UvA-DARE (Digital Academic Repository)

Eclipse : flexible media processing in a heterogeneous multiprocessor template

Rutten, M.J.

Publication date

2007

Document Version

Final published version

[Link to publication](#)

Citation for published version (APA):

Rutten, M. J. (2007). *Eclipse : flexible media processing in a heterogeneous multiprocessor template*. [Thesis, externally prepared, Universiteit van Amsterdam]. Vossiuspers.

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

Eclipse

Flexible Media Processing in a Heterogeneous Multiprocessor Template

Martijn Rutten

UNIVERSITEIT VAN AMSTERDAM

Eclipse

**Flexible Media Processing in a
Heterogeneous Multiprocessor
Template**

Cover design: René Staelenberg, Amsterdam

Cover illustration: The sun's corona during the total eclipse of March 29, 2006.
Cover picture and the picture on page v by Robert Wielinga, taken on the Turkish coast.

ISBN 978 90 5629 484 7

NUR 980

© Martijn Rutten / Vossiuspers UvA — Amsterdam University Press, 2007

All rights reserved. Without limiting the rights under copyright reserved above, no part of this book may be reproduced, stored in or introduced into a retrieval system, or transmitted, in any form or by any means (electronic, mechanical, photocopying, recording or otherwise) without the written permission of both the copyright owner and the author of the book.

Eclipse

Flexible Media Processing in a Heterogeneous Multiprocessor Template

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van Rector Magnificus
prof. dr J.W. Zwemmer
ten overstaan van een door het College voor Promoties ingestelde
commissie, in het openbaar te verdedigen in de Aula der Universiteit
op donderdag 13 september 2007, te 10:00 uur

door

Martijn Johan Rutten

geboren te Deil

Promotiecommissie:

Promotor: prof. dr. L.O. Hertzberger
Co-promotor: dr. ir. J.T.J. van Eijndhoven

Overige leden: prof. drs. M. Boasson
prof. dr. H. Corporaal
prof. dr. C.R. Jesshope
dr. ir. C.T.A.M de Laat
dr. G.J. Muller
prof. dr. ir. P. de With

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The work described in this thesis has been carried out at the Philips Research Laboratories and NXP Semiconductors in Eindhoven, the Netherlands, as part of the Research program.

Eclipse was not named Eclipse at first. Evert-Jan Pol baptized the architecture as LIPS (Local Irregular Processing Structure), focusing on the irregular behavior of compression algorithms and the local interconnect with on-chip memory. This was soon extended to HOTLIPS (Heterogeneous On-chip Template for Local Irregular Processing Structures), to address the idea of designing an architecture template allowing different instances for different products (and because it sounded sexy). Soon thereafter, management obliged us to change name once more, as we had failed to realize the impact the indecent connotation the word 'hot lips' would have on American customers. Thus, the project was coined ECLIPS (Extended CPU Local Irregular Processing Structure), focusing on the possibility to mix hardware and software streaming at a fine granularity. By the time Pieter van der Wolf pointed out that the international community uses a slightly different spelling for a solar eclipse, the ECLIPSE (Extended CPU Local Irregular Processing StructurE) project was well underway.

Acknowledgment

With this thesis, I stand on the shoulders of two remarkable architects, Evert-Jan Pol and Jos van Eijndhoven. Ever so often, I would study a certain aspect of the architecture. Typically, when I proudly presented my solution after careful consideration of all options, Evert-Jan would politely point out to me that I had overlooked a detail that rendered my solution utterly useless. However, my ideas triggered him to change one detail and invent the final solution, letting me wonder why I couldn't have come up with that. Next, Jos van Eijndhoven would pass by as an interested onlooker and within minutes sketch a highly innovative implementation on the white board. Jos and Evert-Jan: a very special thanks for all the support, coaching, and friendship. Evert-Jan, I am extremely grateful for all the architecting experience you taught me over the many years we worked together, and particularly for helping me to reverse engineer the design decisions in the EDDI use cases after a few years.

Clearly, this thesis builds on the work of many other researchers: Egbert Jaspers and Erik van der Tol, showing me what MPEG-2 really means; Pieter van der Wolf—who was one of the first to stress that Eclipse had a value far beyond a TriMedia accelerator. Pieter, thanks for teaching me how to properly structure a document and for all the last-minute reviewing of Eclipse papers, typically a few hours before the submission deadline. Thanks to John Moors, struggling to decrypt my coprocessor shell design document during RTL implementation. Karel Walters, your enthusiasm and skill in developing the Eclipse control software was inspiring! I will never forget the enthusiasm of the Semiconductors and Research teams in India. Ferry, thanks for all the times you rescued me from spending all weekend typing on my thesis in Bangalore!

My gratitude goes out to Bob Hertzberger for supporting this thesis since 2002. Jacqueline, thank you for your help and support during these years! I am deeply indebted to my parents for virtually everything, with a special thanks to my father for reviewing and kayaking with me whenever it's too cold to stay home.

Last but not least, a huge thanks to Katrien, Open Office guru Andrei, Jaap, Maca, Nanni, Enith, Giuseppe, Loes, Wouter, Joep, Joost, Csaba, Thibaut, Alex, Vedran, Andreja, Mathias, Clara, and Jaques for always asking when I would finally finish my thesis and at the same time inviting me to do other fun things...

I will never have enough pages to thank you, Derya, for being so sweet despite all the time I spent behind my laptop and left you alone...tatlı karıcım, sevgin ve desteğin için çok teşekkürler!

July 09, 2007

Martijn Rutten

Contents

1	Introduction.....	1
1.1	Objective.....	2
1.2	Design goal.....	3
1.3	Solution approach.....	4
1.4	Thesis organization and contributions.....	5
1.4.1	Part I: Eclipse architecture.....	5
1.4.2	Part II: Eclipse simulation.....	7
1.4.3	Part III: Eclipse case studies.....	7
1.5	Contributions of the author.....	8
1.6	Discussion.....	9
	Part I Eclipse architecture.....	11
2	Multiprocessor architecture design.....	13
2.1	Performance and cost.....	13
2.1.1	Exploiting application parallelism.....	14
2.1.2	Parallelism in media processing	15
2.1.3	Synchronization of parallel media-processing tasks.....	16
2.2	Flexibility and time to market.....	18
2.2.1	Flexibility of computation.....	19
2.2.2	Flexibility of data routing and synchronization.....	19
2.3	Design reuse and development effort.....	20
2.4	Multiprocessor architecture templates.....	22
2.4.1	SoC architecture templates.....	22
2.4.2	SoC subsystem architecture templates.....	26
3	Eclipse architecture template.....	31
3.1	Eclipse architecture philosophy.....	32
3.1.1	Concurrency.....	33
3.1.2	Infrastructure.....	33
3.1.3	Avoiding deadlock.....	34

3.2	Separation of function and infrastructure.....	35
3.2.1	GetTask.....	36
3.2.2	GetSpace.....	36
3.2.3	PutSpace.....	37
3.2.4	Read.....	37
3.2.5	Write.....	38
3.2.6	Report.....	38
4	Eclipse coprocessor control.....	39
4.1	Data access synchronization.....	40
4.1.1	Random access.....	42
4.1.2	Multicast.....	43
4.1.3	In-place updates.....	44
4.2	Task switching.....	45
4.2.1	Busy wait.....	45
4.2.2	Discarding partial work.....	46
4.2.3	State save in internal coprocessor memory.....	47
4.2.4	State save through a single-access buffer.....	48
4.3	Dynamic reconfiguration.....	49
4.3.1	Synchronized reconfiguration.....	51
4.3.2	Reporting control events.....	53
4.3.3	Formatting data streams.....	53
4.3.4	Forwarding location packets.....	54
5	Eclipse generic infrastructure.....	57
5.1	Stream synchronization.....	58
5.1.1	Local stream administration.....	59
5.1.2	Synchronizing multicast streams.....	60
5.2	Data transport.....	61
5.2.1	Cache indexing through stream information.....	62
5.2.2	Cache coherency through explicit synchronization.....	63
5.2.3	Prefetching on dismissed cache locations.....	65
5.3	Task scheduling.....	67
5.3.1	Robust resource sharing through task budgets.....	68
5.3.2	Addressing dynamic workload through task runnability.....	70
5.4	Performance measurement support.....	72

5.5	Dynamic reconfiguration support.....	72
6	Eclipse coprocessor design.....	75
6.1	Design of reusable Eclipse coprocessors.....	76
6.1.1	Generic process network.....	77
6.1.2	Eclipse-tailored process network.....	77
6.1.3	Eclipse coprocessor design.....	78
6.2	MPEG application analysis.....	79
6.2.1	Generic decoder and encoder models.....	79
6.2.2	Eclipse-tailored decoder and encoder models.....	80
6.3	MPEG coprocessor design.....	81
6.3.1	Variable-length decoding and encoding (VLD).....	81
6.3.2	Run-length, scan, and quantization (RLSQ).....	82
6.3.3	Discrete cosine transform (DCT).....	83
6.3.4	Motion compensation/estimation (MC/ME).....	84
7	Discussion.....	85
	Part II Eclipse simulation.....	89
8	Multiprocessor architecture simulation.....	91
8.1	Application development.....	93
8.2	Architecture development and verification.....	94
8.2.1	Design verification.....	95
8.2.2	Design reuse.....	96
8.3	Architecture design space exploration and validation.....	98
8.3.1	Trace driven simulation.....	98
8.3.2	Execution-driven simulation.....	99
8.4	Multiprocessor architecture simulation environments.....	100
8.4.1	Architecture design-space exploration environments.....	100
8.4.2	Architecture modeling and verification environments.....	104
9	Eclipse simulation technology.....	109
9.1	Retargetable simulation.....	110
9.1.1	Distributed parameter parsing.....	111
9.1.2	Shared template parameters.....	112
9.1.3	Dynamic addition of coprocessor modules.....	114

9.2 Mixed-level simulation.....	116
9.2.1 Architecture modeling.....	116
9.2.2 Coprocessor modeling.....	117
9.2.3 Performance analysis and functional verification.....	119
9.2.4 RTL verification.....	121
10 Eclipse simulation experiments.....	123
10.1 MPEG-2 video decoding behavior.....	124
10.1.1 Decoding behavior for the Teeny sequence.....	125
10.1.2 Decoding behavior for the Tech sequence.....	131
10.2 Cache control.....	134
10.2.1 Cache behavior for producer-consumer.....	134
10.2.2 Cache behavior for single-stream MPEG decoding.....	140
10.2.3 Cache behavior with shared cache lines.....	144
10.2.4 Cache behavior for dual-stream producer-consumer.....	145
10.2.5 Cache behavior for dual-stream MPEG decoding.....	146
10.3 Dynamic reconfiguration.....	150
10.4 Budget-based task scheduling.....	152
11 Discussion.....	155
Part III Eclipse case studies.....	159
12 Video coding subsystem.....	161
12.1 MPEG coding subsystem.....	161
12.1.1 Memory bandwidth.....	162
12.1.2 Coprocessor performance.....	163
12.1.3 Silicon area.....	164
12.1.4 Power consumption.....	164
12.2 Conclusion.....	165
13 Image processing subsystem.....	167
13.1 Edge dependent de-interlacing.....	167
13.2 Coprocessor design.....	168
13.2.1 Input to filter and de-interlacer Y-data streams.....	169
13.2.2 Input to de-interlacer UV-data stream.....	170
13.2.3 Stream of filtered pixels and gradients.....	171

13.2.4 Stream of detected edges.....	171
13.2.5 De-interlacer to output stream.....	172
13.3 EDDI subsystem.....	172
13.3.1 Buffer size.....	173
13.3.2 Memory bandwidth.....	173
13.3.3 Coprocessor performance.....	174
13.3.4 Silicon area.....	175
13.4 Conclusion.....	177
14 Audio accelerator subsystem.....	179
14.1 Processing tiles.....	179
14.1.1 DSP tile.....	180
14.1.2 Accelerator tile.....	181
14.2 Coprocessor design.....	183
14.3 Conclusion.....	184
15 Summary.....	185
A In-place update synchronization.....	189
Bibliography.....	191
Samenvatting.....	199

1 Introduction

New media applications such as high-definition digital television, set-top boxes with time-shift functionality, 3D games, video conferencing, and MPEG-4 interactivity have generated a demand for increasingly flexible consumer electronics products. These products are evolving into multi-functional devices that combine some subset of such media applications. The required set of applications and their format vary per product, per country, and over time as standards evolve. Managing the complexity, design cost, and time to market of these programmable, resource-constrained products requires a generic, scalable media-processing platform that can be deployed in a wide range of such products. With the advance in silicon technology, the development cost of a new chip will eventually outgrow the final silicon cost. At present, consumer electronics vendors show a high interest in full software platforms that—after a large initial development effort—can be reused over a wide range of products [14][135]. In this respect, we will see a crossover point where the rapid turn around time of software solutions outweighs the benefit of reduced silicon cost of dedicated hardware. However, for today’s media-processing products, the silicon cost contributes significantly to the total product cost. Therefore, present software-based solutions are only affordable in the high-end and niche markets, where time to market and the set of product features determine the success of the product and silicon cost or power consumption are not a key concern.

Figure 1. Evolution of products over the product life cycle from first market introduction to mature high-volume products.

For at least the coming decade, there will be a large need for flexible yet highly cost-effective media-processing platforms in the medium to high volume markets. Several vendors are entering the market with system-on-chip (SoC) platforms that

address this need to some extent [2][1][28][104]. NXP Semiconductors has developed a platform concept embodied in the Nexperia line of chips [104]. Such SoCs typically consist of a heterogeneous mix of fully programmable processors (such as MIPS, ARM, and TriMedia processors) and coarse-grained application-specific subsystems (such as MPEG decoders and video filters) optimized for high performance with minimal power consumption and silicon area.

As depicted in Figure 1, deployment of flexible SoC platforms enables a smooth transition from expensive software-based products—which allow early market introduction with a limited production volume—to competitive mixed hardware/software solutions once a firm market position has been established.¹ With the exploding mask costs for upcoming silicon technologies [25] and the decreasing product life cycle of new media applications, the end of the curve in Figure 1 will slowly shift from mostly dedicated hardware solutions to more widely applicable heterogeneous solutions with larger software content. With the tremendous production volumes of consumer appliances, the silicon cost will remain a dominant factor in the total product cost for a long time. Therefore, we expect that a shift towards larger software content in high-volume products will occur at a relatively slow pace.

1.1 Objective

SoC subsystems incur a large part of the total design effort and silicon cost. Currently, hardwired SoC subsystems are dedicated to a single function, and the hardware cannot be reused for other applications within the same domain. Although the generic platform with its interconnect structure can be reused over various generations, a change in application requirements necessitates a redesign of the application-specific subsystems. Clearly, this incurs a large development effort and cost, and thereby increases the time to market of a new SoC-based product.

The particular architecture called *Eclipse* and described in this thesis aims to strengthen the deployment of SoC platforms by creating a higher level of flexibility within the SoC platform. Eclipse defines an architecture template that enables rapid development of SoC subsystems by instantiating the template. The Eclipse template can be instantiated within a wide range of possibilities of coprocessor combinations, memory sizes, etc. The wide applicability of the template reduces the cost of ownership compared to deploying a set of architecture templates, each addressing only a small range of applications. Once it is instantiated on a SoC, an Eclipse module enhances the scope and range of applications that can be imple-

¹ Nevertheless, the author has hitherto not witnessed a strong industrial track record for such a transition process. Current software focused platforms either do not provide the required performance (e.g. high-definition H.264 video decoding), or the higher cost of a software solution is negligible in the total bill of material, e.g. in audio post-processing.

mented efficiently. Hence, it permits SoCs to be applied in a wider range of products.

