the performance under the best static configuration (light shaded area) for comparison. From the figure, we observe that joint data relocation and bus reconfiguration can satisfy a much larger space of data rate requirements compared to a statically configured design (up to 32% data rate improvements when only Turbo decoding executes). Also, comparison of Figure V.6(d) with Figures V.6(b) and (c) indicates that by exploiting both data relocation and bus reconfiguration together, the space of data rates achieved is much larger than when they are individually configured.

V.F Conclusions

In this chapter, we presented dynamically configurable SoC platforms with two different configurability features, namely, data relocation, and bus reconfiguration. We illustrated the interdependence between these features, and presented a platform management methodology for the run-time optimization of such platforms. Experiments on an integrated 802.11a Viterbi and UMTS Turbo decoder design indicate that the proposed technique results in significant performance improvements compared to conventional statically optimized architectures.

The text of this chapter, in part, is based on material that has been accepted for publication in the Design Automation and Test in Europe Conference, 2006. The dissertation author was the primary researcher and author, and the coauthors listed in these publications collaborated on, or supervised the research that forms the basis for this chapter.

VI

Application-Architecture Co-Adaptation

VI.A Introduction

In this previous chapters of this thesis, we presented general-purpose, dynamically configurable platforms, an important design alternative to customized hardware solutions (*e.g.*, ASICs, custom SOCs). We described several dynamic configurability options for SoC platforms, and presented dynamic platform management, a methodology for the run-time, application-specific customization of such platforms. We illustrated that such an approach, by adapting the platform to time-varying application requirements, can provide significant benefits in terms of overall system performance, application concurrency, and energy-efficiency.

Complementary to platform customization is the requirement that the applications themselves also be customized to the characteristics of the platform on which they execute. Recognizing this, implementations of popular applications tailored for specific platforms (*e.g.*, wireless platforms) have started to emerge [171, 172]. However, as devices become increasingly multi-functional, the availability of computing resources within the underlying platform (*e.g.*, processor cycles, memory, bus bandwidth) may also exhibit significant dynamic variation (*e.g.*, due to variable application concur-

rency). As a result, applications that are only tailored to the peak, or average processing capabilities of a platform may often result in sub-optimal execution. Many existing and emerging applications (such as image compression, security protocols, audio/video streaming) provide the flexibility of selecting algorithm parameters, or even selecting which algorithms to use, at run-time [173, 174, 175, 176]. For example, MPEG4 allows for varying the intra-frame refresh rate, while the SSL protocol provides clients with the flexibility of selecting appropriate ciphering and authentication algorithms. The flexibility offered by such applications can be exploited in order to enable dynamic tradeoffs between application quality and the load imposed on the underlying platform.

For highly customized application-architecture solutions, it is important to customize both the applications and the architecture towards each other. Therefore, in this chapter, we focus on *application-architecture co-adaptation* techniques for the integrated configuration of both the executing applications (to regulate application resource usage), and the underlying platform architecture (to tailor it to the requirements imposed by the applications).

VI.A.1 Application-Architecture Co-Adaptation: Overview

Figure VI.1 illustrates the overall concept of application-architecture co-adaptation. We consider dynamically configurable SoC platforms, on which would execute multiple applications that are flexible, enabling a tradeoff between application quality and platform resource usage via application parameters. As described earlier, the dynamic platform management layer is responsible for understanding the time-varying requirements imposed by the applications, and appropriately customizing the configuration of the platform at run-time. However, in cases, the platform may be unable to support these requirements, even after intelligent platform management, due to inherent constraints on the available hardware resources (CPU capacity, battery-life, *etc*). In such cases, along with platform management, the executing applications themselves can also be customized using a *dynamic application management* layer, so that their load imposed on the platform is sustainable (Figure VI.1). Such application management,

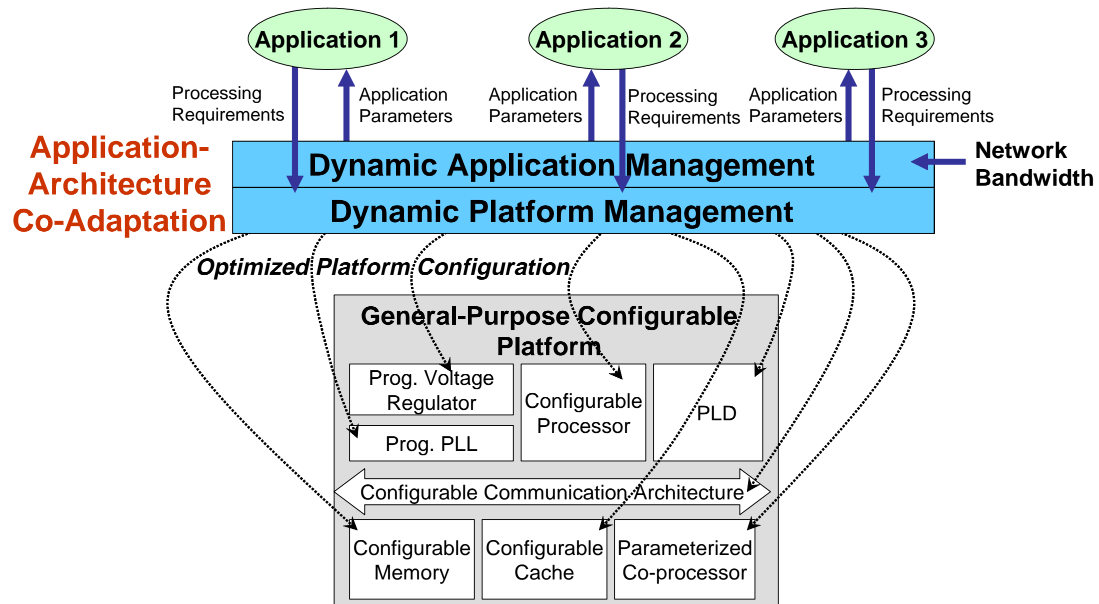


Figure VI.1: Application-architecture co-adaptation for the integrated configuration of both applications and the platform architecture

however, should minimize the resulting loss in application quality. Dynamic application management and platform management are tightly coupled together, and operate synergistically for optimized application-architecture co-adaptation.

One of the areas in which such an approach would be important is wireless application processing. Rapid growth in functional integration, as well as complexity of individual functions, coupled with stringent limitations on cost, size, and battery capacity, make the design of such systems a challenging task. These objectives can be successfully achieved only through a careful process of customizing both the processing platform, and the applications that are targeted towards it [177]. For wireless systems, it is crucial to take into account not only the platform resources, but also the available network bandwidth, which can change over time depending on dynamic channel conditions, number of users, *etc* (Figure VI.1). This work focuses on co-adaptation techniques for such wireless application processing systems.

VI.A.2 Chapter Overview

In this chapter, we present application-architecture co-adaptation techniques for the dynamic and synergistic customization of flexible wireless applications and the underlying platform on which they execute. The proposed approach, which is described in the context of a wireless image delivery system, aims at improving application metrics (*e.g.*, latency, quality, energy consumption), while faced with potential variations in (i) resource availability in the platform architecture, and (ii) available network bandwidth.

While application adaptation in response to network variability has been extensively studied, the benefits of adapting applications to variations in platform resources has not been throughly investigated. We quantitatively illustrate the shortcomings of conventional approaches, in which applications are statically customized, or that are straightforward extensions of network-centric adaptation techniques. We also illustrate the advantages of joint adaptation of applications and the platform. We present a methodology for such co-adaptation, and describe it in detail for a wavelet-based wireless image application executing on a frequency and voltage scalable platform. The methodology involves characterizing the parameter space of the application in terms of its impact on platform resource usage, network bandwidth, and application quality metrics. The results of these characterizations are then used to guide the co-adaptation algorithms, that respond to variations in platform resource availability and network bandwidth, by suitable adjustment of application and architecture parameters.

Detailed experiments were conducted for the wireless image delivery system by implementing it on a Linux-based iPAQ-3765 device connected to the internet via a high-data-rate cellular access technology (CDMA 1x) [178]. The experiments demonstrate that the proposed co-adaptation policies are successful in satisfying application requirements, while minimally impacting user experience, and also provide energy savings of up to 70%.

The rest of this chapter is organized as follows. In the next section, we illustrate the advantages of application-architecture co-adaptation, using a wireless image delivery system as an example. In Section VI.C, we describe a general methodology for

co-adaptation, highlighting the various steps. In Section VI.D, we describe in detail, the application of the methodology to the image delivery system, providing in turn, a detailed analysis of the impact of application parameters, and algorithms for co-adaptation. In Section VI.E, we present experimental results that evaluate the advantages of our approach. Finally, in Section VI.F, we summarize the work presented in this chapter.

VI.A.3 Related Work

Technologies for providing dynamic configurability in SoC platforms, and techniques for exploiting them in response to time-varying application requirements have been studied. These were surveyed extensively in Chapter II, and include the work presented in the previous chapters of this thesis. However, they focus on hardware configuration, and do not usually co-ordinate with application-level parameter control policies.

Many applications are flexible, providing opportunities for dynamic customization through selection of algorithms and algorithm parameters [173, 174, 175, 176]. Detailed analysis of the impact of application parameters can be found in the corresponding literature [179, 180]. The flexibility offered by such applications has been successfully exploited to dynamically trade-off application quality for consumed network bandwidth [181, 182, 183, 184]. These techniques, however, lack system-level co-ordination, either ignoring dynamic variability of processing resources, or ignoring customization opportunities provided by the platform hardware.

These observations have led to recent research on integrated adaptation frameworks that aim at holistically addressing both application-level as well as architecture-level parameters [185, 186, 187]. Advances include the development of software infrastructure that provides interfaces for co-ordinated adaptation between applications, system software (OS, middle-ware), and the underlying hardware [185, 186]. The availability of such infrastructure and potentially standardized interfaces would significantly ease the deployment of techniques such as those described in this chapter. Integrated adaptation approaches have also been proposed as a way to greatly increase the dynamic

power range of systems, in order to better suit the widely variable efficiency characteristics of renewable power sources such as solar cells [187]. While co-ordinated adaptation between applications and architectures is an area that has started to receive interest, there remains a lack of case studies that document in detail, and quantitatively illustrate the steps in designing co-adaptation policies for specific application-architecture combinations.

VI.B Motivation

With the proliferation of mobile handheld devices, there is growing demand for wireless image data services, such as camera phones, remote medical monitoring, home security monitoring, *etc*. An important requirement of such services is to be able to efficiently retrieve and display images on the mobile handhelds in real time. In this section, we first describe such a wireless image delivery system based on a Linux-based hardware platform. Using this case study, we quantitatively illustrate the shortcomings of conventional approaches, and motivate the need for adapting applications to dynamic variations in the availability of processing resources. We also illustrate the impact of co-ordinated adaptation of both application and architectural parameters.

VI.B.1 Case Study: Wireless Image Delivery System

Figure VI.2 illustrates the overall framework for the image delivery system. The image application executes on a wireless platform, and downloads and displays images in real-time from an image web server. There might be other applications also executing on the platform, leading to a variability in the processing resources (CPU) available for the image application. The bandwidth available on the wireless link may also vary over time due to variations in channel conditions, number of users, *etc*. We next describe the image delivery application, followed by a description of the hardware platform.