An example is the case of a time-shift recorder, which combines standard-definition MPEG-2 decoding and encoding functionality. In the Viper architecture [32] of NXP's Nexperia-Home platform [104], the MPEG-2 decoding function is mapped on a dedicated MPEG-2 coprocessor and the encoding is performed in software by the TriMedia processor [115]. Cost reduction along the curve of Figure 1 can be obtained by mapping both the encoding and decoding functions onto an MPEG-2 instantiation of the Eclipse template. Eclipse hardware then performs most of the MPEG-2 processing while a small control processor handles the configuration of the Eclipse network and realizes the remaining MPEG-2 functionality in software (e.g. parsing picture properties).

1.2 Design goal

The development of the Eclipse template should culminate in a design method for the development of next-generation SoC subsystems for media processing. The Eclipse architecture template should enable a system designer to fulfill several important requirements. Clearly, Eclipse SoC subsystems must support the *performance* requirements of targeted media-processing applications. These subsystems must be *cost-effective* to be applicable in the medium to high-volume product segment. However, Eclipse subsystems must provide sufficient *flexibility* to allow deployment in a wide range of products. In order to accommodate the conflicting requirements between cost and flexibility, the Eclipse template must provide means to optimize implementations for specific cost and flexibility targets.

Eclipse should form a true architecture template in the sense that it is *scalable* with respect to the application requirements and characteristics of the targeted silicon technology, as well as the use of third party components such as SRAM memories. Moreover, the definition of such a template must minimize *development cost* and time to market of Eclipse-based products, for instance by supporting reuse of hardware and software over different Eclipse subsystems.

Finally, the Eclipse architecture template must minimize *development effort* for developing Eclipse subsystems by enabling modular design, as well as for developing SoCs by facilitating smooth integration of Eclipse subsystems into current SoC platforms, for instance the PrimeXsys platform [2]. Easy SoC integration means that both the hardware and software interfaces of an Eclipse subsystem must match the interfaces of the SoC platform towards its subsystems. Moreover, not only the interfaces but also the programming model—and consequently a large part of the architecture philosophy—must match to facilitate the deployment of Eclipse subsystems in current SoC platforms.

1.3 Solution approach

Our aim in the development of Eclipse was to provide a cost-effective and flexible architecture template by exploiting reusable, medium-size computation kernels in a mix of dedicated hardware units (termed *coprocessors* in this thesis) and software tasks executing on a media processor.

The architecture philosophy of Eclipse is based on two principles:

- a rigorous *separation between generic infrastructure and function-specific hardware* (coprocessors) to minimize the effort of designing application-specific coprocessors,
- *Highly autonomous coprocessors* to minimize the impact of adding a new coprocessor and to ensure efficiency and scalability of the Eclipse template.

Throughout Eclipse design, we emphasized on separating the responsibilities of different architectural elements through clean and stable interfaces. Six interface primitives served to separate application-specific functionality from generic infrastructure. The separation facilitates coprocessor design by moving most of the generic functionality to infrastructure modules that only need to be designed and verified once.

In the architecture of the Eclipse infrastructure, we avoided complex centralized modules that control the execution of the subsystem to allow for maximally autonomous coprocessors. This simplifies the effort of adding a new coprocessor to an existing subsystem by avoiding changes to existing, already verified, modules. Moreover, keeping control local to a coprocessor both increases efficiency of handling local requests from the coprocessor and increases scalability of the template by avoiding centralized bottlenecks, e.g. for handling the synchronization of many data streams.

Although we focused on concepts and generic solutions throughout the complete Eclipse project, we quickly implemented a MPEG-2 decoding and encoding application as driver. Since the project had an extensive history in MPEG-2 processing, this first application allowed to focus the development on the generic Eclipse infrastructure. The development of an MPEG-2 Eclipse subsystem enabled an application-architecture codesign approach, which enhanced our insight in the detailed architectural requirements from the application domain.

We developed a highly parameterized simulation model of the Eclipse architecture to obtain feedback on architectural choices early in the design trajectory beyond simple back-of-envelope calculations and specific technology studies. This simulation model allowed a mix of abstract models for the coprocessors with cycle-accurate models for the Eclipse infrastructure. Using abstract coprocessor models permitted focusing on the design of the infrastructure that is the heart of the Eclipse architecture.

1.4 Thesis organization and contributions

This thesis is split into three parts. Part I describes the design of the Eclipse architecture. Part II gives insight in our simulation-based design methodology. Part III concludes this thesis with case studies of Eclipse subsystems for video coding, image processing, and audio acceleration.

1.4.1 Part I: Eclipse architecture

Part I starts with a conceptual description of the aspects involved in designing heterogeneous multiprocessor architecture templates in Chapter 2. In addition, the latter gives an overview of related multiprocessor architectures. It so sets the stage for the introduction of the Eclipse template in Chapter 3.

Chapter 3 introduces the Eclipse architecture philosophy with the basic principles of maximally autonomous coprocessors and the separation between the generic Eclipse infrastructure and function-specific coprocessors. The chapter subsequently defines the corresponding interfaces.

Chapter 4 describes the full potential of this interface by showing how the six interface primitives can be exploited in Eclipse coprocessors. The contributions of of this thesis to the definition of coprocessors is a refinement of the Kahn model in the Eclipse interface to support the following.

- Streaming data via raw data streams with different granularity of production and consumption of data (Section 4.1). Existing streaming interfaces for embedded multimedia applications enforce a fixed communication granularity between a producer and consumer, typically by communicating references to predetermined data structures. Eclipse coprocessors operate on streams of bytes to increase the range of applications in which coprocessors can be deployed, and to avoid dependencies on data width and alignment restrictions of the communication network.
- Additional features to streaming via raw data streams that were thus far restricted to communication of references to separately allocated buffer elements. The interface allows for random access within a constrained window of available data or room in a stream (Section 4.1.1), multicast where multiple consumers share a single stream buffer to minimize memory consumption (Section 4.1.2), and in-place processing to avoid the overhead of copying large amounts of data when a coprocessor only needs to patch/update a few data items in a stream and leave the rest unmodified ((Section 4.1.3).

The generic implementation of a response to the interface calls from the coprocessors is given in Chapter 5. This chapter gives the main architectural concepts and choices underlying the Eclipse infrastructure, with a strong focus on the design of the ‘coprocessor shell’. Although Eclipse defines a heterogeneous solution mixing hardware and software, this thesis does not detail the connection to a DSP

or media processor, nor does it describe the execution of software media-processing tasks. However, Chapter 5 does introduce the control software and interfaces needed to dynamically configure Eclipse to execute a given set of targeted applications. The contributions of this thesis to the design of SoC subsystems are the following.

- The introduction of a generic coprocessor shell that simplifies coprocessor design by providing support for data access and synchronization, task scheduling, and application configuration through explicit interface primitives. State of the art ‘shells’ or ‘wrappers’ only focus on communication and do not include task scheduling and application configuration support.
- Distributed synchronization of data access, organized to quickly serve local requests (Section 5.1). State of the art synchronization techniques have a single, centralized administration of the buffer status of each data stream.
- Micro caches in the shell to isolate the coprocessors from the properties of the communication network, organized to support automatic cache coherency and prefetching as side effect of explicit synchronization actions (Section 5.2). The proposed mechanisms for cache coherency and prefetching are significantly more efficient in terms of complexity and performance than state of the art techniques that are based on analyzing data accesses.
- A distributed task scheduling algorithm that includes buffer information to maximize the possibility that scheduled data-dependent tasks can make actual progress (Section 5.3). Moreover, the algorithm includes properties to isolate the real-time behavior of data-dependent applications that share coprocessor resources. State of the art subsystems rely on global, typically static schedules which require a thorough analysis of all possible combinations of use cases.
- A mechanism to dynamically change the application configuration at unique points in the data processing that executes on multiple, parallel coprocessors (Section 5.5) The mechanism implemented in the infrastructure has no built-in knowledge of the application, format of the data streams, or the requested reconfiguration actions. State of the art techniques for such distributed reconfiguration rely on an imposed formatting of the data streams to convey reconfiguration information directly in the data streams.

The definition of the Eclipse infrastructure and interface to the coprocessors is completed with a design trajectory for creating Eclipse coprocessors. Chapter 6 exemplifies this design trajectory with a case study defining Eclipse coprocessors for MPEG encoding and decoding. The contributions of this chapter are the following.

- A methodology to design reusable coprocessors from multiple, related Kahn-style applications (Section 6.1). State of the art methodologies focus on the transformations of a single application, which does not result in the definition of coprocessors that can be reused across a range of applications.

- The architecture definition of dedicated, multi-tasking coprocessors that support both video encoding and decoding functions (Section 6.2). State of the art coprocessors support either encoding or decoding of a single stream, or resort to more programmable architectures to provide such flexibility at a higher cost.

Chapter 7 completes Part I with discussion.

1.4.2 Part II: Eclipse simulation

Part II, structured similar to Part I, starts with the methodology for designing domain-specific architectures in Chapter 8. This chapter gives a brief overview of multiprocessor architecture simulation techniques, and sets the stage for the introduction of the Eclipse simulation technology in Chapter 9. This chapter outlines our approach to developing modular and parameterized simulation models by gradually refining an abstract model into a cycle-accurate model of the resulting hardware. The contributions of this thesis to the design of SystemC models are:

- design patterns for the dynamic loading of modules and sharing design parameters among modules to initialize highly parameterized simulation models.
- the combination of abstract and cycle-accurate models within a single code base to reduce maintenance overhead.

Chapter 10 completes Part II with a selection of simulation experiments to validate key aspects (cache coherency and prefetching, budget-based task scheduling, and dynamic reconfiguration) of the Eclipse template. The simulations target the infrastructure, not the coprocessors, as the Eclipse template centers around a generic infrastructure. The infrastructure is subjected to a multi-stream MPEG decoder. The data-dependent behavior of an MPEG decoder together with the complexity of handling multiple MPEG streams make this an test bench to challenge the Eclipse infrastructure. Since the experiments are based on simulation, obviously this chapter does not provide a formal proof of correctness. Instead, the experiments give a first indication of correct behavior as an initial proof of concept. In addition, they give insight in the dynamic behavior of the Eclipse subsystem for MPEG decoding as outlined in Chapters 6 and 12. Part II is completed with a discussion (Chapter 11).

1.4.3 Part III: Eclipse case studies

Part III presents three case studies of potential Eclipse applications in the semi-conductors business context. The case studies validate the applicability of the Eclipse template for a wide range of applications. Each case study is concluded with a brief discussion. Chapter 12 outlines an Eclipse subsystem for multi-stream video decoding in digital-television SoCs, which initially triggered the Eclipse concept. This video subsystem is a good benchmark for a demanding

high-performance application. Next, Chapter 13 presents an actual business case for an Eclipse subsystem to handle edge-dependent de-interlacing in storage devices. The business case forms a good benchmark of a low-cost Eclipse subsystem.

Finally, Chapter 14 treats a concept Eclipse subsystem to accelerate audio processing for car infotainment products. This chapter shows the feasibility of cooperation between an Eclipse subsystem and a programmable DSP. Together, the high-performance video subsystem of Chapter 12, the low-cost image subsystem of Chapter 13, and the audio subsystem of Chapter 14 cover a wide range of applications that can benefit from the Eclipse concept. Chapter 15 concludes this thesis with a summary.

1.5 Contributions of the author

Eclipse was designed by a team of architects and engineers at Philips Research and Philips Semiconductors in a joint effort, starting in 1999. This thesis does not address all of the activities of the Eclipse project, but focuses on the areas where the author was actively involved.

The author was the main contributor in the definition of the coprocessor shell that largely defines the Eclipse infrastructure, including the concepts and algorithms for task scheduling, cache coherency and prefetching, and dynamic reconfiguration, as detailed in Chapter 5. The author had only minor contributions to the definition of the DSP shell and the implementation of the communication and synchronization network.

With regard to the definition of coprocessors, the author was the main contributor to the support of random access, multicast, and in-place processing in the Eclipse interface, as detailed in Chapter 4. The author also contributed the methodology to gradually refine application task graphs into reusable coprocessors (Section 6.1). However, he author was only minimally involved in the actual design of coprocessors for video compression (Sections 6.2 and 6.3).

The author contributed the design patterns for initializing parameterized SystemC models, and took the lead in the modeling of the Eclipse infrastructure. The author's involvement in the modeling of coprocessors was minimal. However, all experiments of Chapter 10 were defined and implemented by the author.

The author did not take the lead in the definition of the subsystems for video compression and image compression of Chapters 12-15, but has lead the design of the control software to (re)configure the Sea-of-DSP architecture in Chapter 14.

1.6 Discussion

The aim of the Eclipse design (Section 1.2) is to allow designers to quickly create subsystems that approach the efficiency of dedicated hardware yet have the flexibility to execute a range of (related) applications. The two design principles (Section 1.3) support this aim. The first principle introduced the separation between function and infrastructure. Here, the introduction of the coprocessor shell that offers generic services to the coprocessor via a well-defined interface simplifies coprocessor design and provides opportunities for building up a library of reusable coprocessors to further reduce the effort involved in constructing a new subsystem. The case studies of Part III show that the interface provides a rich set of possibilities to create efficient coprocessors with minimal dependencies towards the rest of the system. These studies also show that the interface can be efficiently implemented in the generic coprocessor shell.

The second principle focused on creating maximally autonomous coprocessors. This is reflected in the distributed implementation of synchronization and task scheduling in the shell. The video compression subsystem described in Chapters 6 and 12 was designed to verify the distributed approach on a known application. The simulation experiments of this subsystem in Chapter 10 show that the distributed synchronization and scheduling gives functionally correct results with a well-balanced schedule. The experiments also demonstrate that the shells can efficiently cache data accesses and quickly respond to synchronization requests with only local information on buffer filling. Moreover, the experiments show that the shells allow for application-level control of (soft) real-time behavior and dynamic reconfiguration without any knowledge of the function implemented in the coprocessors or the data that is communicated in the data streams.

The results of the research project have been transferred to Philips Semiconductors, where an initial RTL implementation of the coprocessor shell has been created. The Sea-of-DSP streaming infrastructure (Chapter 15), deployed in a car radio product, are based on the principles of the Eclipse interface for streaming via raw data streams and application configuration. Despite the positive technical evaluation of the subsystems defined in Part III, Eclipse has not been implemented in a product. As detailed in the discussion sections of Part III, the reasons for this were largely organizational. We observe that long term vision and strong cooperation across organizational boundaries are prerequisites to benefit from an architecture template such as Eclipse that allows reuse of hardware within a single system among different functions or even across a range of products.

Part I

Eclipse architecture

Total solar eclipse, July 11, 1991. Composite photograph by Steve Albers, Dennis Di-Cicco, and Gary Emerson.

The solar corona is a billion times less bright than the naked sun, and a thousand times weaker than our sun-lit sky. Only when the sun itself is covered by the moon during a rare and magnificent total eclipse may one observe the pearly coronal glow. Its white color betrays its origin: ordinary sunlight, scattered in the corona towards our eye. The scattering is caused by loose electrons, freed through ionization in the high temperature (1-2 million degrees) coronal plasma. The intrinsic coronal radiation is at X-ray wavelengths, observable only from spacecraft outside our atmosphere.

This composite of multiple exposures closely matches the visual impression gained by eclipse watchers during totality—the author being one to enjoy this spectacular vista on a Pacific beach near Salina Cruz in Mexico (in contrast to the locals who stayed inside to watch the phenomenon on television, being misinformed by the Mexican government).

2 Multiprocessor architecture design

In the consumer-electronics domain, media-processing applications execute on resource-constrained systems. The performance, flexibility, and cost of such systems are highly interrelated. Their design exhibits three major trends. First, current consumer-electronics chips are heterogeneous multiprocessor systems in order to achieve the required performance while minimizing power consumption and silicon area. Secondly, these systems become increasingly flexible to reduce time-to-market and development cost so that they can be deployed quickly in a wide range of products. Finally, since the amount of functionality that can be contained within a single chip increases rapidly with each technology step, the development effort to design, implement, and test this functionality grows exponentially. These trends are addressed through the deployment of generic SoC platforms and architecture templates. This chapter outlines the main architectural tradeoffs that influence these aspects and underlie the Eclipse architecture's design choices.

2.1 Performance and cost

In consumer electronics, design constraints such as cost, power consumption, performance, and flexibility are weighted differently than in the PC market. Consumer media processors require an order-of-magnitude lower cost combined with a significant demand for performance. The key challenge is to design a flexible media-processing architecture that combines high performance with low power consumption and silicon cost.

The traditional approach to increase a processor's clock frequency to reach desired performance becomes increasingly unattractive as the memory subsystem cannot keep up with the processor performance, and power dissipation becomes critical. The cost of current general-purpose processors grows exponentially as designers are forced to resort to complex solutions to squeeze the maximum performance out of a given design—for instance by increasing pipeline depth, or introducing branch prediction logic and second-level caches. Even as minimum feature sizes continue to shrink in accordance with Moore's law, the gap between hardware cost and performance is likely to increase. In the evolution of uniprocessor systems, next-generation processors that double the area of their predecessor typically do not increase the actual system performance beyond 50 percent.

For instance, the Intel Itanium processor doubled the gate count of its predecessor while system-level performance did not increase beyond 15 percent [55][112].

As the maximum operating frequency is proportional to the supply voltage, power dissipation scales to the third power with increasing clock frequency [145]. In contrast, power dissipation increases only linearly with added parallelism. Therefore, media-processing architectures aim to maximally exploit application parallelism, while only increasing clock frequency with advancing silicon technology.

A large part of the hardware complexity of current general-purpose processors is spent on extracting instruction-level parallelism from a sequential program. In the consumer electronics domain, one has the advantage that parallelism in the targeted applications can be obtained at system level before the applications are mapped onto the hardware. This allows system architectures that exploit parallelism at system level, enabling increase in performance that is in principle proportional to the amount of silicon area and power consumption. However, the actual performance gain depends on the degree of parallelism that the system can exploit in the targeted applications, as well as on the overhead incurred by moving data and synchronizing data access between parallel processes.

2.1.1 Exploiting application parallelism

Media-processing applications typically exhibit parallelism at various levels of granularity. For example, a time-shift recorder may execute in parallel with other applications in the system. Internally, the time-shift recorder consists of coarse-grain encoding and decoding functions that may execute in parallel. Moreover, within the decoding function, many medium-grained tasks can execute in parallel—for example, the discrete-cosine transform (DCT) and quantization tasks. Figure 2 illustrates these different levels of task-level parallelism. Within the DCT task, many operations can execute in parallel, such as the horizontal and vertical one-dimensional DCT.

Figure 2 depicts a *functional partitioning* of the application, in which the application is split into parallel tasks for each logical application function, such as the forward and inverse DCT operations. Functional partitioning is mostly applied to extract application tasks that can be mapped onto domain-specific hardwired processors. Alternatively, video functions generally enable a coarse-grain *data partitioning*, where each decoding process contributes to a part of the composed picture. For instance, the MPEG decoder of Figure 2 can also be partitioned into independent parallel decoders for each MPEG slice [12][33]. Such data partitioning generally applies to homogeneous multiprocessor architectures [135][141]. Audio functions typically do not allow coarse-grain data partitioning and are more suited for functional partitioning.

Figure 2. Functional partitioning of a time-shift recorder to allow task-level parallelism at three different levels of granularity. Level 1 coarse-grain tasks are the MPEG-2 encoder and decoder. Level 2 medium-grain tasks are elements such as the discrete cosine transform (DCT) and its inverse (IDCT), and motion compensation (MC). Level 3 fine-grain tasks are the one-dimensional horizontal and vertical DCTs.

The application designer must choose how to partition the application to maximally exploit the parallelism offered by the architecture [22]. In practice, the application designer may explore a combination of functional and data partitioning at different levels of granularity [33][141]. Clearly, functional and/or data partitioning is required to make the inherently available task-level parallelism explicit. *Eclipse* exploits medium-grain task-level parallelism at the subsystem level, and supports data- and instruction-level parallelism inside its coprocessors.

2.1.2 Parallelism in media processing

An application designer can make task-level parallelism explicit by specifying a media-processing application as a Kahn process network (KPN)—a set of concurrently executing tasks that exchange information solely through unidirectional data streams [80]. A directed graph with a node for each task and an edge for each data stream represents the applications structure. Figure 3 shows such a graph for MPEG-2 video decoding [74]. See Figure 30 for a more detailed MPEG-2 decoder model. *The Kahn model reflects the Eclipse architecture's model of computation.*

The data streams in Kahn networks are buffered. Each buffer is an unbounded FIFO buffer, with one producer and one or more consumers [81]. Reading from a stream with insufficient data causes a consuming task to stall. Kahn formally proved that such a system has a well-defined, unique behavior. In particular, the functional behavior—observed as the sequence of data items that traverse each edge—is independent of the order in which the tasks are executed. Consequently, a Kahn process network is not essentially different from a task and can thus be used as a task in a larger process network.

Figure 3. MPEG-2 video decoder partitioned into medium grain tasks.

This inherent building-block nature is an advantageous feature of Kahn's model. After defining a set of basic functions as tasks, the designer can configure a multitude of applications by creating tasks and connecting them in a graph structure. Describing an application in these generic tasks is followed by a mapping phase in which the designer decides which architecture modules execute which tasks. Usually, not the entire application can be modeled with generic tasks. *In Eclipse, we aim to define the hardware modules such that we map similar Kahn tasks to a single hardware module.* For instance, we may define a generic DCT module that executes both the inverse and forward DCT tasks of Figure 2. For this mapping to make sense, the Kahn task granularity must match the module granularity (Chapter 6).

2.1.3 Synchronization of parallel media-processing tasks

Bounded process networks are a particularly interesting class of Kahn process networks whose execution continues indefinitely within bounded memory—provided they are fed sufficient input. Many practical media- and signal-processing applications fall into this class. The architecture must execute such a process network using bounded memory, preferably not more than necessary to meet the given performance requirements.

Eclipse restricts the size of the communication buffers, combined with an essentially data-driven execution [8][85]. A key drawback of this approach is that it is difficult to determine appropriate buffer sizes. Bounded buffers may cause artificial deadlock, which cannot occur in a Kahn process network with unbounded buffers. *Eclipse coprocessors switch to another task when the active task has insufficient data/room in its input/output buffers to proceed. With the possibility to dynamically increase buffer sizes in control software and a deadlock-free communication network, the Eclipse architecture is free from artificial deadlock.* For a more detailed discussion on deadlock in Eclipse, see Section 3.1. Alternative

approaches avoid artificial deadlock by restricting the flexibility of the Kahn model [87]. With one-place buffers, the Kahn model reflects the Codesign Finite State Machines (CFSM) model [5]. When the buffer sizes are reduced to zero, the Kahn model mimics the Communicating Sequential Processes (CSP) model of computation in which tasks communicate via rendezvous [68].

The size of a communication buffer determines the extent to which the producer and consumer are coupled in execution timing. The factors that determine the choice of buffer size to reach a certain degree of decoupling are the application's processing regularity and the admissible amount of stalling behavior sacrificing parallelism. For instance, regular tasks allow tight coupling with minimal buffering, such as linear video filtering tasks where the worst-case communication requirements equal the average case. Irregular tasks demand looser coupling to allow individual task progress, leading to larger buffer requirements. A typical irregular, data-dependent task is variable-length decoding in MPEG decoding, in which the quantity of input and output data can vary wildly from stream to stream or even within a single stream. A less obvious example is the DCT function: the function itself is regular but the number of DCT-coded blocks to be processed varies per MPEG frame. *A key Eclipse target, video-encoding and -decoding, is highly data-dependent and thus highly irregular.* In practice, the ratio of worst-case versus average load can easily reach a factor of 10. Consequently, we designed Eclipse as a relatively loosely coupled system.

Communication requires both data transport and synchronization. *Data transport* is the movement of data into and out of communication buffers. *Synchronization* is the exchange of information between producers and consumers about the amount of produced or consumed data in the buffer. Thanks to the buffering of communicated data, the producer and consumer need not mutually synchronize individual read and write actions on the stream. Thus, the designer can choose synchronization granularity independent from data I/O. For instance, the designer can choose to synchronize each individual data access. Alternatively, synchronization can be performed at some logical unit of input and output data on which a task operates—for example, the granularity of a picture in an MPEG-decoding task.

Decreasing the function granularity (for example, from an MPEG-decoder function to a DCT function) enhances parallelism and opportunities for reuse. However, as the number of streams passing through the communication network increases, more communication is made visible in the communication network. *Eclipse copes with the demanding communication requirements of medium-grain functions by synchronizing at a finer granularity, for example at the level of MPEG macroblocks instead of synchronizing at picture level.* The concomitant small communication buffers can reside on chip, supported by a dedicated communication network able to cope with the high bandwidth requirements.

2.2 Flexibility and time to market

The international roadmap for semiconductors (ITRS) [73] predicts that while manufacturing complex SoCs will be practical—at least down to 50 nm minimum feature sizes—the cost of masks becomes a major bottleneck in the development of such chips. With the progress in IC technology and the corresponding increase in mask costs, the software content in SoCs for high-volume products will grow to the extent where they become highly flexible to be deployed in a wide range of products, thereby amortizing the mask costs over an increasingly large number of products. However, for the coming decade(s) the cost of masks will not outweigh the benefits of developing new SoCs while progressing along the curve of Figure 1.

Figure 4. Application universe with application domains [91]. This example shows three application domains: media processing, video processing, and control processing. Each domain contains applications with similar characteristics.

The development costs associated with the design of complex SoCs are growing rapidly, while the life span from their first market introduction in a consumer product until they are replaced by a new version decreases rapidly. To reduce development effort and thereby time-to-market, media-processing SoCs must be designed to be deployed in a range of products. Thereto, these systems must be sufficiently flexible to support a variety of applications, as well as the future evolution of a given application. A range of applications with similar characteristics is referred to as an *application domain* (Figure 4). Typically, an SoC will encompass applications from various domains.

Today's flexible systems typically accommodate one or more programmable processors, and may allow programmable connections between the processors. Throughout this thesis, we apply a wide definition of the term *programmable processor*, including hardwired processors for which the function can be changed in software. Examples of programmable processors are CPUs, DSPs, and also FPGA-based reconfigurable processors. The extent of programmability offered by the processors and the interconnection infrastructure defines the flexibility of the system to target different applications within a chosen application domain.

However, a flexible hardware architecture is not enough to provide application flexibility. The underlying programming model and supported tools—such as compilers and performance analysis tools—define to a large extent how much of the architecture flexibility can and will be exploited by the application programmer. Important design considerations are a well-defined and consistently implemented programming model that matches the characteristics of the application domain, as well as the level of abstraction offered by the application programmer's interface (API). The API must match the programmer's view on the architecture and its applications without exposing the programmer to unnecessary details of the architecture. Yet, the API must allow the programmer to control 'emerging properties' of the system such as robustness and performance.

2.2.1 Flexibility of computation

Clearly, flexibility of processing has its cost. For instance, DCT computation for real-time high-definition MPEG decoding will take up a large part of a fully programmable TriMedia TM3260 processor. A dedicated DCT processor is an order of magnitude 10 smaller in silicon area and offers the same performance at a much lower hardware cost, not to mention the reduction in power consumption.

To reconcile efficiency and flexibility, SoC platforms are designed as a heterogeneous mix of hardware and software. For example, complex media-processing SoCs exploit the performance density of sophisticated application-specific subsystems to implement critical parts of the targeted media applications. At the same time, these SoCs give the system flexibility by embedding programmable processors. For competitive media-processing architectures in consumer products, designers maximize performance by exploiting parallelism wherever possible, and they minimize cost by introducing flexibility only where necessary. *Eclipse exploits medium-grain coprocessors in a building-block application approach, while supporting computational flexibility through the inclusion of a programmable media processor or DSP.*

2.2.2 Flexibility of data routing and synchronization

In principle, the partitioning of an application into a structure of interconnected tasks allows a designer to compose new applications as different combinations of these tasks. We refer to such programmability at the level of interactions between tasks as *application configuration*. A media-processing system where the granularity of the hardware modules matches the granularity of the application tasks is highly flexible if this match is complemented by a generic infrastructure that allows run-time instantiation of task graphs. The SoC subsystems implement the tasks, and the SoC infrastructure handles data communication between subsys-

tems. The infrastructure routing flexibility plays an important role in the SoC's application configuration flexibility.

Despite the flexibility of linking SoC subsystems into various application graphs, the coarse granularity of hardwired SoC subsystems renders these subsystems internally inflexible. Compared with the medium-grain tasks in Figure 3, larger functions are necessarily more specialized. For instance, an MPEG-2 decoder block cannot be reused for an MPEG-2 encoding application, even though an MPEG-2 encoder contains a large portion of an MPEG-2 decoder. To allow such subsystem configuration flexibility, the subsystem must support the mapping of smaller-grained application tasks on internal function modules of corresponding granularity. *Eclipse introduces a programmable infrastructure that supports dynamic application configuration linking medium-grain tasks into application graphs.*

The introduction of data routing and synchronization flexibility at a medium grain should not overload the SoC data transport and synchronization capacity. SoC platforms often cluster related processing elements into subsystems that hide the communication and synchronization requirements between their processing elements from the SoC infrastructure. The subsystems of such hierarchic SoCs typically encompass fixed local interconnect structures and dedicated local on-chip memory to manage their internal data transport requirements. Moreover, some subsystems may contain a dedicated control processor to handle the high internal synchronization rates.

2.3 Design reuse and development effort

As the complexity of both hardware and embedded software continues to grow, reuse is a must to reduce development effort. Reuse is promoted by applying architecture templates throughout the design. An *architecture template* is a generic architecture blueprint from which designers can create different hardware instances by fixing design parameters such as number and type of coprocessors, bus widths, and memory sizes. The template fixes responsibilities of architectural elements and defines generic functionality and interfaces. By instantiating such a template, the designer tunes the architecture towards a required application or set of applications with required performance and hardware cost. In doing so, the designer both reduces design effort, since the template fixes many potential design decisions in advance, and reuses generic functionality—for instance for routing data and synchronizing data access. *Eclipse provides an architecture template at the SoC subsystem level.*

The development of an architecture template heavily relies on the separation of responsibilities of architectural elements, as outlined in Table 1. Keutzer et al. [83] stress the importance of separating architecture and communication. We extend their approach to a separation between function and infrastructure architec-

ture. *Infrastructure* is defined as the architectural elements or services needed to execute a desired function. Infrastructure encompasses communication architecture, but also contains task-scheduling support for example. Architecture templates enable reuse of generic infrastructure, whereas instances of the template may allow reuse of generic function units, such as a DCT coprocessor.

Table 1. Examples of function and infrastructure architecture.

	Function	Infrastructure
SoC	MPEG decoder	SoC memory bus
SoC subsystem	DCT coprocessor	FIFO buffer
		Operating system

The definition of function and infrastructure depends on the granularity that is considered. For instance, an SoC subsystem for MPEG coding may be regarded as function module at SoC level. From the subsystem perspective however, the subsystem may contain a generic infrastructure to interconnect smaller-grain MPEG functions (coprocessors). *The Eclipse template provides generic services to application-domain specific coprocessors through a well-defined set of interface primitives.*

Figure 5. Generic SoC platform with application-domain specific subsystems. This example shows two subsystems connected to the SoC infrastructure. The top-level SoC infrastructure connects to the CPU/DSP and coprocessor, and sees each subsystem as a function module. Internally, each subsystem resembles the overall layout at a smaller granularity, where the coprocessors and CPU/DSP supply the functionality within the subsystem.