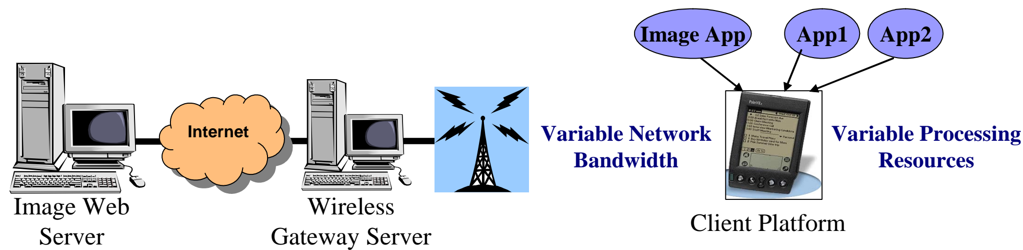


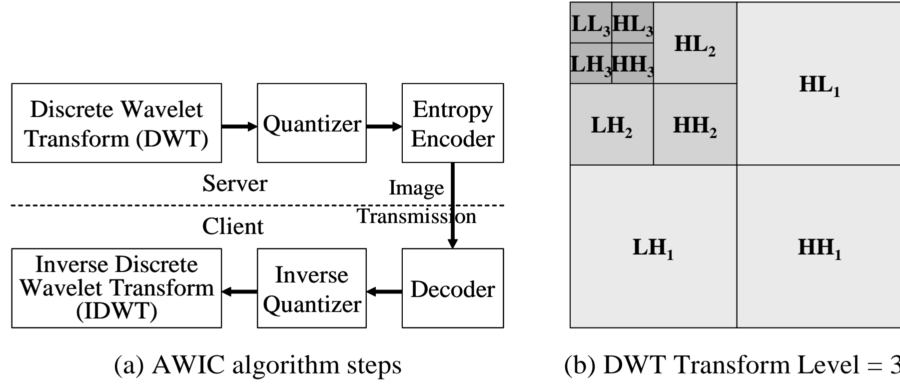
Figure VI.2: Wireless image delivery system

Wavelet-Based Image Compression

The image delivery application is based on the Adaptive Wavelet Image Compression (AWIC) [173] algorithm, which consists of three steps as shown in Figure VI.3(a). First, the discrete wavelet transform (DWT) applies a set of filters to decorrelate the raw image data into different frequency sub-bands (Figure VI.3(b)). The label XY denotes whether the sub-band contains low-pass (L) or high-pass (H) values after the row transform (X) and the column transform (Y). The DWT step can be iteratively applied to the LL sub-band for a variable number of *Transform Levels* (TL). Figure VI.3(b) shows DWT applied to an image with $TL = 3$. Higher TL further decorrelates the image, at the cost of increased computation effort. Next, each sub-band is quantized according to a variable *Quantization Level* (QL). Higher QL facilitates compression, but results in increased loss of data (adversely affecting the image quality). Next, the quantized values are encoded (using Huffman coding) to compress the image. Decompression is symmetric to the compression process, involving decoding, inverse quantization, and inverse DWT (Figure VI.3(a)). By default, the image transform level is fixed at 4, so that the image size is small (TL values higher than 4 lead to negligible size reductions), and the quantization level is fixed at 0, for highest image quality.

Hardware Platform

The wireless client platform is an iPAQ-3765 PDA with an Intel StrongARM SA-1110 processor with 64 MB RAM running the Familiar Linux distribution [188]. The platform connects to the network over a high data-rate, CDMA-1x based Sierra



(a) AWIC algorithm steps (b) DWT Transform Level = 3

Figure VI.3: Adaptive Wavelet Image Compression (AWIC)

Wireless Aircard [178]. Several platforms of this class can be operated at different discrete frequency and voltage levels [154, 189]. The StrongARM has support for 11 discrete frequency levels from 59 Mhz to 206 Mhz, with corresponding supply voltages from 0.83 V to 1.5 V [154]. The time taken by the PLL and DC-DC converter outputs to stabilize each time the voltage and frequency values are changed is 150 μsec [150].

VI.B.2 Co-Adaptation: Illustrative Examples

For this study, we performed experiments downloading and displaying images from the image web server, while concurrently executing an MPEG video application [190] on the client platform. The processor usage of the MPEG application (averaged over a 1 second window) varies over time, depending on its frame-rate and frame characteristics (Figure VI.4(a)). The image application is associated with a soft latency constraint, as might be imposed by data services such as slide shows or remote medical monitoring, such that the total time taken to retrieve and display an image be less than 1 second. Over the time-line shown in Figure VI.4(a), the image viewing application is required to display 150 different images, one every second. Image quality is measured using peak signal-to-noise ratio (PSNR). [1]

We consider three versions of the image viewing application.

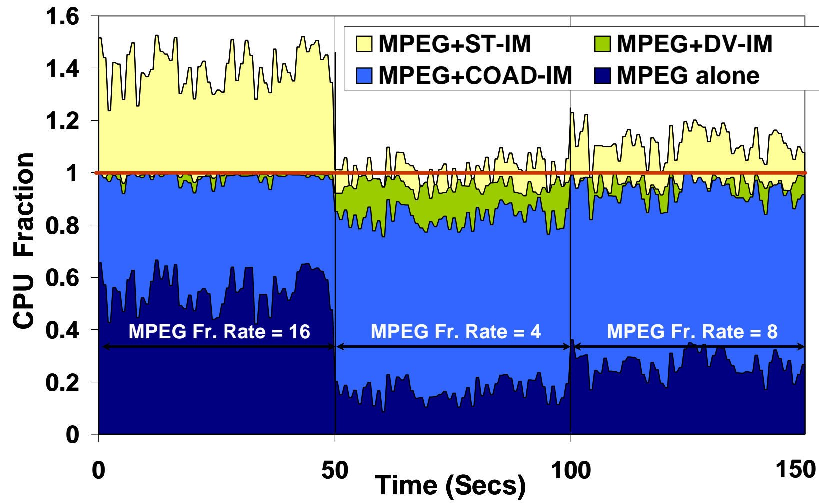
[1] To highlight the importance of variability of processing resources, in this study, we assume that sufficient network bandwidth is available. However, as borne out by experimental results, the co-adaptation techniques adequately account for network variabilities as well.

(i) ST-IM: In this, the image application parameters are statically fixed at their default values ($TL = 4$ and $QL = 0$), irrespective of the client platform resources.

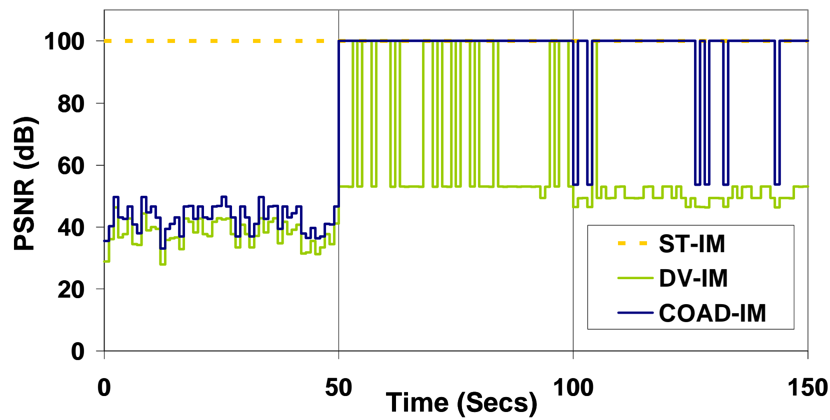
(ii) DV-IM: Techniques for customizing applications to available network bandwidth typically rely on trading the application quality for the volume of data communicated (*e.g.*, using parameters such as the application quantization level). The DV-IM (data volume) version uses a similar concept, in which the image QL parameter is adapted based on the availability of processing resources. This is because increasing QL during periods of lower processor availability helps reduce the amount of data to be decompressed, resulting in lower processing requirements. TL is fixed at its default value of 4, since it results in smaller image sizes.

(iii) COAD-IM: In this version, the full space of application and architecture parameters are considered, and configured in a co-ordinated manner, considering dynamic variability in the platform resources.

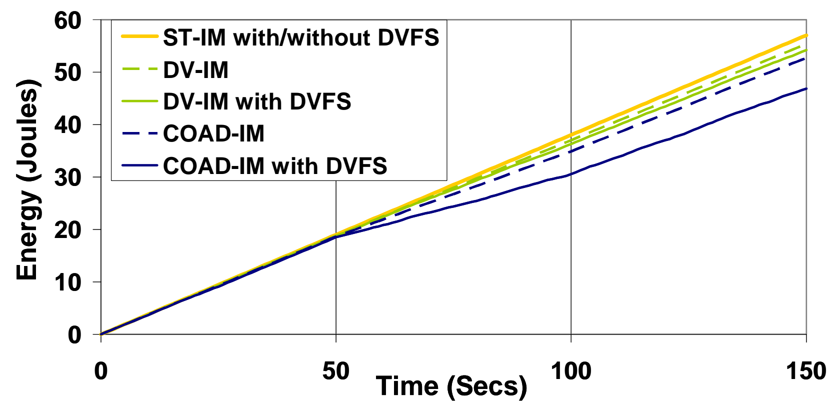
Example 1: We measured the total CPU requirements imposed on the iPAQ-3765 to meet the latency constraints of the MPEG and the image applications (under each of the above three versions) when both execute concurrently (Figure VI.4(a)). The solid horizontal line (CPU Fraction = 1) indicates 100% CPU utilization. Figure VI.4(b) illustrates the corresponding image quality (PSNR) for each application version. PSNR = ∞ (highest image quality) is plotted at 100 dB for reference. We observe that for ST-IM, the image quality is always the highest (PSNR = ∞).However, Figure VI.4(a) illustrates that the CPU requirement often significantly exceeds 100%, indicating that the image application *often fails to meet performance requirements* in the presence of the MPEG application. Both DV-IM and COAD-IM can meet the image latency constraint (total CPU fraction required is less than 1) under varying loads imposed by the MPEG application. However, Figure VI.4(b) illustrates that the PSNR achieved by DV-IM is often poor, compared to COAD-IM. This is because COAD-IM is able to *customize the application to varying platform constraints through a more optimized selection of application parameters* compared to DV-IM. This indicates that using network-centric



(a) Processor utilization under different application versions



(b) PSNR under different application versions



(c) Energy consumption under different application versions

Figure VI.4: Concurrent execution of MPEG and different versions of the image application: timeline of (a) processor utilization, (b) PSNR and (c) energy consumption

adaptation schemes, which merely regulate the volume of data processed, may unnecessarily compromise image quality when the processing platform is the bottleneck.

Example 2: From Figure VI.4(a), we observe that, in general, optimizing application parameter selection reduces the CPU utilization. In particular, the reduction under COAD-IM is significantly more than DV-IM. To examine the potential for exploiting this through appropriate configuration of the platform, we compare the scenarios wherein (i) the platform is always operated at a fixed frequency and voltage (206 MHz, 1.5 V), and (ii) a customized dynamic voltage and frequency scaling (DVFS) policy that works in close co-ordination with the application parameter setting policy is used (described in Section VI.D). Figure VI.4(c) shows the cumulative energy consumed by the platform over time for each version of the image application, both with, and without the integrated DVFS strategy. For ST-IM, both with and without the integrated DVFS, the platform always operates at the highest frequency and voltage, due to the high processing load (Figure VI.4(a)), and hence consumes the maximum energy. Under COAD-IM, the total load imposed on the platform is much lower than the other cases, especially during periods of low MPEG utilization (*e.g.*, frame-rate=4), due to better dynamic adjustment of image application parameters (Figure VI.4(a)). This leads to lower energy consumption, even without integrated DVFS (due to longer CPU idle times). However, with the integration of a DVFS strategy, we observe that the savings in energy consumption are as much as 37% during reduced MPEG utilization. For the entire timeline, we observe an overall energy savings of 18%.

Summary: From these studies, we note the following points:

- The execution of the MPEG application resulted in significant dynamic variability in the available processing resources. Under this scenario, keeping the image application parameters constant leads to failure in meeting performance constraints.
- Simple techniques that merely trade off image quality for the volume of data processed often do not suffice, since they are significantly sub-optimal from the standpoint of regulating the requirements imposed on the hardware platform. This

suggests that dynamic application management needs to be based on a systematic analysis of the entire space of application parameters, while considering the characteristics of the underlying hardware.

- The example demonstrated how optimizing the image delivery parameters *enables* more aggressive frequency and voltage scaling, which if properly exploited, leads to large savings in energy consumption. This motivates *co-adaptation* techniques that integrate *platform*-aware, *application* customization, with *application*-aware, *platform* customization.

Finally, it bears mentioning that we drew similar conclusions from studies that evaluated the importance of considering variability in the network bandwidth for the case study system. Since such benefits have been well-documented in the past, we omit those studies here.

VI.C Co-Adaptation Methodology

In this section, we first formulate the problem of application-architecture co-adaptation, and then present an overview of our methodology.

VI.C.1 Problem Definition

We consider parameterized applications executing on a wireless client platform, transferring data to/from an application server. The application is associated with a set of performance requirements, and a set of parameters that influence usage of network bandwidth, usage of platform resources, and application quality metrics. The bandwidth on the wireless link may vary over time. Additionally, the available processing resources of the platform may also vary with time, depending on the load imposed by other concurrent applications. The aim of co-adaptation is to select appropriate application parameters and the platform configuration at run-time, taking into account the available network and platform processing resources, such that the performance requirements are satisfied, while optimizing application quality metrics.

VI.C.2 Methodology Overview

The overall methodology for co-adaptation is illustrated by Figure VI.5. The inputs to the methodology include (i) the application (mapped to the client platform), (ii) the set of application parameters, which include algorithm parameters (*e.g.*, quantization levels), or higher-level parameters that select between alternative algorithms (*e.g.*, rendering techniques), (iii) typical input stimuli, and (iv) a simulation model or implementation of the platform. In the off-line phase, a detailed analysis of the impact of the parameters is performed. This includes profiling the application to capture the dependence of the amount of data transferred by the application, and application quality metrics (*e.g.*, PSNR) as a function of the parameters. Architectural simulation of (or actual execution on) the target platform is used to quantify the impact of the parameters on the usage of different platform resources (*e.g.*, CPU). The result of this phase is a set of models, or look-up-tables (LUTs) that enable fast evaluation of the bandwidth usage, platform resource usage, and application quality metrics, under given parameter values.

The run-time phase (Figure VI.5) includes the design of the co-adaptation policy itself. The inputs to the policy include (i) the available network bandwidth, which we assume is obtained via on-line bandwidth monitoring utilities (*e.g.*, Iperf [191]), (ii) the available platform processing resources, which are obtained through system services that monitor resources such as CPU, memory, *etc.*, and (iii) performance requirements of the application, which are either specified by the user, or may be hard-coded in the application. Note that, the interval over which resource availability is measured should be chosen carefully. Small intervals could result in oscillations in application behavior, whereas large intervals may result in poor response times to dynamically changing conditions. The co-adaptation policy uses these inputs and the models developed in the previous step for the optimized selection of both application parameters and the platform configuration in an integrated manner. For selecting the application parameters, a naive policy might exhaustively search its parameter space. However, for most applications, the impact of the parameters on application quality, network bandwidth, and platform resource usage is predictable, and hence, can be used to efficiently select optimized

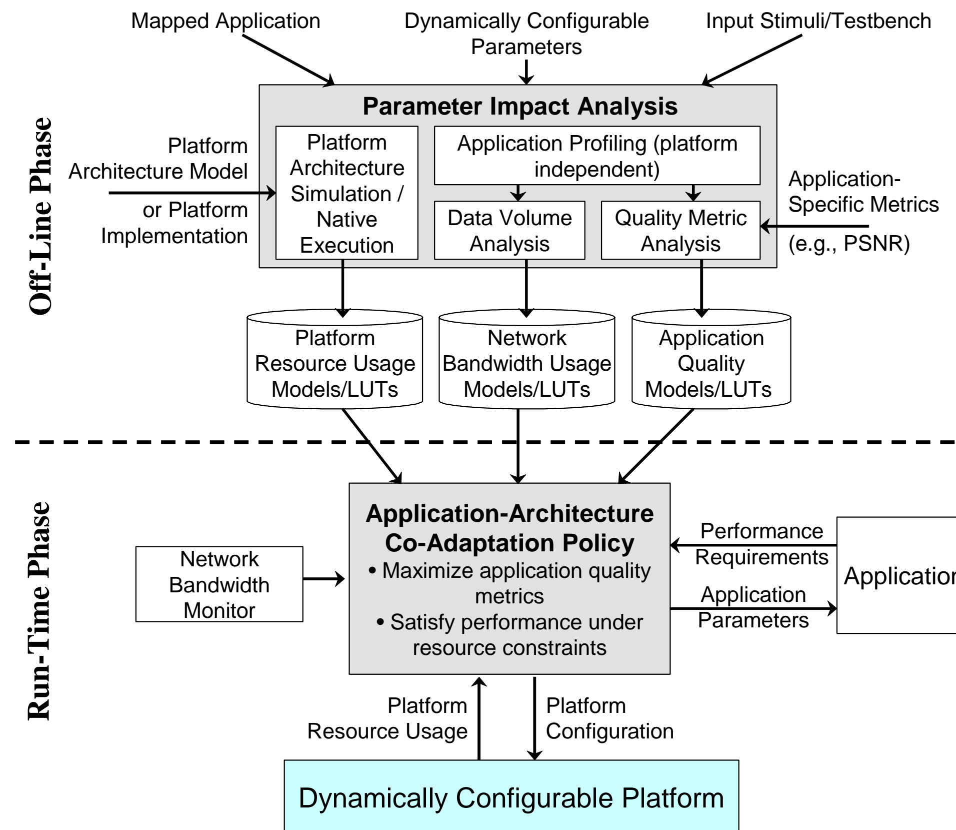


Figure VI.5: Application-architecture co-adaptation methodology

parameters (as illustrated in the next section). The granularity at which application parameters are selected is, in general, application dependent. Certain parameters may be changed at relatively small timescales with little overhead (*e.g.*, quantization levels of an image), while other parameters affect larger timescales (*e.g.*, the choice of security algorithm used for a secure file transfer).

The co-adaptation policy is implemented on either (i) only the client platform, when the application parameters do not affect the communicated data (*e.g.*, parameters that select the rendering algorithm used by the application), or (ii) both the client platform and the web server, when application parameter selection requires cooperation between the two (*e.g.*, ciphering algorithm used for a secure transaction).

In the next section, we describe how this co-adaptation methodology is applied to the wireless image delivery system (Section VI.B.1).

VI.D Application of Co-Adaptation to Wireless Image Delivery

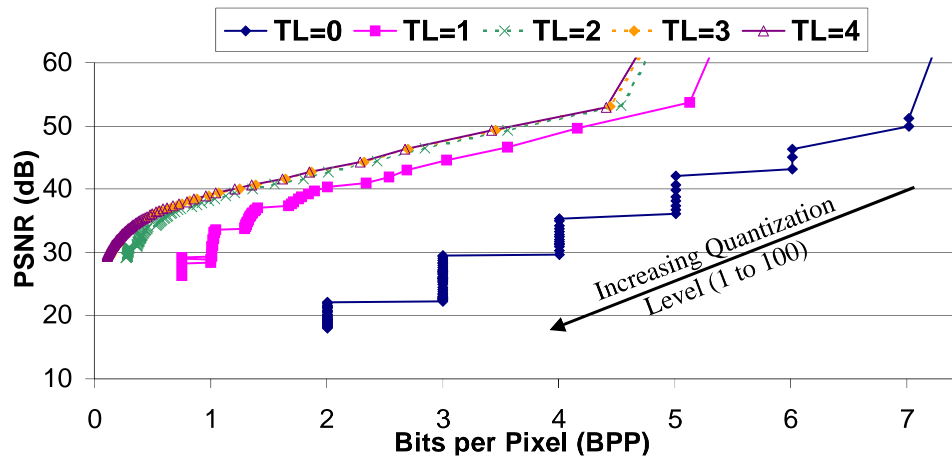
In this section, we describe the different steps of the co-adaptation methodology for the wireless image delivery system, presenting in turn, a systematic analysis of the impact of different parameters, and the design of the run-time co-adaptation policy.

VI.D.1 Parameter Impact Analysis

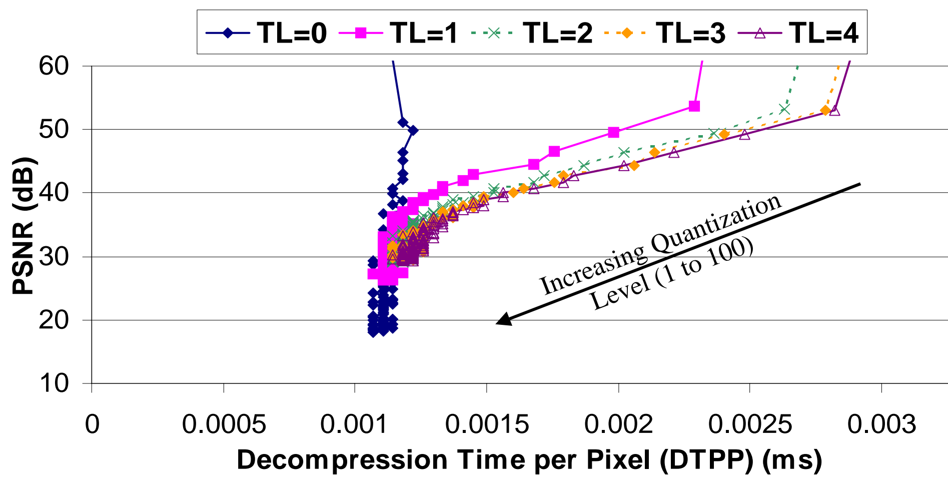
Off-line parameter impact analysis (Figure VI.5) of the image application was performed for the hardware platform to quantify the impact of the image transform level (TL) and quantization level (QL) on compressed image size, image decompression time on the client platform, and image quality ($PSNR$). Figure VI.6(a) shows the impact of TL and QL on the $PSNR$ and compressed image size (normalized to the number of bits per pixel (BPP)), averaged over a set of benchmark images [192]. From the figure, we observe that for a constant TL, as QL increases, the image quality and size (BPP) both decrease, due to greater truncation of pixel values. Also, we note that the $PSNR$ versus compression (BPP) trade-off is superior for *larger* values of TL. This is because with increasing TL, the image is further decorrelated, allowing better compression (*i.e.*, smaller BPP). Increasing TL beyond 4 provides insignificant benefits, and increasing QL beyond 100 leads to very poor image quality.

Figure VI.6(b) shows the impact of TL and QL on the $PSNR$ and decompression time at the client platform (normalized to the Decompression Time per Pixel ($DTPP$)) for the same benchmarks. The figure shows that for a constant TL, as QL increases, the image quality as well as the decoding complexity decreases, since increasing QL leads to smaller images. However, unlike the trade-off in Figure VI.6(a), the $PSNR$ versus $DTPP$ tradeoff is superior at *smaller* TL values, since this results in fewer iterations of IDWT, which reduces the decompression time.