A second aspect of minimizing development effort through reuse is the deployment of maximally independent SoC subsystems with clearly defined and stan-

standardized interfaces to the outside world. SoC integrators cope with the exploding complexity of SoC integration by instantiating such modules in a plug & play like fashion as illustrated in Figure 5.

While such subsystems may internally consist of a heterogeneous mix of hardware and software and contain embedded cores, embedded memory, and communication networks tuned to the application domain, the SoC integrator should be able to manage complexity by regarding the module as a black box, a SoC-level accelerator that is highly tuned to a targeted application domain. Thus, the subsystem must be designed to shield the internal complexity from the SoC integrator, for instance by removing any dependency on SoC-level control processors to schedule the execution of processing tasks inside the subsystem. Clearly, this delineation affects both hardware and software. While the concept of SoC design by composition of subsystems through standardized hardware interconnect interfaces [45][65] is increasingly applied in SoC platforms, an accompanying software platform with a well-defined programming interface to the subsystems is rarely found in the literature. Current efforts to standardize programming interfaces for audio and video streaming in OpenMAX [106], and 3D graphics in OpenGL 2.0 for embedded platforms are a first step in this direction. Nevertheless, neither interface addresses subsystems that consist of more than one processor.

2.4 Multiprocessor architecture templates

This section aims to place the Eclipse architecture in a larger context of SoC and SoC subsystem architecture templates. We therefore discuss a selection of multiprocessor architecture templates, both at the SoC and subsystem level. The architecture templates are selected on their relation to Eclipse, and/or to display a broad spectrum of architectures with respect to the architectural trade-offs discussed in this chapter. Rutten [119] provide more extensive overviews.

2.4.1 SoC architecture templates

Table 2 gives an overview of three SoC-level architecture templates: NXP Semiconductors Nexperia-DVP [104][32], TI OMAP [28], and the Philips Research CAKE architecture [42][135].

Table 2. Overview of SoC architecture templates.

	Nexperia-DVP	OMAP	CAKE
Target	Media processing (audio/video coding, image enhancement, connectivity) for digital TV, digital video recorders, and set-top box	Media processing (video messaging, video conferencing, web browsing) for 2.5G/3G wireless handsets and PDAs	High-end media processing (image enhancement, content analysis, H.264 coding) for new products and prototyping

	Nexperia-DVP	OMAP	CAKE
Programming model	G+4 (control), TSSA (streaming)	API for communication between CPU and DSP, user interface, and OS services. DSP tasks under CPU control. DSP tasks communicate through sockets.	C/C++, POSIX threads, YAPI, TSSA
Processors	VLIW (TriMedia), RISC CPU (MIPS), hardwired subsystems (MPEG decoder), peripherals	RISC CPU (ARM9), DSP (TX c55x), and peripherals	Homogeneous set of heterogeneous tiles that contain a number of VLIW processors (TM-Light), CPUs (MIPS), and accelerators (embedded FPGA)
Data path	32-bits bus to shared off-chip DDR memory	DMA to on-chip SRAM and 16-bits bus to off-chip SDRAM	Shared L2 cache to shared on-chip memory inside a tile; packet-based torus network between tiles
Data transport	Frame buffers in off-chip shared memory, data and instruction caches in the processors.	Frame buffers in off-chip shared memory, data and instruction caches in the processors.	Cache coherent shared-memory access
Synchronization	Software (interrupts). All subsystems execute under software control	Software (mailbox interrupts).	Hardware (cache coherency). Software packet routing between tiles
Multitasking	Fixed-priority preemptive task scheduling on CPU and TM	Standard OS on the CPU, Proprietary kernel on the DSP	Non-preemptive task scheduling with dynamic task migration
Data transport size	128 byte (cache-line refill)		L1 and L2 cache line refills
Synchronization size	Field		Macroblock up to field
Task switch rate	~1 kHz		

Nexperia-Digital Video Platform (NXP Semiconductors)

The NXP Nexperia-Digital Video Platform (Nexperia-DVP) [104] specifies a family of SoCs and a software platform for advanced set-top box, digital-video recorder, and digital television products. Nexperia-DVP primarily targets source decoding, comprising conditional access, descrambling, MPEG-2 video and audio decoding, graphics generation, image enhancement, image composition, and display. The platform revolves around a MIPS and TriMedia combination, augmented with a large number of dedicated subsystems—such as a video scaler and a slice-level MPEG-2 video decoder—and peripherals. The MIPS processor performs system control functions and runs an OS (e.g. WindRiver’s VxWorks or Linux). The TriMedia processor executes a variety of DSP functions in time-shared mode, such as MPEG stream demultiplexing and header parsing, audio processing (AC3, PCM, AAC), and soft-modem (V.34, V.90). The Viper

(PNX8525) [32] and Viper 2 (PNX8550, featuring 2 TriMedia cores) chips form the carrier products of the Nexperia-DVP initiative.

The Nexperia-DVP software platform furnishes a high degree of transparency and uniformity across processors. The programmable processors and dedicated subsystems each run their own thread(s) of control, offering efficient task-level parallelism at system level. Task scheduling mechanisms include priority-based scheduling using rate-monotonic analysis (RMA), reservation-based scheduling, and static scheduling in the uncompressed video domain. All functions share access to a uniform off-chip memory for inter-processor communication. Off-chip memory enables allocation of communication buffers that are large enough to achieve run-time system-control in software with acceptable interrupt rates. The TriMedia Streaming Software Architecture (TSSA) plays a central role in the handling of audio and video streams, addressing inter-processor synchronization, data exchange, buffer management, and application setup.

Intended as Nexperia subsystem, Eclipse is designed to copy the key properties of the TSSA programming model: memory-to-memory streaming with separated synchronization and data transport and software-controlled application configuration. Due to Nexperia-DVP's unified memory architecture, all subsystems share access to a single memory bus connected to external DDR RAM. Eclipse relies on on-chip memory to hide its inter-coprocessor communication from the SoC interconnect, and thus avoids additional traffic on this critical system resource.

OMAP (Texas Instruments)

Implementations of the OMAP platform enable multimedia capabilities—such as video messaging, web browsing, video conferencing, etc.—in 2.5G and 3G wireless handsets and PDAs. OMAP defines a heterogeneous multiprocessor architecture. OMAP1510 and OMAP5910 devices revolve around a TI enhanced ARM9 for general purpose tasks and OS/user interface processing, and a TI c55x DSP offering high efficiency for real-time signal-processing tasks. A separate DMA engine supports direct data streaming between CPU and DSP.

OMAP supports the Open Mobile Application Processor Interface (OPMAPI) to ensure interoperability between chips of different suppliers. OMAP platform centers around the standardized interfaces—called *sockets*—between subsystems based on the Open Core Protocol (OCP) [65]. These sockets cover dataflow aspects such as endianness, address granularity, as well as clock cycle budgets and interface signal timings, interrupt and reset signals, and bus access behavior.

OMAP's eXpressDSP software provides an abstraction layer for hardware services written for a range of hardware implementations (e.g. DMA), and a modular kernel for the DSP—called DSP/BIOS—that includes virtual channels for DSP-CPU communication. OMAP algorithms follow strict rules for memory use and access to hardware resources, facilitating the integration of externally developed algorithms.

OMAP's focus on sockets as the SoC subsystem's interface—including bus protocols, signal timing behavior, and guidelines/rules for algorithm development—paves the way for easy integration at SoC level. Eclipse relies on Nexperia-DVP SoC integration in a similar fashion, using standardized communication protocols (Philips proprietary protocols: DTL for communication between devices and MTL for communication to off-chip memory). At the application level, OMAP algorithms explicitly specify exact (typical and worst-case) timing, DMA, and memory requirements [139]. How to specify overall timing behavior with the highly dynamic workload of Eclipse's media-processing applications and the distributed scheduling of Eclipse coprocessors is topic of further research.

CAKE (Philips Research)

The aim of the Philips Research CAKE architecture [42][135] was to provide a low-cost competitor to the Sony CELL architecture, focusing on the video and audio domain rather than gaming. The highly programmable CAKE architecture can be deployed as a scalable platform for rapid market introduction of new functionality and as prototyping environment.

A CAKE *tile* clusters a heterogeneous mix of VLIW cores, CPUs, and accelerators, connected to memory. The cores' caches, called level-one caches, are kept coherent by means of a hardware cache-coherence protocol. A large second-level cache connects all level-one caches to shared memory. Multiple tiles can be interconnected by a packet-switched network to form a homogeneous network of tiles in ring or torus topology. The interconnecting of tiles gives an extreme scalability in performance, where an increasing number of tiles can be integrated in a single chip as IC technology progresses.

CAKE aims to reduce software development effort (time-to-market) at the cost of silicon area. The cache coherency protocol provides a shared-memory view within a tile, which renders flexibility in, for instance, allocating communication buffers or dynamically migrating tasks from one processor to another. The shared-memory view offers a large flexibility in porting application software. On top of the basic services provided by the tile run-time firmware (TRT), such as multithreading and semaphores, CAKE supports a range of programming models such as KPN (YAPI, TSSA), or POSIX threads.

Eclipse may offer a path to a low-cost CAKE tile once the product's functionality stabilizes and the volume ramps up. Internally, an Eclipse tile should not rely on hardware cache coherency, which is likely too costly for the small-sized Eclipse coprocessors. Instead, Eclipse exposes the inquiry/commit protocol directly in the programming model, with cache coherency as a beneficial side effect. Eclipse explicitly allocates FIFO buffers in on-chip memory for each stream. For streaming applications, it is unlikely that CAKE's approach with coherent caches can rival Eclipse's buffering scheme in reliable performance.

2.4.2 SoC subsystem architecture templates

While SoC-level templates are rapidly gaining more ground in the systems design community, templates for subsystem design are relatively unexplored. Table 3 gives an overview of four SoC subsystem architecture templates: the MOVE project at Delft University of Technology [26], Philips Research Prophid/CPA [136], Intel MXP [72], and Philips Research Eclipse [121].

Table 3. Overview of SoC subsystem architecture templates.

	MOVE	Prophid/CPA	Intel MXP	Eclipse
Target	Media processing functions in software (color conversion)	High-throughput, real-time video display processing (noise reduction, sharpness improvement, 100Hz conversion, multi-window TV)	Image processing (color-space conversions, G4 and JPEG image compression, filter operations) for digital copiers, scanners, and printers.	Irregular, soft real-time media processing (multi-stream video coding, image enhancement)
Programming model	C/C++, instruction-set architecture, transport triggered	KPN with static bandwidth assignment. Task graph configuration in a central controller	Dataflow, shared registers	KPN with separate data transport and synchronization. Task graph configuration distributed over all processors
Processors	Fine grain function units (ALU, lookup table, MAC, application-specific FUs)	Weakly programmable application specific processors (horizontal sample-rate converter, peaking)	Heterogeneous set of programmable image-processing processors (ISPs) with programmable function units (input, output, general purpose, MAC), memory, and accelerators (2D filters, Huffman encode/decode, DCT).	Media processor and weakly programmable application-specific processors (DCT, VLD)
Data path	Register files, network of move buses (no bypass network, bypass performed in software)	Switch matrix. Worst-case bandwidth assignment per field by global controller. 32 byte input/output FIFOs per processor	Crossbar with 8 programmable communication channels per ISP routes data to other ISPs, 18-channel DMA units, and expansion interfaces.	Bus to shared on-chip memory. Separate, tiny read and write caches per processor hide latency and bus width/alignment
Data transport	Data transports to/from register/FU, statically scheduled by the compiler.	Routing packets through channels defined in the switch matrix arbiter to input/output FIFOs in processor buffer memory	Routing 16-bit data elements through programmable communication channels between ISPS. Reading/writing to shared registers between function units within an ISP	Reading/Writing byte vectors to channel FIFOs in shared memory

	MOVE	Prophid/CPA	Intel MXP	Eclipse
Synchronization	Each FU has assigned one of its operand registers as <i>trigger register</i> . The operation start with a move to an FU's trigger register	Data-driven. Input/output buffer space available via hardware signaling	Semaphore mechanism to synchronize data transfer between function units within the ISP.	Data-driven. Separate from data transport. Distributed buffer space administration in hardware
Multitasking	Not supported	Distributed, non-preemptive task scheduling in hardware, local to the processors based on packet/buffer space availability. Task state memories per processor.	Not supported, the five function units inside the ISP operate concurrently.	Distributed, non-preemptive task scheduling in hardware, local to the processors based on buffer space availability and cycle budgets
Data transfer size	4 byte (1 operand)	16 bits (one pixel)	16 bits	128 bits
Synchronization size	4 byte (register)	32 byte (FIFO buffer empty/full)	16 bits	~100 byte (DCT block)
Task switch rate	-	~4 MHz (service cycle of 4 time slots of 16 clock cycles on the switch matrix)	-	~1 kHz (dependent on task budget setting)

MOVE (Delft University of Technology)

MOVE architectures are based on the transport-triggering concept. Transport-triggered architectures (TTA) are programmed by specifying data transports, or *moves*, between function units and with register files, instead of operations performed by function units in classical instruction-set processors. TTAs can be compared to VLIW architectures (e.g. TriMedia). Whereas each VLIW instruction specifies a number of RISC-like instructions, the fields in a TTA instruction specify the required data transports. These transports may trigger operations as side effect. Compared with traditional operation-triggered architectures, the main advantage of TTAs lies in the simplicity of the design, and in the new possibilities for compiler optimizations [26].

The MOVE framework exploits these advantages for semi-automatic generation of application-specific MOVE processors. The framework comprises:

- *Software framework*. Generation of instruction-level parallel code from C/C++ (compiler, assembler, architecture simulator and debugger);
- *Hardware framework*. Processor generation (synthesizable VHDL) based on a generic architecture template, also used to assess timing, area, and power consumption of a TTA instantiation;
- *Optimizer*. Exploration of the design space for a specific application using the former components. It generates the set of architecture parameters to be evaluated next.

MOVE architectures can be deployed as the core of an application-specific SoC subsystem. The MOVE framework is well suited for Eclipse coprocessor design, similar to Krishnan's [86] study of a VLIW-style DCT coprocessor using A|RT designer. An open question is in how far the MOVE architecture template can support the interaction with the Eclipse coprocessor shell.

Prophid/CPA (Philips Research)

The Prophid architecture template targets real-time video-display processing, defining a data-driven, or stream-based multiprocessor SoC subsystem [91]. The CPA forms a specific instance of the Prophid architecture template [136][76].

Prophid follows the Kahn programming model, consisting of tasks interconnected by logical FIFO channels. Each task is mapped to an application-specific processor, such as the Advanced Dynamic Noise Reduction (ADNR) processor, or the horizontal/vertical sample rate converter. Channels are implemented by a non-blocking, reconfigurable communication network—a programmable switch matrix in the CPA—and input/output FIFOs local to the processor. Prophid realizes its non-blocking communication network by allocating worst-case bandwidth to tasks with hard real-time constraints, and providing sufficient buffering in a special DMA processor to hide latency to external memory. For most applications, the required bandwidth allocation can be computed off-line.

CPA processors are multi-tasking, with dedicated input/output FIFOs and state memory per task, allowing task switches on a clock-cycle basis. Hereto, each processor is surrounded by a generic *processor shell* that dynamically schedules the tasks to execute on the processor based on data availability. Another function of the processor shell is to provide the interface to a control bus. Via this interface, the processor can load parameters or instructions that control the mode settings for the different tasks that execute on the processor.

The *global controller* takes care of the application configuration and related communication network setup. The application configuration is programmed by a host CPU and loaded into one of the two configuration memories in the global controller. These two *partitions* enable a gradual and run-time controlled transition between two unsynchronized application graphs.