In summary, higher QL values lead to lower image quality, size and decompression time. However, for the same image quality, *higher TL values result in smaller image sizes, while lower TL values result in smaller decompression times*. Hence, for



(a) PSNR vs image size (bits per pixel)



(b) PSNR vs decompression time per pixel

Figure VI.6: Impact of image transformation level (TL) and quantization level (QL) on image quality, size and decompression time

the image application to meet a total latency constraint (sum of image transmission and decompression time), while maximizing the image quality, the parameters have to be chosen carefully, depending on whether the network bandwidth and/or the platform is the bottleneck.

The result of this off-line step is a table called the QL-TL Table, shown in Figure VI.7. The table contains $PSNR$, BPP and $DTPP$ values averaged over many images for all TL values (0 to 4), and all QL values (0 to 100). This table corresponds to the three LUTs illustrated in Figure VI.5 and is used by the run-time co-adaptation policy, as described next.

QL \ TL	0	1	2	3	4
0	PSNR = ∞ BPP = 8.02 DTPP = 0.0012	PSNR = ∞ BPP = 6.22 DTPP = 0.0025	PSNR = ∞ BPP = 5.91 DTPP = 0.0029	PSNR = ∞ BPP = 5.91 DTPP = 0.0032	PSNR = ∞ BPP = 5.94 DTPP = 0.0031
1	PSNR = ∞ BPP = 8.02 DTPP = 0.0011	PSNR = 53.66 BPP = 5.13 DTPP =0.0023	PSNR = 53.17 BPP = 4.54 DTPP = 0.0027	PSNR = 53.06 BPP = 4.44 DTPP = 0.0027	PSNR = 53.00 BPP = 4.41 DTPP = 0.0027
99	PSNR = 18.13 BPP = 2.01 DTPP = 0.0011	PSNR = 26.31 BPP = 0.75 DTPP =0.0011	PSNR = 29.12 BPP = 0.28 DTPP = 0.0012	PSNR = 29.46 BPP = 0.14 DTPP = 0.0012	PSNR = 29.37 BPP = 0.11 DTPP = 0.0012
100	PSNR = 18.03 BPP = 2.01 DTPP = 0.0011	PSNR = 26.28 BPP = 0.75 DTPP =0.0011	PSNR = 29.11 BPP = 0.28 DTPP = 0.0012	PSNR = 29.41 BPP = 0.14 DTPP = 0.0012	PSNR = 29.13 BPP = 0.11 DTPP = 0.0012

PSNR decreasing
BPP decreasing
DTPP decreasing

Figure VI.7: QL-TL Table used by run-time co-adaptation policy

VI.D.2 Run-Time Co-Adaptation Policy

The next phase of the methodology (Figure VI.5) is the design of the co-adaptation policy. We first define the notations used, and then describe the co-adaptation algorithms.

Notation and Formulation

We consider that the client platform architecture is capable of operating over a range of frequencies $F = \{f_1 < f_2 < ... < f_{max}\}$, with corresponding voltage levels.

The image application has a soft total latency constraint (T_{total}) which should not be exceeded by the sum of the transmission latency (T_{trans}) and decompression latency (T_{dec}). For simplicity, we assume that the time consumed by the server and the wired links is negligible. The number of pixels in an image is denoted by $\mathcal{P}$. For a given TL and QL, $BPP(TL, QL)$, $DTPP(TL, QL)$ and $PSNR(TL, QL)$ denote the bits per pixel, decompression time per pixel, and $PSNR$ of the image, respectively, and are obtained from the QL-TL table (Figure VI.7). Q_{max} is the maximum achievable $PSNR$ from the table (in this case ∞). Let the available network bandwidth be BW. The CPU fraction (on average) collectively consumed by other (higher priority) applications at the highest frequency (f_{max}) is CPU_{other}, and the total CPU fraction used (including the image application) is CPU_{total}. [2] The purpose of the co-adaptation policy is to select TL, QL, and the minimum platform frequency f, such that the $PSNR$ of the image is maximized, while satisfying the latency constraint T_{total}, without exceeding a CPU requirement of 100%. This can be expressed as:

$$\begin{aligned} maximize: &\quad PSNR(TL, QL) \\ subject\,to: &\quad T_{trans} + T_{dec} \leq T_{total} \\ &\quad CPU_{total} \times f_{max}/f \leq 1 \end{aligned} \tag{VI.1}$$

Combining the inequalities and substituting terms for the T_{trans} and T_{dec} we get:

$$\begin{aligned} maximize: &\ PSNR(TL, QL) \\ subject\,to: &\ \frac{\mathcal{P} \times BPP(TL, QL)}{BW} + \frac{\mathcal{P} \times DTPP(TL, QL)}{f/f_{max} - CPU_{other}} \leq T_{total} \\ &\ and\ f = min\{f_i : f_i \in F\} \end{aligned} \tag{VI.2}$$

We next describe the co-adaptation algorithm for the selection of TL, QL and f so as to satisfy the above equations.

[2] Note that, using average values is permissible since the latency constraint is a soft one.

Co-Adaptation Algorithm

From Equation VI.2 we make the following two key observations:

- For any solution TL, QL and f of Equation VI.2, if $PSNR(TL, QL) < Q_{max}$, then $f = f_{max}$. This is because if $f < f_{max}$, then the client platform could have been operated at a higher frequency in order to further improve the image quality.
- For any solution TL, QL and f of Equation VI.2, if $PSNR(TL, QL) = Q_{max}$, then the platform can operate at a frequency $f \leq f_{max}$, since increasing the frequency cannot further increase the image quality.

Based on the above observations, it can be seen that the selection of the image compression parameters, TL and QL, can be made independent of the platform operating frequency, f. The problem can thus be solved using a distributed approach where TL and QL are selected at the server assuming that the client platform executes at the highest frequency, f_{max}. The platform parameters (frequency and voltage) are chosen at the client platform taking into account the selected image TL and QL values, which are embedded in each image.

Selection of Application Parameters at the Server: The server chooses TL and QL for each image request based on network bandwidth (BW), CPU load (CPU_{other}), and the image latency constraint (T_{total}), which are provided to it by the client application. As explained earlier, the server can safely assume that the client platform executes at f_{max}. Hence, the constraint in Equation VI.2 can be simplified by substituting f by f_{max}. The server uses the QL-TL Table (Figure VI.7) as follows. From the table, it can be seen that, for a given TL, as QL increases, the $PSNR$, BPP and $DTPP$ all decrease. Hence, the optimal TL and QL values can be selected by a binary search of the QL-TL Table along the rows of the table (*i.e.*, along the QL values). At each search step, all the entries in the row (*i.e.*, for all the TL values) are checked to see if they satisfy the constraint. If none of them do, then the rows corresponding to higher QL values are searched. If one or more of the table entries satisfy the constraint, then the entry with the highest $PSNR$ value is examined. If this $PSNR$ is higher than

the previously best encountered $PSNR$, then this value is stored along with the corresponding TL and QL, and the search is continued along the rows with lower QL. Otherwise, the rows with higher QL values are searched. At the end of the search, the stored QL and TL values are used as the image parameters. This algorithm is optimal and its complexity is $O(|TL|log_2(|QL|))$.

Selection of platform parameters at the client: At the client, the decompression time under the selected TL and QL values is calculated from the QL-TL Table. We assume that the execution time of the other tasks can be estimated on arrival using accurate off-line characterization techniques (such as described in this thesis) or using predictive estimation techniques [151]. At the arrival of each task instance (including the image viewing application), the schedulability test for EDF [152] is used to select the lowest frequency, f, at which the set of tasks can still meet their performance requirements. This is given by $\sum_{i=1}^{N} \frac{ET_i}{P_i} = \frac{f}{f_{max}}$, where N is the number of tasks, and ET_i and P_i are the estimated execution time and the time-interval of task i, respectively. The voltage level corresponds to the selected frequency setting.

VI.E Experimental Results

In this section, we present experimental results that study the effectiveness of the proposed co-adaptation techniques for the wireless image delivery system.

VI.E.1 Experimental Methodology

The image application was implemented in C and compiled for the hardware platform described in Section VI.B.1. The image decompression times were measured using the *gettimeofday()* system call. $PSNR$ and image size analysis was performed by profiling the application over a large set of images. The energy consumption estimates include both the CPU and the network interface card energy, and were obtained using cycle-accurate, software energy profiling [154] for the platform, and datasheet specifications for the network interface card [178].

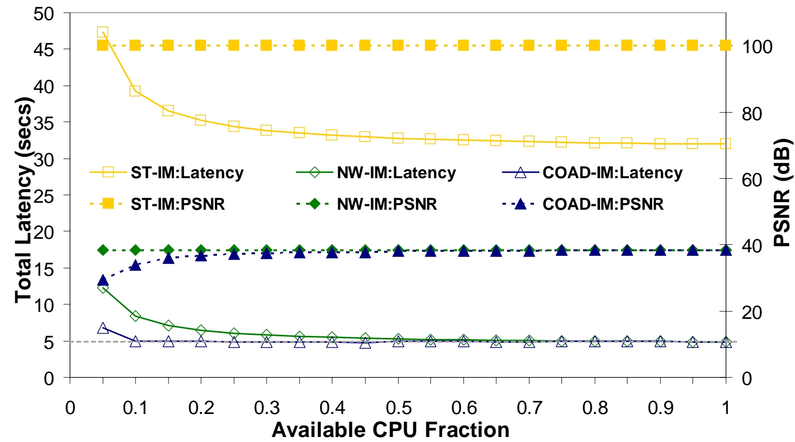
In the following experiments, we compare the co-adaptation based image application (COAD-IM) with two other versions: (i) ST-IM, where the image parameters are fixed (described in Section VI.B.2), and (ii) NW-IM, where the image TL and QL parameters are adapted taking only the network bandwidth availability into account.

VI.E.2 Impact on Total Latency and PSNR

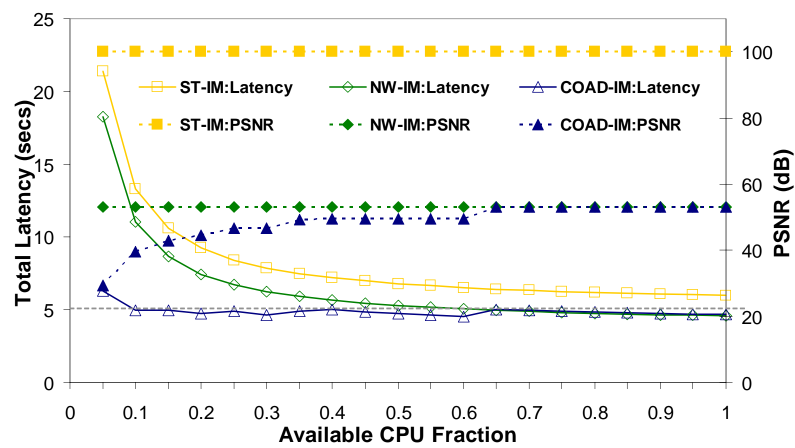
We study the total latency and $PSNR$ of the image application under different network bandwidths and CPU availabilities for two cases: (i) where the other applications executing on the platform impose constant processing load over time, and (ii) where the load imposed on the platform varies at random. The total latency constraint for the image application was set at 5 seconds.

Constant Load

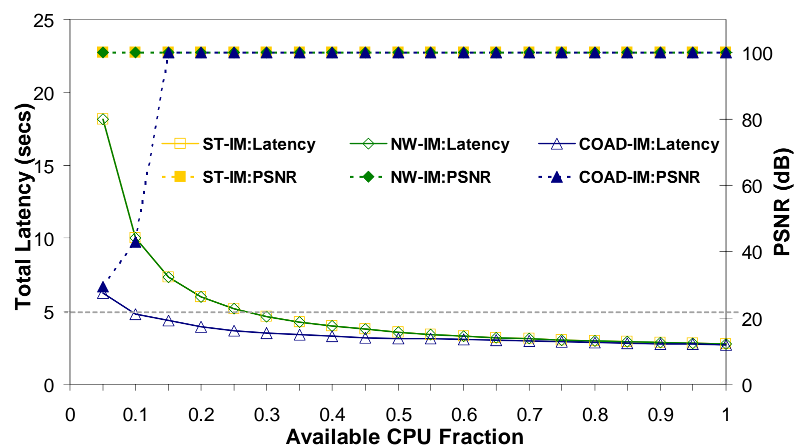
Figure VI.8 illustrates the total latency and the corresponding $PSNR$ of the image application under three different network bandwidths and for different platform CPU availabilities, for the three versions of the image application. The solid lines plot the total latency (primary Y-axis), and the dotted lines plot the $PSNR$ (secondary Y-axis). $PSNR = \infty$ (highest image quality) is plotted at 100 dB for reference. From the graphs, we observe that ST-IM results in $PSNR = \infty$ for all cases, but fails to meet the latency constraint at all CPU availabilities at 50 Kbps (Figure VI.8(a)) and 300 Kbps (Figure VI.8(b)). At 800 Kbps (Figure VI.8(c)), it meets the constraint only when more than 25% of the CPU is available. In the NW-IM version, the quality of the image is regulated, depending only on the network bandwidth (from 38.42 dB at 50 Kbps, to ∞ at 800 Kbps). However, in cases where the CPU availability is low, the total latency often fails to meet the constraint (*e.g.*, 11 sec at 300 Kbps and 10% CPU). COAD-IM is able to meet the latency constraint at almost all the points considered. For a given network bandwidth, the $PSNR$ under COAD-IM increases with the CPU availability, until it converges with the plot for the NW-IM version (when the transmission latency starts outweighing the decompression latency). These results show that through better param-



(a) Latency and PSNR at 50 Kbps



(b) Latency and PSNR at 300 Kbps



(c) Latency and PSNR at 800 Kbps

Figure VI.8: Total Latency and $PSNR$ for different versions of the image application under varying network bandwidth and CPU availability

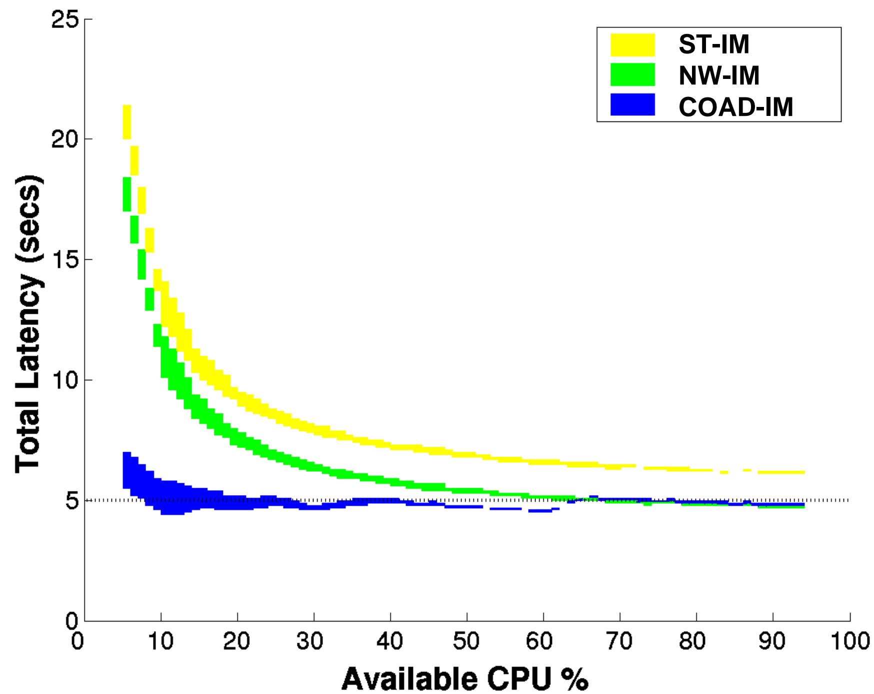
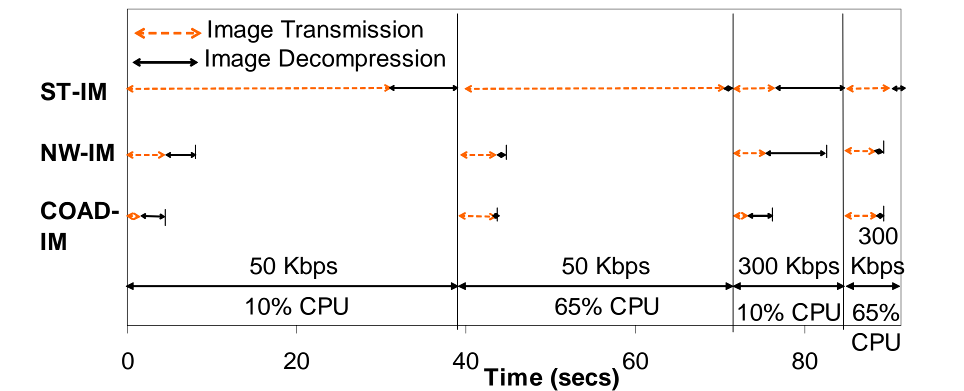


Figure VI.9: Distribution of total latency for image application under uniform random variation of CPU availability

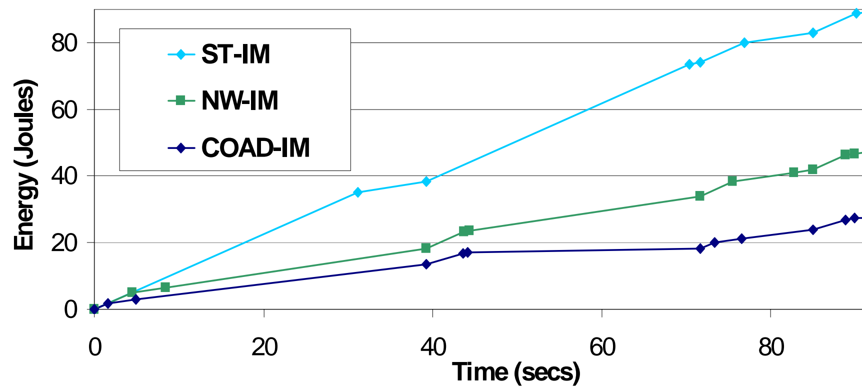
eter selection, COAD-IM is able to meet the latency constraint under a wide variety of bandwidths and CPU availability, while minimizing the loss in image quality.

Random Load

In reality, the processing load imposed by the applications on the platform may vary dynamically, and hence using average CPU availability to guide the parameter selection decisions (instead of worst-case) could lead to violations of the latency constraint. To evaluate the impact of this, we executed the three versions of the image application along with another application having a period of 100 ms and whose actual CPU requirements were uniformly randomly distributed about its average CPU requirement. Figure VI.9 shows the distribution of the total latency of the image application at 300 Kbps under different average CPU availabilities for all the three versions. The shaded regions correspond to the latency values within which 99% of the total number



(a) Image application under varying bandwidths and CPU availability



(b) Energy consumption under different cases

Figure VI.10: Energy savings under co-adaptation

of measured latency values lie. Note that, the shape of this graph resembles the plot in Figure VI.8(b) (in which the other applications imposed constant load). This shows that the actual latency values are very closely distributed about the expected latency values. In addition, the spread of the total latency values decreases with increasing CPU availability. These results indicate that the co-adaptation techniques perform well even under variations in the actual CPU availability.

VI.E.3 Impact on Energy Consumption

We next evaluate the energy savings made possible by the proposed co-adaptation based approach. Figure VI.10(a) illustrates a timeline for the execution of the three versions of the image application under varying network bandwidth and CPU

availability. The dotted arrows correspond to image transmission and the solid arrows correspond to image decompression. Figure VI.10(b) shows the cumulative energy consumption over the same timeline for the three versions of the image application. From the graph, we observe that NW-IM achieves 47% energy savings over ST-IM. This is mainly due to energy savings in the network card owing to decreased transmission times. However, COAD-IM results in large savings in both the network interface card (due to decreased transmission times) and the processor (due to setting of platform frequency and voltage in a manner that is aware of the application parameters), leading to large energy savings overall: 42%, compared to NW-IM, and 70%, compared to ST-IM.

VI.F Conclusions

In this chapter, we presented application-architecture co-adaptation, a methodology for the dynamic and synergistic customization of both applications as well as the underlying platform architecture. We illustrated the overall concept of co-adaptation, and described it in detail in the context of a wireless image delivery system. Experimental results indicate that the proposed techniques successfully achieve large benefits in application performance, quality, and energy efficiency under dynamic variation of platform processing resources and network bandwidth.

VII

Future Research Directions

As described in this thesis, it is becoming economically infeasible to develop custom SoCs/ASICs due to the high cost associated with their design, verification, manufacture and test. Platform-based SoCs, which consist of largely pre-designed and pre-verified standard components, are emerging as an attractive alternative, since they can be targeted towards multiple applications, thereby amortizing the cost of platform development over larger markets. However, the widespread adoption of such platforms is limited by concerns about their performance and energy-efficiency. This thesis addressed the problem of enabling the use of platforms in domains where custom approaches have traditionally been used, by provisioning for dynamic configurability in platform components. It introduced the concept of Dynamic Platform Management for the run-time customization of configurable platforms, depending on the time-varying requirements imposed by the executing applications. The techniques proposed in this thesis enabled superior application performance, more efficient utilization of platform resources, and improved energy efficiency compared to conventional statically configured platforms. We also investigated application-architecture co-adaptation techniques for the co-ordinated adaptation of both the executing applications as well as the underlying platform architecture. In this final chapter, we discuss future research directions emerging out of the work presented in this thesis, and point out the challenges and opportunities offered by them.

In this thesis, we studied adaptivity in platform architectures and applications. However, adaptivity can be exploited at different levels of system design, and across different levels, each providing different benefits and research challenges. Hence, moving forward, we envision three main directions along which this work can be extended, as illustrated in Figure VII.1: (i) architectural-level adaptation, (ii) cross-layer adaptation, and (iii) network-wide adaptation. We next discuss each of these in further detail.