Like Prophid, Eclipse provides an infrastructure that allows application-specific processors to operate in parallel and communicate in a flexible manner. Prophid paved the way for data-driven, autonomous processors. Eclipse inherits Prophid's concepts for distributed task scheduling and a generic processor shell. Eclipse's highly irregular, soft real-time applications—as well as the inclusion of application tasks executing on a media processor—led to a different communication infrastructure with larger communication buffers allocated in shared memory.

MXP digital media processor (Intel)

The Intel MXP digital media-processor family, a joint effort between Intel and Xerox, targets document image-processing products such as digital copiers, scanners, and printers. The architecture consists of 4 to 8 application-specific processors—called image signal processors (ISPs). Each ISP interfaces through eight programmable communication channels: two pairs of memory channels plus the six channels used for inter-ISP communication or to drive the off-chip expansion interfaces. The ISP is connected to its communication channels through a programmable crossbar switch that allows input channels to be routed directly to output channels, bypassing the ISP. Each communication input/output port contains a FIFO buffer of two 16-bit words.

Each ISP consists of five function units: input/output units to connect to the eight communication channels, a general-purpose unit with bit-rotation instructions, and two multiply-accumulate units. These function units are based on a general-purpose programming element and extended with domain-specific functionality. Each function unit encompasses local storage for its micro-coded instructions, 16 data registers, and a general-purpose ALU. The ISP also embeds up to two dedicated accelerators (e.g. 2D filters, Huffman encode/decode, DCT). A programmable DMA unit inside the ISP streamlines access of eight data streams to internal SRAM for scratchpad use and table lookup. All units within the ISP communicate data through 16 shared, general-purpose registers, each synchronized via a data-valid register that serves as semaphore.

The ISPs operate fully autonomous in a data-driven fashion, similar to the operation of Propid/CPA processors and Eclipse coprocessors. While the top-level architecture bears many similarities with Prophid/CPA, Intel's MXP deploys a next level of multiprocessor architecture by embedding different instantiations of its ISP template. Eclipse's coprocessor shell template resembles this approach, albeit only addressing communication aspects. Inside the ISP, the function units—based on a generic instruction-set processor with domain-specific extensions—further exploit the concept of architecture templates. The ISP template with programmable function units, accelerators, SRAM, and shared-register communication also resembles a scaled-down version of a CAKE tile (Section 2.4.1). Krishnan [86] studied a similar approach for Eclipse coprocessors, consisting of a dedicated instruction-set processor surrounded by read and write engines. Due to resource limitations within the Eclipse project, such a generic coprocessor template has not been worked out further.

3 Eclipse architecture template

The Eclipse aim is to provide system designers with a means to rapidly create cost-effective SoC subsystems, where each subsystem can be deployed across a range of media-processing applications. To this purpose, Eclipse provides an architecture template at the SoC subsystem level as depicted in Figure 6. Although subsystem-level templates are relatively unknown, they are essential in supporting reuse of design effort by providing a set of parameterized rules for subsystem composition. This chapter describes our subsystem architecture philosophy and outlines the separation between the generic infrastructure and function-specific coprocessors that forms the basis of the Eclipse architecture template.²

Figure 6. Overview of the Eclipse architecture template. This block diagram summarizes the Eclipse concept. It implements strict separation between medium-granularity function-specific coprocessors and generic infrastructure. The shells realize application flexibility by interconnecting coprocessors in a programmable manner. The media processor (CPU/DSP) can also share in the application processing via its own shell.

The Eclipse template targets media applications combining soft real-time and dynamic behavior. To maximize the deployment of a given Eclipse instance, each implementation targets an application *domain* within the realm of the media-processing domain. To ensure cost-effective solutions, Eclipse instances combine the performance density of function-specific hardwired coprocessors with the flexibility of one or more programmable cores. When an Eclipse instance is programmed for a particular application, it links its coprocessors and programmable cores into a network that mirrors the application structure.

Although they are function specific, the coprocessors are designed to be generic within the chosen application domain. This ensures cost-effectiveness by enabling reuse of the coprocessors within the Eclipse instance over a set of similar applications. Cost-effectiveness as well as the deployment scala of a given Eclipse instance is further increased by enabling time-shared use of the coproces-

² This chapter is based on Rutten et al. [124][121]. Section 3.2 is based on [120].

sors—allowing reuse of coprocessors in concurrent execution of a set of applications, as well as reuse of coprocessors within the execution of a single application.

3.1 Eclipse architecture philosophy

The Eclipse template exploits application-level parallelism by concurrently executing medium-grained functions in function-specific coprocessors and/or software executing on a media processor or DSP. Functions eligible for coprocessor implementation are those common to media applications, such as the DCT transform used by decoders and encoders for JPEG, MPEG, and digital video applications. Typically, the functions eligible for software implementation are specific to one application only—such as still-texture decoding in MPEG-4—or are likely to change as standards evolve. At runtime, the Eclipse infrastructure links these medium-grained functions into a Kahn-style application graph, using on-chip communication and data buffering.

Figure 7. Application mapping to the Eclipse architecture template. An application is depicted as a set of interconnected tasks. Tasks communicate with each other via buffered data streams. Tasks read and write data to a data stream through input and output ports respectively. Tasks execute on either the DSP or hardware coprocessors. The dashed arrows show the mapping of the application (on top) to the architecture (at the bottom). In this mapping, the DSP executes task A, coprocessor 1 executes both task B and C in a time-shared fashion, and another coprocessor executes task D.

Figure 7 shows how application tasks and stream buffers map to the Eclipse coprocessors and FIFO buffers allocated in on-chip memory. Unlike fully hardwired SoC subsystems, the Eclipse infrastructure is programmable and provides the flexibility to configure a given Eclipse instance for different application graphs. Eclipse coprocessors exploit the performance density of dedicated hardware function units and are only weakly programmable.

3.1.1 Concurrency

The irregularity of media applications led us to design Eclipse for average-case performance, leading to a cost-effective solution compared with a worst-case design. Consequently, Eclipse is designed for soft real-time processing. To reduce the effect of load variations, Eclipse applications are deployed as memory-to-memory functions. The inputs and outputs of the Eclipse application are buffered in large (e.g. frame) buffers in off-chip memory. In addition, Eclipse applications must support a graceful degradation mechanism in overload situations—such as frame skipping in MPEG—to avoid further deadline misses whenever the slack provided by the input/output buffers is insufficient.

The Eclipse template ensures scalability by avoiding complex centralized modules that control large parts of the architecture. Thus, all Eclipse coprocessors execute in parallel and autonomously, without requiring CPU support for task scheduling or synchronization. New silicon technologies allow efficient coprocessors with sufficient computation speed for time-shared use. Such multitasking coprocessors are essential for configuring a range of applications and reapplying the same coprocessors at different places in an application task graph. Each coprocessor can execute multiple tasks from a single Kahn network or from multiple, possibly different, networks in a time-shared fashion. Thus, application complexity is not restricted to the number of coprocessors in the architecture. Moreover, the programmable media processor or DSP can perform part of the application functionality when an application requires functionality beyond the implemented set of coprocessors.

3.1.2 Infrastructure

The strong flexibility requirements led us to design the Eclipse infrastructure with a centralized memory in which communication buffers can be allocated at run-time. Here, the flexibility benefit of a centralized and shared memory outweighs the scalability benefit of distributed memory. The flexible connection of medium-grained functions requires significant communication bandwidth from the system. For media processing, the streaming nature of the application functions generates a high locality of reference, that is, consecutive references to the memory address of neighboring data. Eclipse instances exploit this characteristic to provide high data throughput (GBytes per second) through deployment of a shared wide (for example, 128 bits) bus combined with communication buffers in a wide on-chip memory. The choice of bus and memory width gives some level of scalability. The clear separation between infrastructure and coprocessors outlined below also facilitates the application of multiple, possibly distributed memories at the cost of reduced flexibility.

In designing Eclipse, we strictly separated application functionality from the generic interconnect structure. The resulting uniform interface separates computa-

tion hardware, or coprocessors, from generic support for multitasking, synchronization, and data transport. This interface not only keeps coprocessor design simple but also facilitates coprocessor reuse over a set of media applications. The interface copes with irregular and unpredictable application loads by separating data transport from data access synchronization. The combined requirements of scalability and cost-effectiveness led us to a novel approach: distributed scheduling and distributed synchronization with high task-switching and synchronization rates, supported by a generic hardware implementation dedicated to each coprocessor called the *coprocessor shell*.

3.1.3 Avoiding deadlock

The architect of a concurrent system such as Eclipse must ensure that deadlocks due to resource constraints cannot occur. In implementing the Kahn model, buffer sizes must be bounded. Exposing concurrency (through the infinite buffers in the Kahn model) on an implementation with finite resources (buffers) may lead to artificial deadlock. Eclipse tasks cannot continue processing when there is insufficient room in an output buffer for writing. Eclipse replaces the traditional blocking write with a non-blocking inquiry—the `GetSpace` primitive detailed in the next section—that allows the coprocessor to switch to another task when the inquiry indicates there is insufficient room in the buffer. Permitting a task switch on a full buffer avoids deadlocks caused by resource reservation.

Small buffer sizes may conflict with the application's communication requirements and lead to artificial deadlock. Computing the required buffer sizes for an application off-line is known to be difficult for applications that can be statically analyzed as the required buffer sizes depend on the execution schedule. For data-dependent applications, such off-line calculation is impossible. Basten and Hoogerbrugge [8] dynamically, at run time, detect deadlock caused by restricted buffer sizes and resolve the deadlock by increasing critical buffer sizes. Eclipse supports such mechanisms with measurement hardware in the shells and control software running on a control processor (CPU) that can increase buffer sizes at runtime by changing the buffer allocations in shared memory and configuring the new buffer sizes in the shells. Even in the case of deadlock, the control processor can access the memory and program the shells via a separate configuration bus/interface, not depicted in Figure 7.

The Eclipse communication network consists of separate networks for data transport and synchronization. The synchronization network is used to exchange synchronization messages between coprocessor shells such that each shell can locally maintain the buffer filling of the streams to which its coprocessor tasks connect. To avoid communication deadlock, the shells always accept incoming messages, even when the message relates to a task that is currently not executing on the co-

processor. Hence, sending messages on synchronization network can never block a coprocessor task.

In Kahn process networks, the output is independent of the order in which tasks execute. This property also holds for Eclipse, provided that the coprocessors do not implement non-deterministic behavior by misusing the Eclipse interface. To avoid deadlock by keeping a coprocessor (a compute resource) occupied, the coprocessor should not busy wait for input data or output room (Section 4.2.1) but instead free the coprocessor and allow other tasks to make progress. The budget mechanism in the task scheduler (Section 5.3) in each coprocessor shell ensures that tasks cannot monopolize the coprocessor and cause starvation.

With the above measures to avoid deadlocks introduced by the architecture, the only other causes of deadlock are in the application itself, which for instance may occur on cycles in the application graph. This is a general issue in Kahn process networks. However, industrial practice has shown that it is relatively easy to avoid deadlock for the targeted multimedia applications.

3.2 Separation of function and infrastructure

The infrastructure that transports and synchronizes data between coprocessors can vary between different Eclipse instances to match communication bandwidth requirements. The *coprocessor shell* shown in Figure 7 facilitates reuse of a coprocessor design among different instances with different communication network characteristics by introducing a stable interface that separates the coprocessor design from the infrastructure, and vice versa. Moreover, the shell simplifies coprocessor design by absorbing many system-level issues, such as multitasking, stream synchronization, and data transport. Thus, coprocessor designers can concentrate on application functionality.

The architecture template deploys distributed shells, such that each shell can be close to the coprocessor it serves. The coprocessor-shell interface defines a uniform set of primitives with coprocessor-specific parameters, such as the data path's width. We designed the shell's internal architecture as a parameterized template to facilitate reuse in an Eclipse instance. A product engineer derives shell instances with coprocessor-specific parameter settings from this generic template. Examples of such parameters are the data width of the read and write interface between the coprocessor and shell and the size of data caches in the shell. Although this thesis focuses on the coprocessor shell, the described concepts also apply to the shell of the media processor (the DSP in Figure 7). The media processor shell can implement part of its functionality in software, as Figure 7 shows, to increase flexibility and reduce hardware cost.

Each coprocessor interacts with its shell through six generic interface primitives: **GetTask**, **GetSpace**, **PutSpace**, **Read**, **Write**, and **Report**. Figure 7 represents the primitives as the *task-level interface* between coprocessors and their shells. For

all six primitives, the coprocessor has the initiative, and the shell provides the requested service as a slave. All interface calls block until the shell replies. Coprocessor functions (tasks) issue the read, write, and synchronization (**GetSpace** and **PutSpace**) requests through input or output ports. Ports are logical entities used to link a request to a particular data stream, and are not directly reflected in physical interfaces. Different tasks on the same coprocessor may have a different number of input and/or output ports. The implementation of the six primitives gives rise to only five different physical interfaces between the coprocessor and the shell, as the **GetSpace** and **PutSpace** primitives share a single physical interface. All five physical interfaces use a simple handshake-protocol for communication.

The following subsections give the syntax for each primitive and explain briefly the corresponding semantics. Although developed for Eclipse, the six interface primitives are broadly applicable in multiprocessor solutions for the media-processing domain, both in software and in hardware. Moving into the realm of networks on silicon, such an interface will be a key element in providing a structured approach for building complex SoCs.

3.2.1 GetTask

```
task_id = GetTask( &task_info );
```

The coprocessor calls this primitive whenever it allows a task switch to another task mapped on the coprocessor. By calling **GetTask**, the coprocessor signals to the shell that the coprocessor is ready to start a new task execution. The service request means: “What should I (the coprocessor) do next?” The shell responds by furnishing 2 pieces of information: `task_id` and `task_info`. The `task_id` identifies the next task to execute on the coprocessor. The `task_id` is needed as parameter in subsequent **GetSpace**, **PutSpace**, **Read**, **Write**, and **Report** calls. Only if the coprocessor implements state saving on task switches, `task_id` may also be used to identify task states. The `task_info` is provided to the coprocessor to indicate the precise function to be executed in the task. The interpretation of this bit field is done by the coprocessor, not by the shell. The field may indicate to the coprocessor for example whether a forward or inverse DCT is to be executed, or may contain pointer information where anchor frames are located in global memory.

3.2.2 GetSpace

```
is_granted = GetSpace( task_id, port_id, size );
```

The coprocessor calls this primitive to inquire about available data for reading, or available room for writing. The amount of data available for reading and the amount of room available for writing is indicated by the uniform term *space*. The service request means: “Is there space of a certain size available in the stream?” The `task_id` and `port_id` parameters are supplied by the coprocessor to indicate

for which task and port it is requesting the service. The size parameter indicates how many bytes of space the coprocessor is inquiring. The shell responds with a binary answer: yes or no. A ‘yes’ answer provides the task with access rights to the requested space.

The shell internally combines the `task_id` and `port_id` arguments into an identifier of the associated stream. Although `GetSpace` and `PutSpace` do not distinguish between input and output ports, a `GetSpace` on an input port inquires about available data for reading, whereas a `GetSpace` on an output port inquires about available room for writing.

3.2.3 PutSpace

```
PutSpace( task_id, port_id, size );
```

The coprocessor calls this primitive to commit some data that has been written, or room that has been created by reading available data. A `PutSpace` call on an input port commits empty room available for writing, whereas a `PutSpace` call on an output port commits valid data written in the stream buffer. The amount of data that has been written and the amount of room that has been read is indicated by the uniform term *space*. The service request means: “Please update locally and signal to whom else in the system it may concern that some space has become available.” The `task_id` and `port_id` parameters are supplied by the coprocessor to indicate for which task and port, and thereby for which data stream, it is requesting the service. The size parameter indicates how many bytes of space the coprocessor is releasing. The coprocessor is supposed to commit only space that the coprocessor previously acquired through `GetSpace`.