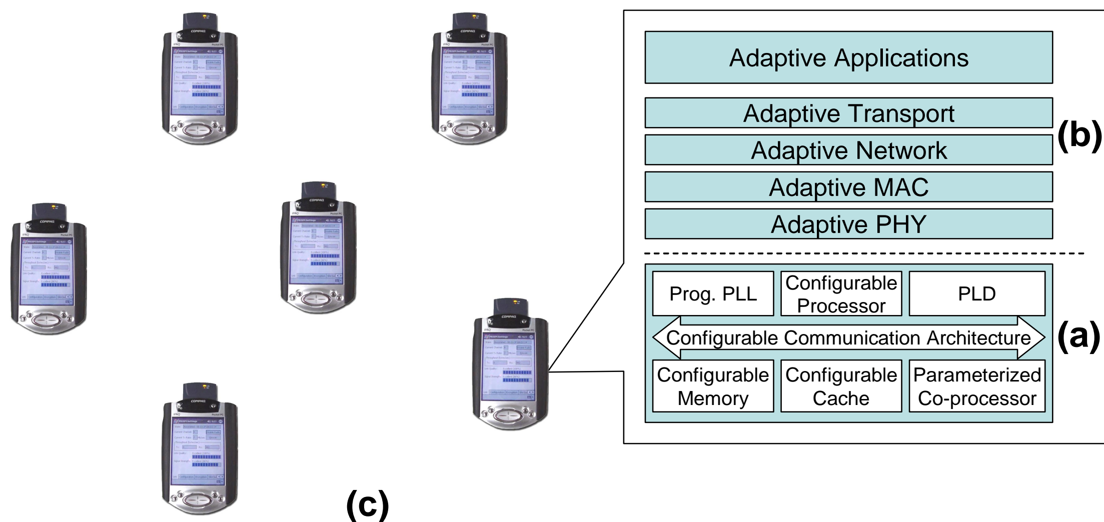


Figure VII.1: Future research directions: (a) architectural-level adaptation, (b) cross-layer adaptation, and (c) network-wide adaptation

VII..1 Architectural-Level Adaptation

In this thesis, we surveyed numerous technologies that have been developed in recent years to enable the dynamic configuration of platform components and architectural parameters. Many of these technologies have reached relative maturity, a few of them having made their appearance in commercial products. We also proposed novel techniques to provision for configurability in the on-chip bus architecture and in the memory subsystem. However, incorporating more configurability in platforms would enable them to be better adapted to the characteristics of the executing applications.

Hence, there is a need to (i) develop technologies for providing more configurability features in platform components, and (ii) design platforms featuring multiple configurability options together (Figure VII.1).

This thesis also described dynamic platform management techniques for the integrated run-time optimization of a few platform components together. We illustrated the importance of holistic platform management that takes into account the interdependence between the configurations of the individual platform components. Similar platform management techniques are required for platforms featuring other avenues of run-time configuration. Therefore, an important research challenge that needs to be addressed lies in the development of comprehensive platform management techniques for the co-ordinated adaptation of multiple platform components together.

VII..2 Cross-Layer Adaptation

This thesis explored application-architecture co-adaptation, for the synergistic adaptation of both the executing applications as well as the underlying platform architecture, and described it in the context of a wireless image delivery system. Such co-adaptation techniques need to be generalized to encompass other configurable applications and architectural features. Also, previous work has studied the adaptation of different layers of the network protocol stack, such as the transport layer [193], network layer [194], medium access control (MAC) layer [195] and the physical layer [196]. However, for optimized system operation, there is a need for co-ordinated adaptation across all these protocol layers, the executing applications and the platform architecture (Figure VII.1). This calls for new *cross-layer* adaptation methodologies that can understand the interactions between the different layers of system operation, by passing appropriate information between the different layers, and adapt the system as a whole. We believe that such system-wide integrated adaptation will provide significant improvements in application performance, optimized usage of network and platform resources, and large energy savings.

VII..3 Network-Wide Adaptation

Finally, the work presented in this thesis was limited to providing and exploiting adaptivity within an embedded system node. However, these nodes may be connected as a network of nodes (Figure VII.1). An example of such a network is a wireless sensor network. Such networks are characterized by the following requirements:

- They may need to measure (sense) and process different types of data at different times.
- Upgrading or adding new software and functionality to the nodes may be required, after the network is deployed.
- Different nodes may need to be shut-down or put in low-power modes over time to save energy, without adversely affecting the data collection process.
- When some nodes fail or are disabled, the functionality of the nodes and the routing of data within the network may need to be changed.

Addressing these requirements calls for the development of new *network-wide* adaptation techniques for the co-ordinated adaptation of the different nodes in the network. These techniques may either be centralized, with one node making all adaptation decisions, or distributed, where each node makes adaptation decisions in cooperation with the other nodes. Such network-wide adaptation techniques will enable the deployment of networks that can support a diverse set of functionality at low-cost and with a long network lifetime.

Bibliography

[1] T. Pering, T. Burd, and R. Brodersen, "Voltage scheduling in the lpARM microprocessor system," in *Proc. Int. Symp. Low Power Electronics & Design*, pp. 96–101, July 2000.

[2] "International Technology Roadmap for Semiconductors, 2003 Edition." http://public.itrs.net/Files/2003ITRS/Home2003.htm.

[3] "Moore's Law." http://www.intel.com/technology/silicon/mooreslaw/.

[4] "GeForce 7800 FAQ." http://www.nvidia.com/page/geforce_7800_faq.html.

[5] W. Trybula, "A common base for mask cost of ownership," in *Proceeedings of SPIE Symposium on Photomask Technology*, pp. 318–323, Sept. 2003.

[6] "International Technology Roadmap for Semiconductors, 1999 Edition." http://public.itrs.net/Files/1999_SIA_Roadmap/Home.htm.

[7] A. Khan, "Challenges in achieving first-silicon success for 10M-gate SoCs: a silicon engineering perspective." Special session, Design Automation Conference, http://www.dac.com/39th/39acceptedpapers.nsf/browse, June 2002.

[8] C. Souza, "Semiconductor IP houses struggle to survive as ASIC design starts continue to dwindle." http://www.my-esm.com/showArticle.jhtml?articleID=10100072, May 2003.

[9] "Nomadik multimedia processor." http://www.st.com/stonline/prodpres/dedicate/proc/proc.htm.

[10] "Platform-based design taxonomy, Version 1.00 (PBD 1 1.0)." VSIA Platform-Based Design DWG, http://www.vsi.org/documents/directory.htm#pbd110, Dec. 2003.

[11] "Instant Silicon Solution Platform (Structured ASIC)." http://www.necel.com/issp/english/index.html.

[12] "Chipx, inc.." http://www.chipx.com.

[13] "Rapidchip." http://www.lsilogic.com/products/rapidchip_platform_asic/index.html.

[14] "Panel:(When) Will FPGAs Kill ASICS?," in *Proc. Design Automation Conference*, pp. 321–322, June 2001.

[15] "Altera corp.." http://www.altera.com.

[16] "Xilinx inc.." http://www.xilinx.com.

[17] "OMAP Platform." http://focus.ti.com/omap/docs/omaphomepage.tsp.

[18] "Intel Network Processors." http://www.intel.com/design/network/products/npfamily.

[19] "Tensilica Xtensa processors." http://www.tensilica.com.

[20] "Nios Embedded Processor." http://www.altera.com/literature/lit-nio.html.

[21] "MIPS 32 4KE Family Specifications." http://www.mips.com/content/Products/Cores/32-BitCores/MIPS324KEFamily%/.

[22] "AMBA 2.0 Specification." http://www.arm.com/products/solutions/AMBA_Spec.html.

[23] "CoreConnect Bus Architecture." http://www-03.ibm.com/chips/products/coreconnect/.

[24] "Pico express." http://www.synfora.com/products/picoexpress.html.

[25] T. D. Givargis and F. Vahid, "Platune: a tuning framework for system-on-a-chip platform," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, pp. 1317–1327, Nov. 2002.

[26] "Quicksilver Technology." http://www.qstech.com.

[27] "PACT XPP Technologies." http://www.pactcorp.com.

[28] "Excalibur embedded processor solutions." http://www.altera.com/products/devices/excalibur/exc-index.html.

[29] "Nexperia Mobile Solutions." http://www.semiconductors.philips.com/markets/communications/nexperia/%.

[30] "ARM926EJ-S PrimeXsys Platform." http://www.arm.com/products/solutions/sysp-primexsys-arm926ej-s.html.

[31] "TI Applications Processing Solutions for All Mobile Market Segments." `http://focus.ti.com/general/docs/wtbu/wtbugencontent.tsp?templateId=61%23&navigationId=11956&path=templatedata/cm/general/data/`.

[32] "Crescendo - Media Processing Solution Kits." `http://www.improvsys.com/SolutionKits/Crescendo.cfm`.

[33] R. I. Bahar and S. Manne, "Power and energy reduction via pipeline balancing," in *Proc. Int. Symp. Computer Architecture*, pp. 218–229, July 2001.

[34] R. Maro, Y. Bai, and R. I. Bahar, "Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors," in *Proc. Workshop Power-Aware Computer Systems*, pp. 97–111, Nov. 2000.

[35] A. Iyer and D. Marculescu, "Power aware microarchitecture resource scaling," in *Proc. Design Automation & Test Europe (DATE) Conference*, pp. 190–196, Mar. 2001.

[36] D. Folegnani and A. Gonzlez, "Energy-effective issue logic," in *Proc. Int. Symp. Computer Architecture*, pp. 230–239, July 2001.

[37] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi, "An adaptive issue queue for reduced power at high performance," in *Proc. Workshop Power-Aware Computer Systems*, pp. 25–39, Nov. 2000.

[38] D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources," in *Proc. Int. Symp. Microarchitecture*, pp. 90–101, Dec. 2001.

[39] S. Ghiasi, J. Casmira, and D. Grunwald, "Using IPC variation in workloads with externally specified rates to reduce power consumption," in *Proc. Workshop Complexity-Effective Design*, June 2000.

[40] S. Manne, A. Klauser, and D. Grunwald, "Pipeline gating: speculation control for energy reduction," in *Proc. Int. Symp. Computer Architecture*, pp. 132–141, July 1998.

[41] D. Brooks and M. Martonosi, "Adaptive thermal management for high-performance microprocessors," in *Proc. Workshop Complexity-Effective Design*, June 2000.

[42] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proc. Int. Symp. Microarchitecture*, pp. 172–180, Dec. 1994.

[43] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera reconfigurable functional unit," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pp. 87–96, 1997.

[44] D. W. Wall, "Limits of instruction-level parallelism," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 176–188, Apr. 1991.

[45] A. R. Chandrakasan and R. W. Brodersen, *Low Power Digital CMOS Design*. Kluwer Academic Publishers, Norwell, MA, 1995.

[46] A. J. Stratakos, S. R. Sanders, and R. W. Brodersen, "A low-voltage CMOS DC-DC converter for a portable battery-operated system," in *Proc. IEEE Power Electronics Specialist Conf.*, pp. 619–626, Apr. 1994.

[47] W. Namgoong, M. Yu, and T. Meng, "A high-efficiency variable-voltage CMOS dynamic DC-DC switching regulator," in *Proc. Int. Solid-State Circuits Conference*, pp. 380–381, Feb. 1997.

[48] M. Fleischmann, "LongRun Power Management: Dynamic power management for Crusoe processors." `http://www.transmeta.com/pdfs/paper_mfleischmann_17jan01.pdf`.

[49] "Intel XScale Technology." `http://www.intel.com/design/intelxscale/`.

[50] "AMD PowerNow Technology." `http://www.amd.com/us-en/assets/content_type/DownloadableAssets/Power_%Now2.pdf`.

[51] W. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Proc. USENIX Symposium on Operating Systems Design and Implementation*, pp. 13–23, Nov. 1994.

[52] K. Govil, E. Chan, and H. Wasserman, "Comparing algorithms for dynamic speed-setting of a low-power CPU," in *Proc. Int. Conf. Mobile Computing and Networking*, pp. 13–25, Nov. 1995.