3.2.4 Read

```
Read( task_id, port_id, offset, size, &data );
```

The coprocessor calls this primitive to read some data from a particular stream, indicated by task and port ID. The service request means: “Please provide me so many bytes of data from this stream at that offset.” The `task_id` and `port_id` parameters are supplied by the coprocessor to indicate for which task and port it is requesting the service. The size parameter indicates how many bytes of data the coprocessor is requesting. The offset parameter indicates the offset in bytes from the current point of access in the data stream, as maintained in the shell. This is detailed in Section 4.1. The shell answers to the read request by furnishing the requested data by value in the data parameter. The coprocessor is supposed to read only data from space that the coprocessor previously acquired through `GetSpace`.

3.2.5 Write

```
Write( task_id, port_id, offset, size, data );
```

The coprocessor calls this primitive to write some data to a particular stream, indicated by the task and port ID. The service request means: “Please write so many bytes of data into this stream at that offset.” The `task_id` and `port_id` parameters are supplied by the coprocessor to indicate for which task and port it is requesting the service. The `size` parameter indicates how many bytes of data the coprocessor is offering to write. The coprocessor is supposed to write only data into space that the coprocessor previously acquired through `GetSpace`.

3.2.6 Report

```
Report( task_id, report_type, report_id );
```

The coprocessor calls this primitive to report some significant event to the system, such as an error or the detection of a particular state in the processing, such as the end of an MPEG frame. The service request means: “Please note that the following event occurred, and notify if necessary the configuration manager.” The `task_id` parameter is supplied by the coprocessor to indicate for which task it is requesting the service. The `report_type` parameter indicates which type of event occurred: an error or the encounter of a particular event in the processed data stream, called a *location*. The mechanism of adding ‘location events’ to the data stream is subject of Section 4.3. The `report_id` parameter specifies which specific event occurred (error number or location number).

4 Eclipse coprocessor control

In the design of Eclipse coprocessors, Kahn application models are gradually refined into task-level code that uses the six Eclipse primitives. The refined code forms the starting point for the low-level coprocessor design. This chapter shows how we use the six Eclipse primitives of the previous chapter (Chapter 3) in domain-specific coprocessors.³ The response to the coprocessor's requests is implemented in the coprocessor shell, which is the subject of the next chapter (Chapter 5).

Eclipse coprocessors explicitly specify the moments at which they can switch between tasks, thereby avoiding the hardware costs required for state saving at arbitrary times. Multitasking on Eclipse coprocessors is a shared responsibility of the shell and the coprocessor. The shell handles task scheduling whereas the coprocessor provides task switch points and saves and restores the task state (if any) upon a task switch. The coprocessor can continue to the point where it has minimal or no state. At that point, the coprocessor asks its shell for its next task by calling the `GetTask` primitive. We refer to the intervals between `GetTask` inquiries as *processing steps*. The coprocessor executes an infinite loop of such processing steps.

Note that processing steps must be bounded in time to ensure robust execution of the cooperative task scheduling between coprocessor and shell. In addition, every coprocessor must accommodate a (hardware) reset input through which a control task can reset the coprocessor to a predefined state in case of an error condition. These restrictions are part of the acceptance criteria for embedding a new coprocessor into an Eclipse instantiation.

The simplified code in Figure 8 shows a top-level coprocessor control loop as an example of multitasking coprocessor design using the six Eclipse primitives. The use of the `Report` primitive is omitted for simplicity. The pseudo code illustrates the coprocessor control to handle multitasking, synchronization, and data communication needed to execute the coprocessor functionality, as implemented by the `compute` function. The coprocessor only builds up task state by reading from the input stream after it ensures it can complete the processing step successfully. Before the task starts a `Read` or `Write`, it uses `GetSpace` to test whether sufficient data is available for reading and sufficient room is available for writing. After a

³ This chapter is based on Rutten et al. [124][121]. Sections 4.1.1 and 4.1.3, as well as Section 4.2 are reworked from Rutten et al. [123]. Section 4.2.2 builds on patent [39]. Section 4.3 is based on Rutten et al. [129]. Refer to Van Eijndhoven [35] for additional code examples and a description of pipelined coprocessor design.

Read or **Write**, the amount of generated data or room is committed to the shell via **PutSpace** calls.

```
while(true) {
    // Perform a single processing step
    task_id = GetTask(&task_info);

    // Is there data/room for reading/writing?
    blocked = !GetSpace(task_id, IN, INSIZE)
        || !GetSpace(task_id, OUT, OUTSIZE);
    if (blocked) continue; // No useful work to do

    Read(task_id, IN, 0, INSIZE, &in_data);
    PutSpace(task_id, IN, INSIZE); // Commit room

    Compute(task_info, in_data, &out_data);

    Write(task_id, OUT, 0, OUTSIZE, out_data);
    PutSpace(task_id, OUT, OUTSIZE); // Commit data
}
```

Figure 8. Example of a top-level coprocessor control loop.

4.1 Data access synchronization

From a coprocessor task port's viewpoint, a data stream looks like an infinite data tape with a current access point, as Figure 9 shows. With the **GetSpace** call, the coprocessor asks the shell permission for access to a certain data space ahead of the current access point. Data space signifies available data for reading from an input data stream, or available room for writing data to an output stream. If the shell grants permission, the coprocessor can perform **Read** or **Write** actions inside this requested space using variable-length data (through the size argument), and at random access positions (through the offset argument). When sufficient data or room is not available, the shell denies permission by returning false on the **GetSpace** call.

The coprocessor is responsible for functionally correct behavior when using the interface primitives. For example, the coprocessor must adhere to denied **GetSpace** requests and must not attempt to read or write data outside the window of granted space. Thus, when **GetSpace** fails, the coprocessor task cannot proceed, and either the coprocessor can switch tasks or the task can keep trying to proceed by repeatedly issuing **GetSpace** requests. After one or more **GetSpace** calls—and optionally several **Read/Write** actions—the coprocessor can decide it is finished with processing the data and issue a **PutSpace** call. This call advances the access point a specified number of bytes ahead, in a size constrained by the previously granted space.

Figure 9. Synchronization and data I/O through a single port: (a) initial 'data tape' situation with the current access point, (b) `GetSpace` action provides a window on requested space, (c) Read/Write actions on contents, (d) `PutSpace` action moves the access point ahead.

Synchronization of data transport—implemented through the `GetSpace` and `PutSpace` synchronization primitives—is fully separated from the actual data transport, implemented through the `Read` and `Write` primitives. The size argument of `GetSpace` and `PutSpace` calls allows the coprocessor to synchronize streams at a granularity and rate that differs from the individual `Read` and `Write` calls. The number of bytes that can be transferred on a single `Read` or `Write` request is restricted to the width of the corresponding coprocessor-shell interface. Thus, a coprocessor may need to issue multiple reads or writes to transfer one logical unit of data, such as a block of 8x8 DCT coefficients. Synchronization is mostly done at a data grain that is meaningful to the application context in order to avoid building up internal state between task executions. Within this constraint, the coprocessor designer must balance the overhead incurred by a high synchronization rate versus the high buffer requirements of synchronizing between coprocessors at a low rate.

The coprocessor implements the synchronization behavior of Figure 9 by maintaining the space (data or room) acquired by successful `GetSpace` calls per task port. On every granted `GetSpace` of g bytes, the coprocessor updates this internal state *acquired_space* of the port:

$$acquired_space = \max(acquired_space, g). \quad (1)$$

On every granted `PutSpace` action of p bytes, the coprocessor reduces the acquired space:

$$acquired_space = acquired_space - p. \quad (2)$$

For correct operation, the coprocessor must satisfy the constraints of (3-5) on each task port. In essence, the coprocessor task may only commit data via

PutSpace that is previously granted by **GetSpace** (3), and the coprocessor cannot write or read beyond previously granted space (4, 5).

$$p \leq \text{acquired_space}, \quad (3)$$

$$\text{write_offset} + \text{write_size} \leq \text{acquired_space} \quad \text{for output ports}, \quad (4)$$

$$\text{read_offset} + \text{read_size} \leq \text{acquired_space} \quad \text{for input ports}. \quad (5)$$

In Kahn process networks, data access and synchronization are coupled; a task blocks whenever it tries to access unavailable data. As a result, the ordering of data in the streams is independent of the scheduling order of the tasks. With Eclipse's separated synchronization and data access, the coprocessor designer can potentially violate this basic property of Kahn networks. An example of such a violation is a coprocessor task that selects data from two inputs—the choice of input defined by their data availability (**GetSpace** return value)—and subsequently outputs the selected data to its output. This leads to erroneous behavior when a consumer task assumes a predefined ordering of the elements in the output stream of its predecessor. In Eclipse, preventing such non-deterministic behavior is a responsibility of the coprocessor designer.

4.1.1 Random access

Since synchronization is not coupled to individual read and write actions, a coprocessor can randomly access the data within an acquired window of granted space. The **Read** and **Write** primitives therefore allow a random offset from the current point of access through their **offset** argument. One example of random access is a coprocessor that uses a buffer in shared memory as scratch pad, for instance as texture memory in 3D graphics, look-up table, or for storing the state of a task. Contrary to data streams between coprocessors or tasks, this scratch-pad memory is allocated to one task only, and can therefore be accessed without **GetSpace/PutSpace** synchronization.

The Eclipse communication network is optimized for streaming data—through a wide bus to a wide shared memory. Moreover, the Eclipse shells incorporate small read and write caches that can perform automatic prefetching for streaming data access (Section 5.2). Therefore, random access on Eclipse streams should be used cautiously. For instance, the transpose buffer between horizontal and vertical DCT operations can be mapped onto a shared memory buffer, accessed with the Eclipse primitives. However, the random byte-level access to this buffer will induce significant overhead on a bus with a minimal transfer size of 16 bytes. In this case, such a small buffer is better kept local inside the coprocessor.

Coprocessors requiring random access at a coarser grain can increase cost-effectiveness by mapping their buffers in shared memory. This occurs when the coprocessor jumps randomly between large groups of data in the buffer, but accesses

the data within a group in a streaming fashion. This is typically the case in DV decoding, where each macroblock is coded with a fixed number of bytes at a pre-determined position in the bitstream. Whenever the variable-length coded macroblock requires more than this reserved data segment, the remainder is added to a second segment for which the macroblock data does not occupy the full segment. The macroblocks in a frame are shuffled to distribute these macroblock tails evenly throughout the frame. Therefore, the macroblock data within a segment can be accessed sequentially, but random access is needed to access all pieces of bitstream that belong to a single macroblock.

4.1.2 Multicast

Figure 10 depicts a setup where a data stream produced by one task is consumed by two or more different consumers. We denote this stream property with the term *multicast*, conceptually described by Kahn and McQueen [81]. Applications can use such multicast service for instance to distribute a stream to a variable number of consumers, or to (at run-time) tap into a data stream for analysis and debugging.

Clearly, multicast can be avoided by assigning a separate buffer for each consumer and let the producer explicitly send its output data to output ports for each consumer. Such a straightforward approach both decreases the reusability of the producer as it is strictly tied to a number of consumers, and increases communication overhead in buffer size and bandwidth requirements.

Figure 10. Multicast: multiple consumers share a single stream buffer.

Figure 11. Data stream consumed by two independent tasks B and C. The arrows denote the current access points of tasks A, B, and C. The outlined box to the right of A shows the window of available room in which task A writes new data. To the left of A is valid data produced and committed by task A. The outlined boxes to the right of C and B show their respective acquired windows of valid data available for reading. Only when both B and C have completed and committed their read actions, the space left behind by B and C (left side of the picture) becomes available for writing to task A.

Eclipse implements the multicast mechanism in its reusable shells (Section 5.1.2). The separation of synchronization from data access allows a single-buffer implementation that is fully transparent to the coprocessors. That is, the producer has no knowledge of the number of consumers, and each consumer is unaware of other consumers sharing the stream buffer. Figure 11 depicts the data tape view for this setup. Each consumer *B* and *C* maintains its own access point into the buffer. The access windows of *B* and *C* may (partially) overlap.

4.1.3 In-place updates

Figure 12 illustrates a situation in which a task *B* performs *in-place* updates of the data in a stream buffer between two other task *A* and *C*. Compared to connecting separate input and output streams to task *B*, sharing the stream buffer between three tasks can be advantageous when task *B* only performs occasional modifications in the data stream and therefore does not need to transport the entire stream contents.

Figure 12. Task *B* performing *in-place* updates on the stream between *A* and *C*, without copying data.

This is for instance the case when task *B* mostly just watches the data, maybe by inspection of some header information only (not reading all stream data) and mostly allowing the data to pass from *A* to *C* without modification. Relatively infrequently, it could decide to change a few items in the stream. In a practical situation, the configuration manager may intervene in the communication between two hardware coprocessors to patch the stream to correct errors caused by hardware flaws, to adapt the stream towards slightly different stream formats, or simply for debugging purposes.

The separation of synchronization and data transport allows such an efficient implementation. Figure 13 shows the data-tape view of such a setup with three tasks sharing a single buffer in shared memory to reduce memory traffic and coprocessor workload. The figure shows the access points of tasks *A*, *B*, and *C*, where *A* is a writer and leaves valid data behind, *B* performs the *in-place* updates, and task *C* is a reader and leaves empty room behind.⁴

⁴ With the shell implementation of Section 5.1, task *B* actually communicates through two ports: a port for reading data produced by task *A*, and a port for writing data in the same buffer for consumption by task *C*. The synchronization scheme is left as an exercise to the reader (see Appendix A).

Figure 13. Data stream shared by tasks A, B, and C with in-place updates by task B.

4.2 Task switching

Eclipse coprocessors operate on a logical unit of data—such as an 8x8 block of DCT coefficients—encapsulated in a data packet. The coprocessors can have different packet consumption and creation patterns. If consumption at the input is synchronized with packet creation at the output ports, the coprocessor can switch tasks at moments when the data state is void. Typically, coprocessor state is minimal after the processing of a complete packet. For instance, a DCT coprocessor is virtually stateless after processing a block of DCT coefficients.

To avoid context switch overhead, Eclipse coprocessors usually process an integer number of packets in a single processing step. However, at the start of a processing step, the coprocessors cannot always determine the required amount of space for completing the processing step. This is the case if the coprocessor has a data-dependent condition upon which it must read more data from a second input port. In such situations, the coprocessor must inquire for additional space during a processing step and may be unable to continue executing the current task. The coprocessor designer can decide to let the coprocessor busy wait for the space to arrive or to suspend the current task by calling `GetTask`. In the latter case, the coprocessor may have to save and restore state information upon switching tasks. For instance, a variable-length decoder (VLD) must keep track of state information to correctly parse the remainder of an MPEG bitstream in subsequent processing steps.

The following subsections detail these options for handling task switching in the coprocessor. While Sections 4.2.1 and 4.2.2 are applicable for handling the aforementioned conditional I/O, Sections 4.3.3 and 4.3.4 describe state save/restore solutions that also apply in the more general case when task state is not void upon a task switch.

4.2.1 Busy wait

A trivial form of avoiding state save and restore is to avoid calling `GetTask` when the running task blocks on a negative answer to a `GetSpace` request. The coprocessor then ‘busy waits’ on the requested space to arrive by repeatedly calling `GetSpace`. While this simplifies coprocessor control, it endangers the reliability of

the system as other tasks mapped on the same coprocessor must wait for the blocked task to give up the coprocessor resource.

One could reason that whenever the coprocessor task already received some part of a packet, the rest of the packet will be produced soon enough to allow a busy wait. This only holds when the consuming task knows the precise behavior of the producer, creating a dependency between the otherwise independent tasks. However, this assumption is invalid, as the tasks running on Eclipse coprocessors can be connected in various application graphs. For example, the normal predecessor of a DCT task may be a VLD task on a dedicated coprocessor that always completes the production of a DCT packet without interruptions. In another application setup, the DCT input stream may be generated by DSP software, whose preemptive operating system may interrupt the production of a DCT packet for a long time—for instance to handle a higher priority task.

Thus, busy wait cannot be allowed without a time-out mechanism after which the task may need to save its state and must free the coprocessor to allow other tasks to meet their deadlines. Although this does not avoid state save/restore hardware cost, the busy wait with time-out improves performance by avoiding the context switch overhead whenever the requested data arrives before the time-out period.

Using busy wait increases the probability of creating deadlock. The Eclipse shells implement performance-measurement support in hardware. Run-time control by software can use these hardware measurements to detect deadlock and re-adjust application parameters. However, the targeted media-processing applications such as MPEG-2 decoding are sufficiently simple to allow an application expert to guarantee deadlock-free behavior.

4.2.2 Discarding partial work

A coprocessor does not have to surround each **Read** or **Write** request with **GetSpace** and **PutSpace** calls, but can postpone the **PutSpace** actions to the end of a processing step. As long as the coprocessor does not commit consumed or produced data by calling **PutSpace**, the input data remains available in the stream buffer. Thus, upon a negative answer to a conditional **GetSpace** request, the coprocessor can simply discard the current work and continue with another task. When the requested space becomes available, the coprocessor can restart the processing step from the beginning, re-computing the initial part of the processing step, as illustrated by the pseudo code in Figure 14.

The coprocessor control loop of Figure 14 implements a second exit point from the processing step: the **continue** statement inside the **if (more)** condition. However, a single entry point is maintained (the start of the infinite loop). If the second exit point is taken, a later execution for the same task_id will redo the initial part of the processing step, including **Read(IN,...)** and **ComputeA(...)**. The **Read(IN,...)** action will read the same data as before, since the example deliber-

ately postponed committing this read with **PutSpace(IN,...)** until a granted **GetSpace(IN2,...)** assures that the processing step can complete.

A practical example related to this way of postponing the commit of earlier read or write actions is the separate stream holding quantization tables as input to an MPEG-2 quantization coprocessor, detailed in Section 6.3.2. At the start of a new task, the coprocessor reads the quantization table from this input stream. This table remains available in the stream since the coprocessor only commits (via **PutSpace**) the read actions when a data packet from a second input stream—holding the data to be quantized—signifies in the header field of the packet that a new quantization table is required. The coprocessor may re-read the same quantization table from the stream for many processing steps before committing the table data and reading a new table.

```
while(true) {
    task_id = GetTask(&task_info);

    blocked = !GetSpace(task_id, IN, INSIZE)
              || !GetSpace(task_id, OUT, OUTSIZE);
    if (blocked) continue; // No useful work to do

    Read(task_id, IN, 0, INSIZE, &in_data);

    more = ComputeA(task_info, in_data);

    if (more) { // Conditional input
        if (!GetSpace(task_id, IN2, IN2SIZE))
            continue; // Abort processing step
        Read(task_id, IN2, 0, IN2SIZE, &in2_data);
        PutSpace(task_id, IN2, IN2SIZE);
    }
    PutSpace(task_id, IN, INSIZE);

    ComputeB(task_info, in_data, in2_data, &out_data);

    Write(task_id, OUT, 0, OUTSIZE, out_data);
    PutSpace(task_id, OUT, OUTSIZE);
}
```

Figure 14. Aborting a processing step without committing already consumed and produced space.

4.2.3 State save in internal coprocessor memory

The previous two sections avoided saving task state when the running task blocks on conditional I/O. However, discarding partial work is only effective when the cost of re-computation is lower than or equivalent to the cost of performing explicit state save and restore actions. The coprocessor can save and restore task state locally inside the coprocessor, allowing fast context switching without the

need for (externally visible) reads and writes. Clearly, this is only economical if the amount of state memory is relatively small.

The state memory can be seen as a state ‘vector’, indexed by the `task_id` return value of the `GetTask` primitive. In hardware, this is typically implemented as SRAM or a register file, where the most significant address bits are controlled by the `task_id`.

4.2.4 State save through a single-access buffer

Compared to dedicated state memory inside the coprocessor, saving/restoring task state to shared memory outside the coprocessor is more efficient when the size of the task state is considerable. This allows reuse of the memory for other purposes when the maximum number of tasks is not actually configured, as well as allowing more tasks to execute on the coprocessor by allocating a larger state buffer in software. An example where this is applicable is in multi-standard variable-length decoding, where the variable length coding tables may differ for each video standard.

The coprocessor only contains local state memory to hold the state for a single task, and generates reads and writes to replace this state upon a task switch. As the Eclipse communication network provides low-latency data access, these reads and writes pass through the same data path and memory as used for the media traffic. Each task is assigned a dedicated state buffer in shared memory that is not shared by others. Therefore, the coprocessor issues a sequence of `Read` and `Write` calls for state saving without requiring synchronization—without calling `GetSpace` or `PutSpace` on this stream. Note that the coprocessor only knows whether it needs to switch tasks *after* `GetTask` returns with a new `task_id`. Therefore, the `Write` calls pass the previous `task_id` in their first argument to refer to the state buffer of the previous task for saving the state of that task.

The pseudo code of Figure 15 gives a trivial example of this behavior. The example shows a situation where a task cannot issue `GetSpace` for the required output space upfront, as the size of the output data is only known after processing some input data. This situation occurs in variable-length decoding, where the size and type of the output data (picture properties, quantization tables, run-length pairs, etc.) is only known after decoding some part of the incoming bitstream.


```

while(true) {
    task_id = GetTask(&task_info);

    if (task_id != prev_task_id) { // GetTask returns a new task?
        // Save the previous task state
        // and restore the state of the new task
        Write(prev_task_id, STATE, 0, STATESIZE, state);
        Read(task_id, STATE, 0, STATESIZE, state);
        // Store current task id in coproc register
        prev_task_id = task_id;
    }

    // Can only GetSpace on input;
    // output size depends on input data
    // Immediately commit read data; compute is too
    // intensive to discard
    if (!state.writing && GetSpace(task_id, IN, INSIZE)) {
        Read(task_id, IN, 0, INSIZE, &in_data);
        PutSpace(task_id, IN, INSIZE);
        state.out_size = Compute(task_info, in_data, &state.out_data);
        state.writing = true;
    }
    if (state.writing && GetSpace(task_id, OUT, state.out_size)) {
        Write(task_id, OUT, 0, state.out_size, state.out_data);
        PutSpace(task_id, OUT, state.out_size);
        state.writing = false;
    }
}

```

Figure 15. Saving and restoring task state through a single access buffer. The coprocessor commits read data (**PutSpace** on the IN port) before the processing step is completed and thereby builds up internal state. If the subsequent request for empty room in the output buffer (**GetSpace** on the OUT port) fails, the coprocessor aborts and must save its state before continuing with another task. The coprocessor only saves/restores its internal state to/from the Eclipse buffer if it switches to a different task on the next processing step.

4.3 Dynamic reconfiguration

Application configuration concerns the creation of tasks—such as an IDCT or VLD task—and interconnecting tasks to form an application task graph, such as an MPEG-2 decoder. To create a task on an Eclipse coprocessor, the configuration manager (control processor) programs task parameters into the task table of the coprocessor’s shell. The interconnection of tasks entails among others the allocation of buffer space in shared memory and programming buffer address and stream identifiers for synchronization into the shell’s stream table. We define application configuration as all actions needed to set up an application task graph. These actions can be grouped into three categories (slightly adapted from Hofmeister [70]):

- **Topology:** Create and interconnect tasks.
- **Parameters:** Setting task parameters such as the output picture size.
- **Mapping:** The assignment of tasks to (co)processors and the allocation of buffers in memory.

The application configuration is defined separately from the application tasks that perform the actual processing. The flexibility of Eclipse to support a set of applications—and even concurrently execute different applications—implies that application tasks may be deployed in many different configurations.

A product engineer will configure the architecture for a set of predefined applications when deploying the architecture in a product. However, there will often be a large number of desired application configurations, also called ‘use cases’. The system typically moves from one configuration to the next as result of user interaction.

Clearly, hardware resources such as coprocessors and memory space are reused in many configurations. Allocating processing and memory resources for the sum of all configurations at design time is inefficient when the configurations do not execute simultaneously. For instance, configuring task graphs for all 9 possible combinations of high-definition MPEG-2 decoding and standard-definition MPEG-2 decoding and encoding in the Eclipse instance of Chapter 6 requires excessively large task and stream tables and shared on-chip memory. Instead, we adapt the application configuration at run-time, and allocate memory and table entries only for the set of applications that is currently required to execute. We refer to the action of adapting the configuration of a running application as *dynamic reconfiguration*.

Dynamic reconfiguration is required when the mode of operation of the system changes. Such a *mode switch* can be caused by various events in the system. For example, a user may give a command to start or stop an application, or an embedded resource manager in the system may reduce application quality to maintain real-time guarantees in the case of a structural overload. The application input data may also trigger a mode switch, such as in the case in MPEG-4 decoding, where the number and size of elementary streams that need to be decoded in parallel may vary over time within the incoming transport stream. For MPEG-4 main profile, level 4, the transport stream can contain up to 32 visual objects in different elementary streams [75], eliminating all options for a predetermined worst-case resource allocation.

The configuration changes relatively infrequently with respect to data processing. For MPEG-style applications, the maximum rate of reconfiguration is at the granularity of an MPEG frame. With a frame rate of 25 Hz and coprocessors running at 150 MHz, reconfiguration is performed maximally once every 6 Mcycles, or 60.000 processing steps with an average processing step duration of 100 clock cycles. Reconfiguration can be regarded as *quasi static* from the coprocessor perspective, and therefore can best be implemented in software. Moreover, this

maximum reconfiguration rate will only be reached when the application programmer applies dynamic reconfiguration to switch applications every frame by switching task and stream table values—for instance to compensate a shortage of task and stream table entries in the shell hardware with respect to the desired set of concurrently executing applications (see Chapter 5 for a detailed description of the coprocessor shell configuration). Typically, reconfiguration takes place at a lower rate, either to accommodate application mode changes initiated by the user, or to adapt to structural load changes in the application under control of quality-of-service resource management software.

4.3.1 Synchronized reconfiguration

The graph of concurrently operating tasks is a generalized form of a processing pipeline, where tasks hand off workloads to the next task in line. At any particular point in time, the tasks have reached a different stage in processing the application data. As the state and progress of the overall application is distributed in space and time, the switch from one use case to the next cannot be done for all tasks simultaneously, but must be done in synchronicity with the state and progress of each individual task. The key challenge is to reconfigure the individual tasks while maintaining the real-time behavior and data integrity of the overall set of concurrently executing applications.

Events in the system may require a change in application configuration. A simplistic approach would be to directly reconfigure all tasks, for instance by setting new task execution parameters or modify the application structure of tasks and channels. This results in inconsistency problems with the data stored in the stream buffers. A second somewhat less simplistic approach would be to terminate processing by flushing all stream buffers before reconfiguration. This could lead to undesired behavior as such terminate and restart might violate real-time requirements.

Analyzing overall application progress is of continuous concern in systems with data dependent processing and real-time requirements. In addition, debugging applications on multiprocessor systems with multitasking processors requires the ability to set breakpoints per task. Setting task breakpoints requires comparable measures as needed for intrusion of running tasks for application reconfiguration. Opposed to building an application-specific solution, a generic solution for these problems is required, as part of the multiprocessor system platform.

G. Kahn [80] published a mathematical framework to analyze dataflow networks of tasks connected by unbounded buffered channels. Kahn and McQueen [81] formally proved that dynamic reconfiguration of the structure of tasks and channels triggered through elements in the data streams guarantees unique system behavior. However, Kahn only provided a mathematical model, but no reusable implementation method.

The Eclipse coprocessor shell supports application reconfiguration at a deterministic location in the streaming data, as opposed to reconfiguration at a single point in time. This allows individual task reconfiguration while maintaining the pipelined processing and data integrity. This is achieved through inserting so-called *location IDs* in the communication channels that travel synchronously with the data through the application. A location ID is a number that identifies a specific position in the data stream. The location ID does not contain the graph reconfiguration information itself, but is merely used to trigger reconfiguration by the control processor at a desired point in the application data processing.

Figure 16 depicts how a location ID travels through tasks A, B, and C, and triggers each task to suspend at the same location in the processing of the data stream. Recognizing the location ID is part of the task functionality, and the resulting action takes place at a well-defined point in the data stream. Therefore, application reconfiguration at these points, driven by this mechanism, is a deterministic action and in principle independent from task scheduling issues. As a result, the Eclipse reconfiguration mechanism fits the formal reconfiguration model presented by Kahn and McQueen [81] in 1977.

Figure 16. Suspending tasks on location packets. (a) Normal operation with three data packets in the output buffer of task A. (b) Task A generates a location packet (depicted as triangle) on its output and suspends execution. (c)/(d) Task B and C continue processing data packets. (e) Task B recognizes the location packet on its input, forwards the packet to its output, and suspends execution. (f) Task C recognizes the location packet on its input, forwards the packet to its output, and suspends execution.

4.3.2 Reporting control events

When a coprocessor task encounters a location ID in its input stream, it passes the observed location ID to the shell along with the ID of the currently executing task through the **Report** primitive. The shell contains a programmable field per task to store a location ID of current interest. Upon reception of a location ID from the coprocessor, the shell compares the received ID with the programmed ID for this task. Two special ID values are reserved to match any received location ID or none at all. Upon a match, the shell can suspend further processing of this task, and can send an interrupt to a control processor. The control processor can subsequently analyze and/or reconfigure the local task state under software control. After reconfiguration, the control processor can instruct the shell to resume the task. For a multitasking coprocessor, the coprocessor can continue processing other tasks while reconfiguration takes place.

The **Report** primitive can also be used to reconfigure the task or application on the sole initiative of the coprocessor. Thereto, the `report_type` argument of a coprocessor **Report** request (Section 3.2.6) discerns between location IDs and other events. The shell *always* generates an interrupt to the control processor on events other than location IDs. This way, the coprocessor can signal a request for reconfiguration or report an error that occurred in the coprocessor and requires attention from control software. For instance, a coprocessor may issue an error report when it encounters an unsupported video format while parsing a video stream. Coprocessor-initiated reconfiguration requests are typically caused by information in the input data stream that demands changes in the overall application graph. For instance, a demultiplexer task can report the start of an additional video object stream in an MPEG-4 transport stream, requiring instantiation of an additional decoding task.

4.3.3 Formatting data streams

The shells, offering the 6-primitive interface to the coprocessors, implement transportation of byte-level streams without being concerned about data contents. On the other hand, coprocessors concentrate on operations to perform on this data. Hence, Eclipse instances are free to choose a particular formatting of its data streams. This section proposes a particular data stream format that applies well to video processing systems and is deployed in the video coding subsystem of Chapter 6.

We decided on a uniform structuring of the data streams, to be implemented by all coprocessors. This is a packet formatting of the streams, providing a common way to handle (dynamically) variable-sized data chunks, as well as providing a facility for interleaving different data types, such as media data versus meta-data, and for forwarding information to other tasks further down the application graph. As result, all Eclipse data streams are formatted as a stream of packets, where

each packet takes the form of either a ‘short’ or a ‘long’ style according to Figure 17.

For efficient packetization of the media and meta data, variable length packets are used. A uniform packet header contains information on the packet type and its payload size. The packet format provides a 7-bit header field, next to a 1 to 256-byte data payload. The packet type indicates what the coprocessor should do with the data, such as process it as basic media data, or interpret as meta-data and update the process context state. Furthermore, the coprocessor may not recognize a packet type, upon which the coprocessor forwards the packet to its output ports. This mechanism allows to send control data through the application graph—avoiding overhead in allocating dedicated streams for this purpose—and allows to change context at a synchronized point in the data. One example is passing picture property data embedded in a stream of macro block (or pixel) data.