[53] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proc. Int. Symp. Low Power Electronics & Design*, pp. 76–81, Aug. 1998.

[54] D. Grunwald, P. Levis, K. Farkas, C. B. Morrey, and M. Neufeld, "Policies for dynamic clock scheduling," in *Proc. Symp. Operating Systems Design and Implementation*, pp. 73–86, Oct. 2000.

[55] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced CPU energy," in *Proc. Symp. Foundations of Computer Science*, pp. 374–382, Oct. 1995.

[56] Y. Shin and K. Choi, "Power conscious fixed priority scheduling for hard real-time systems," in *Proc. Design Automation Conference*, pp. 134–139, June 1999.

[57] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava, "Power optimization of variable-voltage core-based systems," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 1702–1714, Dec. 1999.

[58] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proc. Symp. Operating Systems Principles*, pp. 89–102, Oct. 2001.

[59] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas, "A framework for dynamic energy efficiency and temperature management," in *Proc. Int. Symp. Microarchitecture*, pp. 202–213, Dec. 2000.

[60] C. J. Hughes, J. Srinivasan, and S. V. Adve, "Saving energy with architectural and frequency adaptations for multimedia applications," in *Proc. Int. Symp. Microarchitecture*, pp. 250–261, Dec. 2001.

[61] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Proc. Int. Symp. Computer Architecture*, pp. 233–244, May 2002.

[62] M. C. Huang, J. Renau, and J. Torrellas, "Positional adaptation of processors: application to energy reduction," in *Proc. Int. Symp. Computer Architecture*, pp. 157–168, June 2003.

[63] A. Malik, B. Moyer, and D. Cermak, "A programmable unified cache architecture for embedded applications," in *Proc. Int. Conf. Compilers, Architecture and Synthesis for Embedded Systems*, pp. 165–171, Nov. 2000.

[64] D. H. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *Proc. Int. Symp. Microarchitecture*, pp. 248–259, Nov. 1999.

[65] S. H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar, "An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance I-caches," in *Proc. Int. Symp. High-Performance Computer Architecture*, pp. 147–157, Jan. 2001.

[66] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache architecture for embedded systems," in *Proc. Int. Symp. Computer Architecture*, pp. 136–146, June 2003.

[67] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji, "Adapting cache line size to application behavior," in *Proc. Int. Conf. Supercomputing*, pp. 145–154, June 1999.

[68] D. T. Chiou, *Extending the reach of microprocessors: column and curious caching*. PhD thesis, Massachusetts Institute of Technology, 1999.

[69] P. Ranganathan, S. Adve, and N. P. Jouppi, "Reconfigurable caches and their application to media processing," in *Proc. Int. Symp. Computer Architecture*, pp. 214–224, June 2000.

[70] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *Proc. Int. Symp. Microarchitecture*, pp. 245–257, Dec. 2000.

[71] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. L. Scott, "Integrating adaptive on-chip storage structures for reduced dynamic power," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, pp. 141–152, Sept. 2002.

[72] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic cache partitioning for simultaneous multithreading systems," in *Proc. IASTED Int. Conf. Parallel and Distributed Computing and Systems*, pp. 635–641, Aug. 2001.

[73] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM Trans. Computer Systems*, vol. 10, pp. 338–359, Nov. 1992.

[74] C.-K. Luk and T. C. Mowry, "Memory forwarding: enabling aggressive layout optimizations by guaranteeing the safety of data relocation," in *Proc. Int. Symp. Computer Architecture*, pp. 88–99, May 1999.

[75] M. Kandemir and I. Kadayif, "Compiler-directed selection of dynamic memory layouts," in *Proc. Int. Symp. Hardware/Software Codesign*, pp. 219–224, Apr. 2001.

[76] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 63–74, Apr. 1991.

[77] O. Temam, E. D. Granston, and W. Jalby, "To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts," in *Proc. ACM/IEEE Conf. Supercomputing*, pp. 410–419, 1993.

[78] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Cache-conscious structure layout," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 1–12, May 1999.

[79] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 229–241, May 1999.

[80] R. Banakar, S. Steinke, B. S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proc. Int. Symp. Hardware/Software Codesign*, pp. 73–78, May 2002.

[81] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic management of scratch-pad memory space," in *Proc. Design Automation Conference*, pp. 690–695, June 2001.

[82] "Rambus 128/144-Mbit Direct RDRAM Datasheet." http://www.rambus.com/downloads/rdram.128s.0059-1.11.book.pdf.

[83] "Samsung DRAM Datasheets." http://www.samsung.com/Products/Semiconductor/DRAM/TechnicalInfo/DataS%heets.htm.

[84] "Intel 440BX AGPset: 82443BX Host Bridge/Controller Data Sheet." http://www.intel.com/design/chipsets/datashts/29063301.pdf, Apr. 1998.

[85] "Intel 850 Chip Set." http://www.intel.com/design/chipsets/850/.

[86] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin, "DRAM energy management using software and hardware directed power mode control," in *Proc. Int. Symp. High-Performance Computer Architecture*, pp. 159–169, Jan. 2001.

[87] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, "Power aware page allocation," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 105–116, Nov. 2000.

[88] V. D. L. Luz, M. Kandemir, and I. Kolcu, "Automatic data migration for reducing energy consumption in multi-bank memory systems," in *Proc. Design Automation Conference*, pp. 213–218, June 2002.

[89] "Sonics μNetworks Technical Overview." http://www.sonicsinc.com/sonics/support/documentation/whitepapers/data%/Overview.pdf.

[90] R. Bashirullah, W. Liu, and R. K. Cavin, "Low-power design methodology for an on-chip bus with adaptive bandwidth capability," in *Proc. Design Automation Conference*, pp. 628–633, June 2003.

[91] K.Lahiri, A.Raghunathan, and S.Dey, "Design of high-performance system-on-chips using communication architecture tuners," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 6, pp. 919–932, 2004.

[92] "Virtex-II Pro / Pro X FPGAs." http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_i%i_pro_fpgas/.

[93] "Configurable System-on-Chip Platforms." http://www.triscend.com.

[94] "Atmel Field Programmable System Level Integrated Circuits (FPSLIC)." http://www.atmel.com/products/FPSLIC/.

[95] "Elixent: The D-Fabrix Array." `http://www.elixent.com/products/array.htm`.

[96] "Toshiba Media Embedded Processor (MeP)." `http://www.mepcore.com/english/`.

[97] "Press Release: Toshiba and Elixent develop SoC RISC processor for post-production reconfiguration of multimedia applications." `http://www.mepcore.com/english/frameset5_e.html`.

[98] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pp. 12–21, Apr. 1997.

[99] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," in *Proc. Design Automation Conference*, pp. 507–512, June 2000.

[100] G. Stitt, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: a first approach," in *Proc. Design Automation Conference*, pp. 250–255, June 2003.

[101] "Linux kernel CPUfreq subsystem." `http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufreq.html`.

[102] R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," *Proc. IEEE*, vol. 89, pp. 490–504, Apr. 2001.

[103] "Smart interconnects for rapid SoC development." Sonics Inc., `http://www.sonicsinc.com`.

[104] S. J. Lee, S. J. Song, K. Lee, J. H. Woo, S. E. Kim, B. G. Nam, and H. J. Yoo, "An 800Mhz star-connected on-chip network for application to system on a chip," in *Proc. Int. Solid-State Circuits Conference*, pp. 468–469, Feb. 2003.

[105] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C. A. Zeferino, "SPIN: a scalable, packet switched, on-chip micro-network," in *Proc. Design Automation & Test Europe (DATE) Conference*, pp. 70–73, 2003.

[106] F. Karim, A. Nguyen, and S. Dey, "An interconnect architecture for networking system on chips," *IEEE Micro*, vol. 22, pp. 36–45, Oct. 2002.

[107] S. I. Han, A. Baghdadi, M. Bonaciu, S. I. Chae, and A. A. Jerraya, "An efficient scalable and flexible data transfer architecture for multiprocessor SoC with massive distributed memory," in *Proc. Design Automation Conference*, pp. 250–255, June 2004.

[108] D. Wingard and A. Kurosawa, "Integration architecture for system-on-a-chip design," in *Proc. Custom Integrated Circuits Conference*, pp. 85–88, May 1998.

[109] R. Yoshimura, K. T. Boon, T. Ogawa, S. Hatanaka, T. Matsuoka, and K. Taniguchi, "DS-CDMA wired bus with simple interconnection topology for parallel processing system LSIs," in *Proc. Int. Solid-State Circuits Conference*, pp. 370–371, Feb. 2000.

[110] K. Lahiri, A. Raghunathan, and G. Lakshminarayana, "LOTTERYBUS: a new communication architecture for high-performance system-on-chip designs," in *Proc. Design Automation Conference*, pp. 15–20, June 2001.

[111] T. Meyerowitz, C. Pinello, and A. Sangiovanni-Vincentelli, "A tool for describing and evaluating hierarchical real-time bus scheduling policies," in *Proc. Design Automation Conference*, pp. 312–317, June 2003.

[112] R. B. Ortega and G. Borriello, "Communication synthesis for distributed embedded systems," in *Proc. Int. Conf. Computer-Aided Design*, pp. 437–444, Nov. 1998.

[113] A. Pinto, L. P. Carloni, and A. Sangiovanni-Vincentelli, "Constraint-driven communication synthesis," in *Proc. Design Automation Conference*, pp. 783–788, June 2002.

[114] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Fast exploration of bus-based on-chip communication architectures," in *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, pp. 242–247, Sept. 2004.

[115] J. Hu and R. Marculescu, "DyAD — smart routing for networks-on-chip," in *Proc. Design Automation Conference*, pp. 260–263, June 2004.

[116] J. A. Rowson and A. Sangiovanni-Vincentelli, "Interface-based design ," in *Proc. Design Automation Conference*, pp. 178–183, June 1997.

[117] M. Caldari, M. Conti, M. Coppola, S. Curaba, S. Pieralisi, and C. Turchetti, "Transaction-level models for AMBA bus architecture using SystemC 2.0," in *Proc. Design Automation & Test Europe (DATE) Conference*, pp. 26–31, 2003.

[118] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 1059–1076, Sept. 2001.

[119] "Virtual component interface standard, Version 2 (OCB 2 2.0." VSIA On-Chip Bus DWG, `http://www.vsia.org/documents/vsiadocuments.htm#ocb220`.

[120] "Open core protocol international partnership (OCP-IP)." `http://www.ocpip.org`.

[121] "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications." ANSI/IEEE Std 802.11, 1999 Edition (R2003)

http://standards.ieee.org/getieee802/download/802.11-1999.pdf.

[122] "ARM9E Family: ARM946E-S." http://www.arm.com/products/CPUs/ARM946E-S.html.

[123] "Modelsim 5.7e." http://www.model.com.

[124] R. Mahmud, "Techniques to make clock switching glitch free." [Online]. Available: http://www.eetimes.com/story/0EG20030626S0035.

[125] L. Benini, A. Bogliolo, G. A. Paleologo, and G. D. Micheli, "Policy optimization for dynamic power management," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 813–833, June 1999.

[126] F. Douglis, P. Krishnan, and B. Bershad, "Adaptive disk spin-down policies for mobile computers," in *USENIX Symp. Mobile and Location Independent Computing*, pp. 121–137, Apr. 1995.

[127] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman & Company, San Francisco, CA, 1979.

[128] K. Lahiri, A. Raghunathan, and S. Dey, "Design space exploration for optimizing on-chip communication architectures," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, pp. 952–961, June 2004.

[129] "Synopsys DesignWare Intellectual Property." http://www.synopsys.com/products/designware/designware.html.

[130] W. Dai, L. Wu, and S. Zhang, "UCSC Floorplanning Tool." http://www.soe.ucsc.edu/research/surf/GSRC/progress.html.

[131] "ARM Foundary Program." http://www.arm.com/community/ARM_silicon_design/Foundry.html.

[132] " Design Compiler 2003.12, Synopsys Inc.." http://www.synopsys.com/products/logic/design_compiler.html.

[133] "NEC Cell-based IC: CB-12 L/M/H Type (Features/Basic Specifications)." http://www.necel.com/cbic/en/cb12/cb12.html.

[134] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*. Addison-Wesley, Menlo Park, CA, 1990.

[135] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes," pp. 1064–1070, May 1993.

[136] "Universal Mobile Telecommunications Systems (UMTS); Multiplexing and channel coding (FDD)." 3GPP TS 25.212 version 3.4.0 Release 1999 http://www.3gpp.org/ftp/Specs/archive/25_series/25.212/25212-340.zip.

[137] "Physical Layer Standard for cdma2000 Spread Spectrum Systems." 3GPP2 C.S0002-0, Version 1.0, July 1999 http://www.3gpp2.org/Public_html/specs/C.S0002-0_v1.0.pdf.

[138] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: Turbo codes," *IEEE Trans. Communications*, vol. 44, pp. 1261–1271, Oct. 1996.

[139] M. C. Valenti and J. Sun, "The UMTS Turbo code and an efficient decoder implementation suitable for software-defined radios," *Int. Journal Wireless Information Networks*, vol. 8, pp. 203–215, Oct. 2001.

[140] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," pp. 1009–1013, June 1995.

[141] G. Masera, G. Piccinini, M. Roch, and M. Zamboni, "VLSI architectures for turbo codes," *IEEE Trans. VLSI Systems*, vol. 7, pp. 369–379, Sept. 1999.

[142] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEEE Journal Solid-State Circuits*, vol. 27, pp. 473–484, Apr. 1992.

[143] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems ," in *Proc. Int. Symp. Hardware/Software Codesign*, pp. 73–78, May 2002.

[144] "ARM966E-S: Embedded core with flexible memory system & DSP instruction set extensions." http://www.arm.com/products/CPUs/ARM966ES.html.

[145] "Intel XScale Microarchitecture." ftp://download.intel.com/design/intelxscale/XScaleDatasheet4.pdf.

[146] "ARM922T Technical Reference Manual." http://www.arm.com/pdfs/DDI0184B_922T_TRM.pdf.

[147] "Specifications of the 3GPP Confidentiality and Integrity Algorithms (Documents 1 and 2)." http://www.3gpp.org/TB/Other/algorithms.htm.

[148] K. Balachandran, "Convergence of 3G and WLAN." IEEE Intl. Conf. on Communications, May 2002, http://www.icc2002.com/notes.html.

[149] J. Blyler, "3G/WLAN connectivity becomes a reality." January/February 2003, http://www.wsdmag.com/Articles/ArticleID/6695/6695.html.

[150] "Intel SA-1110 Processor." http://www.intel.com/design/pca/applicationsprocessors/1110_brf.htm.

[151] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram, "Frame-based dynamic voltage and frequency scaling for an MPEG decoder," in *Proc. Int. Conf. Computer-Aided Design*, pp. 732–737, Nov. 2002.

[152] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[153] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. D. Micheli, "Dynamic voltage scaling and power management for portable systems," in *Proc. Design Automation Conference*, pp. 524–529, June 2001.

[154] A. Sinha and A. P. Chandrakasan, "JouleTrack - a web based tool for software energy profiling," in *Proc. Design Automation Conference*, pp. 220–225, June 2001.

[155] "ARM Developer Suite (ADS) version 1.2." http://www.arm.com/products/DevTools/ADS.html.

[156] T. Pering and R. Brodersen, "Dynamic voltage scaling and the design of a low-power microprocessor system," in *Proc. Power Driven Microarchitecture Workshop*, June 1998.

[157] "Excalibur Device Overview." http://www.altera.com/literature/ds/ds_arm.pdf.

[158] "EPXA1 Development Board Hardware Reference Manual." http://www.altera.com/literature/manual/mnl_epxa1_devbd.pdf.

[159] P. R. Panda, N. D. Dutt, A. Nicolau, F. Catthoor, A. Vandecappelle, E. Brockmeyer, C. Kulkarni, and E. D. Greef, "Data memory organization and optimization in application-specific systems," *IEEE Design & Test Magazine*, vol. 18, pp. 56–68, May 2001.

[160] "Philips debuts reference design for converged handsets." http://www.eet.com/news/latest/showArticle.jhtml?articleID=159402638, Mar. 2005.

[161] K. Sekar, K. Lahiri, A. Raghunathan, and S. Dey, "FLEXBUS: a high-performance system-on-chip communication architecture with a dynamically condfigurable topology," in *Proc. Design Automation Conference*, pp. 571–574, June 2005.

[162] T. D. Givargis, F. Vahid, and J. Henkel, "Evaluating power consumption of parameterized cache and bus architectures in system-on-a-chip designs," *IEEE Trans. VLSI Systems*, vol. 9, pp. 500–508, Aug. 2001.

[163] P. Grun, N. Dutt, and A. Nicolau, "Memory system connectivity exploration," in *Proc. Design Automation & Test Europe (DATE) Conference*, pp. 894–901, Mar. 2002.

[164] A. Papanikolau, K. Koppenberger, M.Miranda, and F. Cathoor, "Memory communication network exploration for low-power distributed memory organizations," in *Proc. IEEE Workshop Signal Processing Systems*, pp. 176–181, Oct. 2004.

[165] P. Knudsen and J. Madsen, "Integrating communication protocol selection with hardware/software codesign," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 1077–1095, Aug. 1999.

[166] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Trans. Information Theory*, vol. 13, pp. 260–269, Apr. 1967.

[167] "Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications." IEEE Std 802.11a-1999 `http://standards.ieee.org/getieee802/download/802.11a-1999.pdf`.

[168] "Part 16: Air Interface for Fixed Broadband Wireless Access Systems." IEEE Std 802.16-2004 `http://standards.ieee.org/getieee802/download/802.16-2004.pdf`.

[169] B. Vucetic and J. Yuan, *Turbo Codes: Principles and Applications*. Kluwer Academic Publishers, Norwell, MA, 2000.

[170] A. Aguiar and J. Klaue, "Bi-directional WLAN channel measurements in different mobility scenarios," in *Proc. IEEE Vehicular Technology Conf.*, pp. 64–68, May 2004.

[171] V. Gupta and S. Gupta, "KSSL: Experiments in wireless internet security." `http://research.sun.com/techrep/2001/smli_tr-2001-103.pdf`.

[172] "Open GL for Embedded Systems (OpenGL ES)." `http://www.khronos.org/opengles/`.

[173] J. J. Rushanan, "AWIC: Adaptive Wavelet Image Compression for Still Image." MTR-97B0000041, The MITRE Corporation, Bedford, MA, Sept. 1997.

[174] "RC6 Block Cipher ." `http://www.rsasecurity.com/rsalabs/node.asp?id=2512`.

[175] "MPEG 4 ISO/IEC International Standard: Overview." http://www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm.

[176] "SSL 3.0 Specification." http://wp.netscape.com/eng/ssl3/.

[177] N. R. Potlapally, S. Ravi, and A. Raghunathan, "System design methodologies for a wireless security processing platform," in *Proc. Design Automation Conference*, pp. 777–782, June 2002.

[178] "Sierra Wireless AirCard 555 Wireless Network Card." http://www.sierrawireless.com/ProductsOrdering/spec_555.asp.

[179] J. P. Woodard and L. Hanzo, "Comparative study of turbo decoding techniques: an overview," *IEEE Trans. Vehicular Technology*, vol. 49, pp. 2208–2233, Nov. 2000.

[180] S. Contini, R. L. Rivest, M. J. B. Robshaw, and Y. L. Yin, "The security of the RC6 block cipher." ftp://ftp.rsasecurity.com/pub/rsalabs/rc6/security.pdf, Aug. 1998.

[181] A. Vetro, J. Cai, and C. W. Chen, "Rate-reduction transcoding design for wireless video streaming," *Journal Wireless Communications and Mobile Computing*, vol. 2, pp. 625–641, Sept. 2002.

[182] D. G. Lee, D. Panigrahi, and S. Dey, "Network-aware image data shaping for low-latency and energy-efficient data services over the Palm wireless network," in *Proc. World Wireless Congress (3G Wireless)*, May 2003.

[183] "PacketVideo multimedia technology overview: standards, algorithms, and implementations." http://www.packetvideo.com/pdf/pv_whitepaper.pdf.

[184] "Real System Production Guide." http://service.real.com/help/library/guides/production8/htmfiles/intro%.htm.

[185] W. Yuan, K. Nahrstedt, S. Adve, D. Jones, and R. Kravets, "Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems," in *Proc. SPIE/ACM Multimedia Computing and Networking Conference*, pp. 1–13, Jan. 2003.

[186] S. Mohapatra and R. Cornea and H. Oh and K. Lee and M. Kim and N. Dutt and R. Gupta and A. Nicolau and S. Shukla and N. Venkatasubramanian, "A cross-layer approach for power-performance optimization in distributed mobile systems," in *Proc. Int. Parallel and Distributed Processing Symp.*, p. 218a, Apr. 2005.

[187] D. Li and P. H. Chou, "Application/architecture power co-optimization for embedded systems powered by renewable sources," in *Proc. Design Automation Conference*, pp. 618–623, June 2005.

[188] "The Familiar Project." http://familiar.handhelds.org.

[189] "The Itsy Pocket Computer." http://research.compaq.com/projects/Itsy/itsy.html.

[190] "The Berkeley MPEG Player." http://bmrc.berkeley.edu/frame/research/mpeg/mpeg_play.html.

[191] "Iperf Version 1.7.0." http://dast.nlanr.net/Projects/Iperf/.

[192] "The USC-SIPI Image Database." http://sipi.usc.edu/services/database/Database.html.

[193] Y. X. Niu and C. S. Hong and B. D. Chung, *Information Networking, Wired Communications and Management: ICOIN (Lecture Notes in Computer Science)*, ch. An adaptive TCP for enhancing the performance of TCP in mobile environments, pp. 516–526. Springer-Verlag, 2002.

[194] P. Goyal, H. M. Vin, C. Shen, and P. J. Shenoy, "A reliable, adaptive network protocol for video transport," in *Proc. IEEE Infocom*, pp. 1080–1090, Mar. 1996.

[195] N. Ramos, D. Panigrahi, and S. Dey, "ChaPLeT: channel-dependent packet level tuning for service differentiation in IEEE 802.11e," in *Proc. Intl. Symp. Wireless Personal Multimedia Communications*, pp. 86–90, Oct. 2003.

[196] T. Ue, S. Sampei, N. Morinaga, and K. Hamaguchi, "Symbol rate and modulation level-controlled adaptive modulation/TDMA/TDD system for high-bit-rate wireless data transmission," *IEEE Trans. Vehicular Technology*, vol. 47, pp. 1134–1147, Nov. 1998.