Figure 17. Data stream packet formats. The first bit of the first header byte on the left discerns between short (above) and long (below) packet types. For long data packets, the second byte gives the size of the variable-length payload.

One particular type of packet to be recognized by all coprocessors is the ‘location’ packet. When the task reads a packet and finds out that its type is ‘location’, it takes the (usually 1-byte) payload value as location identification, and passes this value to the shell through an invocation of the **Report** primitive. When the shell receives a location ID through the Report interface, it compares that value to the contents of a pre-programmed location field in the shell. Upon a match, this task is disabled for further scheduling, and optionally an interrupt is sent out to the control processor.

4.3.4 Forwarding location packets

As tasks are aimed to be generic and reusable, they are unaware of their position in the processing graph. Tasks do not know which other task provides their input or consumes their output. To an even stronger extent, tasks do not know which other tasks are sensitized by application control to a particular location ID that is encountered. In this respect, the location ID is a message “to whom it may concern”. In order to guarantee that *anyone concerned* receives the location ID, the location ID is forwarded to *everyone* in the task graph.

Thus, the intention of location packets is a flow of such packets through the entire application graph, such that precisely once each task will check the location argument value in its shell to allow reconfiguration. The intention is to freeze the implementation of the task functionality, and allow application flexibility by

providing freedom in instantiating these tasks in different application graphs. As recognizing and forwarding location packets is a part of the task functionality, a fixed implementation should be correct for all conceivable graphs. The simplest generic approach, which entails blindly forwarding received location IDs on any input to all outputs, can be problematic if the graph contains cycles, as we do not want such packets to be running around forever in the graph, or possibly causing exponential growth of the number of circulating packets.

In practice, the problem is not as hard as this general statement, as the set of provided functions has limited connection possibilities, due to stream data typing. A major step towards solving the problem is to set a convention on the location functionality.

- A task should be capable of detecting location packets on any of its input streams.
- Only from one particular port, called the *sensitive* input port, it should process such packets, leading to a location **Report** call to the shell, and copying of the packet to each output stream.
- On the other input (*insensitive*) ports such packets are dropped without interpretation.

Figure 18 depicts this setup. When used with appropriate care, this organization can avoid circulating location packets around forever in any application graph structure. If the sensitive input port (p_2 in Figure 18) is not used to connect feedback edges, then each task in the graph recognizes location IDs exactly once. The selection of this ‘special’ task input depends in principle on the structure of the task graph of the application, and which task inserts the location ID into the stream. Therefore, the selection of the sensitive port should be made task-instance and location-source dependent. An appropriate mechanism to configure the selection of the sensitive port is to program task information for each task in the shell, information that is passed by the shell to the coprocessor when the task is activated. In particular cases where the coprocessor implements a function with very different port characteristics, it might be possible to fix the sensitive input to a particular port at silicon design time.

Figure 18. Dropping and copying location packets.

5 Eclipse generic infrastructure

Eclipse provides a generic infrastructure used by all coprocessors. The Eclipse infrastructure centers around the coprocessors shells that maintain the application graph structure and implement a large part of the generic functionality, including stream synchronization, data transport, task scheduling, performance measurement, and dynamic reconfiguration support. This chapter describes how we implement the response to each of the six Eclipse primitives in the coprocessor shell as part of the Eclipse infrastructure, independent of the coprocessor implementation outlined in Chapter 4.⁵

Figure 19. Internal structure and interfaces of the coprocessor shell template. The arrows indicate the direction in which actions are initiated. Four units handle the different requests from the coprocessor (at the top), and if necessary issue requests on the Eclipse interconnect (at the bottom). The DTR and DTW handle coprocessor I/O requests, the SS maintains the administration of buffer space for all relevant data streams, and the TS selects the appropriate tasks to execute on the coprocessor and handles control events from the coprocessor.

Figure 19 depicts the internal structure of the coprocessor shell and its interfaces. The shell separates its functionality in four different modules, each with a distinct responsibility. This strict separation of responsibilities aids in having a parameterized shell template, up to the extent where complete modules are included/excluded in a specific instance of the shell template. This chapter gives a high-level description of the functionality implemented in these modules. A more detailed specification of each module is given by Rutten [120].

The data transporters (DTR and DTW) in Figure 19 handle read and write access to stream buffers in shared memory and implement the **Read** and **Write** primi-

⁵ This chapter is based on Rutten et al. [121] and patent [41]. Section 5.2 is based on [127] and patents [37][38][41]. Section 5.3 is based on [125] and patent [122]. Finally, Section 5.5 is based on patent [126].

tives. The stream synchronizer (SS) takes care of stream synchronization and implements the **GetSpace** and **PutSpace** primitives. Finally, the task scheduler (TS) handles task selection and (re)configuration by implementing the **GetTask** and **Report** primitives.

In addition to the interfaces depicted in Figure 19, the coprocessor shell connects as slave to a peripheral interconnect bus for configuration management. The synchronization network is implemented as a dedicated token-ring structure. The connection to the data transport network consists of a read DTL interface and a write DTL interface [113]. Each interface can be hooked up to a separate bus, or both interfaces can be hooked up to a single bus. The particular protocol (such as DTL, AHB, AXI) to connect the shell to shared memory may vary with the different instances of the Eclipse template. The shell hides the chosen bus protocol from the coprocessor. In the Eclipse project we focused on the Philips proprietary DTL protocol, which is the default interface in the Nexperia-DVP platform.

5.1 Stream synchronization

Communicating a stream of data requires a FIFO buffer with a finite and constant size preallocated in shared on-chip memory. The shell applies a cyclic addressing mechanism for proper FIFO behavior in the linear memory address range. For this cyclic addressing mechanism, the shell uses the buffer size and current access point, as maintained in the shell, and the size and offset arguments of the **Read/Write** requests.

Figure 20. Basic stream mapped to a finite FIFO buffer. Task B consumes data written by task A. Task A consumes empty room (or meaningless rubbish) left behind by B. The arrows denote the current access points of task A and B as stored in the shell.

Figure 20 depicts the fixed-size cyclic memory space used as a FIFO buffer. The rotation arrow in the center shows the direction in which **GetSpace** calls confirm the granted window for **Read/Write**, which is the same direction in which **PutSpace** calls move the access points ahead. The small arrows denote the current access points of tasks A and B. In this example, A is a producer and hence leaves proper data behind, whereas B is a consumer and leaves empty space (already consumed data) behind. The lightly shaded region ahead of each access point represents the access window acquired through **GetSpace**.

5.1.1 Local stream administration

Each shell locally contains the configuration data for streams incident to tasks mapped on its coprocessor and locally implements all the control logic necessary to properly handle this configuration data. The shells implement a local *stream table*, which contains a row of fields for each stream, or more precisely, for each access point. To handle the setup shown in Figure 20, the coprocessor shells of tasks *A* and *B* each contain one stream table row, holding among others the following fields:

- A *space* field containing a (maybe pessimistic) distance from its own access point to the other access point in this buffer. The space value corresponds to the amount of available data for reading from an input port or the available room for writing to an output port.
- A *stream identifier* denoting the remote shell with the task and port of the other access point in this buffer.

Figure 21 shows that the shell of coprocessor *B* can answer a **GetSpace** request immediately by comparing the requested *g* bytes with the locally stored space value. When the shell of coprocessor *A* receives a **PutSpace** request, it locally decrements its space field with the indicated *p* bytes and sends a ‘putspace’ message to the shell of coprocessor *B*. This remote shell holds the other access point and increments its space field upon receiving the ‘putspace’ message.

Figure 21. Updating local space administration and sending ‘putspace’ messages.

As illustrated in Figure 21, the shells update the *space* field for a stream as follows:

$$space_t = \sum_i m_i - \sum_i p_{local,i} + initial_space, \quad (6)$$

with, *m* the space received by incoming ‘putspace’ messages, and *p_{local}* the space committed by local **PutSpace** calls. In essence, the available space at time *t* is the difference between all previous local and remote commit actions on the stream. The shell uses a single hardware implementation for producers and consumers on the stream. Control software initializes the *initial_space* value to the buffer size for producer and to zero for consumer tasks.

The shell ensures the following response to **GetSpace** requests from the local coprocessor:

$$granted_space_i \leq space_i . \quad (7)$$

On each local **PutSpace** request, the shell emits a ‘putspace’ message on the synchronization network. The updates to the space field in the receiving shell may lag behind due to the messaging delay in the synchronization network:

$$\sum_i m_i = \sum_i p_{remote,i} - \sum_i f_i , \quad (8)$$

with p_{remote} the space committed by the remote coprocessor and f the messages that are still in-flight on the synchronization network.

The synchronization mechanism implemented in the coprocessors (1-5) and the implementation in the shell (6, 7) and synchronization network (8) ensures that the access windows of A and B in Figure 20, never overlap. Neither the messaging delay, nor potential out-of-order servicing of synchronization messages in the network affects functional correctness. The combination of (6) for tasks A and B with (8) shows that the total space on a stream is bounded and therefore permits a circular buffer implementation. Due to messaging delay in the synchronization network, the space values of shells A and B may not always sum up to the entire buffer size, but may momentarily contain a pessimistic value:

$$space_{A,i} + space_{B,i} = buffer_size - \left(\sum_i f_{A,i} + \sum_i f_{B,i} \right) . \quad (9)$$

The distributed synchronization mechanism allows the shell to immediately and locally answer all coprocessor **GetSpace** and **PutSpace** requests, while sending and servicing synchronization messages between shells in the background. Note that different performance/cost trade-offs—for instance a lookup into centralized buffer administration on every **GetSpace** request, as described by Nieuwland et al. [102]—are possible without affecting the coprocessor-shell interface.

5.1.2 Synchronizing multicast streams

The Eclipse shell has provisions for synchronizing multicast streams, as described in Section 4.1.2. Figure 22 depicts the synchronization messages sent between producer A and consumers B and C . For clarity, we separate **PutSpace** requests into `put_room` for signaling available room in the buffer, and `put_data` for signaling data availability. The figure shows that multicast only affects the synchronization of the producer.

Figure 23 depicts the cyclic buffer synchronization. The synchronization mechanism ensures a normal pair-wise ordering between A and B , next to a pair-wise ordering between A and C , while B and C have no mutual constraints. A multicast stream is programmed into the producer’s shell as a linked list of stream table en-

tries, one entry for each consumer. This way, the producer shell keeps track of available space separately for each consumer on the multicast stream. To keep this fully transparent to the producer task, the shell issues `put_space` messages to all consumers on a single `PutSpace` command from the producer. Likewise, a producer's `GetSpace` command checks the available space in the stream buffer with respect to all consumer stream entries. This implementation provides very little overhead for the majority of cases where multicast is not used, and at the same time does not limit multicast to 2-way only.

Figure 22. Synchronization messages for a multicast stream. Task A commits produced data to both task B and C. Both task B and C commit consumed data towards task A.

Figure 23. Cyclic buffer with a single producer A, and multiple consumers B and C. The arrows denote the current access points of task A, B, and C. Tasks B and C consume valid data produced by task A and leave behind empty space (or meaningless rubbish). The access point of task A may not go beyond the access points of B or C to ensure functional correctness.

5.2 Data transport

Coprocessors transport all media data to and from their shells through `Read` and `Write` operations. The shells internally compute the actual address into a cyclic stream buffer in shared memory and access the data:

$$write_address_t = \left(\sum_i p_{producer,t} + write_offset_t \right) \bmod buffer\ size, \quad (10)$$

$$read_address_t = \left(\sum_i p_{consumer,t} + read_offset_t \right) \bmod buffer\ size, \quad (11)$$

with p the space committed by local `PutSpace` calls. To simplify the modulo calculation in hardware, the buffer size is restricted to a power of two and the shell stores the logarithm of the buffer size instead of the actual size.

The shells provide the **Read** and **Write** interface to hide aspects such as the width of system data paths; data alignment in memory and cyclic buffer addressing, and data stream caching—including coherency and prefetching control. Shells incorporate caches that play an important role in decoupling the coprocessor from the global communication network.

The main responsibility of the caches in the shell is to transform unaligned access by the coprocessor to aligned memory transfers, hide data access latency, and reduce the number of accesses to memory. The following subsections detail the three most important and novel concepts applied in these caches. Impartial to the described caching techniques, we chose to separate the read and write data path to more easily support parallel read and write requests, for instance from a pipelined coprocessor. The remainder of this section focuses on read accesses to the read cache, as these are the most challenging with respect to cache coherency and prefetching. However, the proposed techniques apply equally well to handling write requests, or an alternative implementation with shared read/write caches.

5.2.1 Cache indexing through stream information

With the advent of media processing, caching techniques optimized for streaming data have received an increased attention in the form of stream buffers [79], stride-prediction tables [23][50], and stream caches [154]. Oftentimes, these techniques are applied to traditional (multi-way) associative caches. The canonical form for selecting victims in a fully loaded associative cache is the least-recently used (LRU) mechanism [67]. Such victimize strategies are oblivious to the stream associated with cached data, causing cache contention when the processor accesses multiple data streams through its cache. To some extent, techniques have been devised to avoid contention by extracting stream information from the processor's access pattern and separate accesses to buffered streams into cache blocks accordingly. This section outlines a more cost-effective cache organization that exploits the processor's knowledge of the application structure.

Stream buffers in shared memory compete for shared resources, such as cache lines and a limited number of banks to store address tags. The coprocessor tasks are I/O intensive, requiring efficient cache behavior. Thus, contention on the cache resources leads to large and unpredictable task execution delays. For each **Read** or **Write** access, the coprocessor tasks pass a `task_id` and `port_id`. The shell internally computes the actual address into a cyclic stream buffer in shared memory. Note that the `port_id` has local scope for each task. The shell combines the `task_id` and `port_id` to form a stream identifier. To limit cache contention, the shell indexes its read or write caches through this stream ID, effectively decoupling the caching of stream content of different streams.

The stream ID can be used to select a row of cache blocks. However, we chose to share cache rows over different tasks to limit the cost of cache memory in the

shell. Thus, the shell only uses the `port_id` to select a cache row, and cache rows are shared over equivalent `port_id`'s of different tasks. Moreover, instead of directly addressing of the cache row by the `port_id`, the shell applies a hashing function by which it translates the port ID into a smaller number of cache rows. Figure 24 depicts this cache organization. For the implementation in the shell, we chose the hashing function to be a simple modulo operation over the number of rows. This way, a single task may share a single cache row over multiple task ports. This is cost-effective when for instance all media data is read through the first task port, and the task only occasionally reads some meta data from its second task port. Sharing the cache row then avoids the hardware cost of a full row of cache locations for the second task port.

Figure 24. Addressing cache locations through `port_id` and `address`. A subset of the address bits is used to address a column in the cache. The port ID of the read request is used to lookup a row index in the cache. The cache returns the data word at the selected column and row index.

Figure 24 depicts a direct-mapped cache organization. This means that every port ID and address combination maps to a single cache location. Within this row, a cache block is indexed through the lower bits of the I/O address. Thereto, the number of cache blocks in the row is restricted to a power of two. This results in a simple and cost-effective cache implementation in the coprocessor shell. Clearly, such a scheme can be extended to more general set-associative cache organizations, where the stream ID selects a cache row and the lower bits of the address select a set of cache blocks. The actual data word is then further located through tag matching on the address.

5.2.2 Cache coherency through explicit synchronization

One of the notoriously difficult problems in the field of cache technology for inter-processor communication is cache coherency [27]. In any multiprocessor system that deploys caches to access shared memory, cache coherency must be enforced to ensure that each processor reads properly updated data values from shared memory. When an Eclipse coprocessor reads data from a stream buffer through its shell's private cache, the Eclipse needs to ensure validity of the read data.

A major responsibility of the caches in the coprocessor shell is to hide the global interconnect data transfer size and data transfer alignment restrictions from the coprocessor. The inquiry/commit synchronization scheme that Eclipse coprocessors use to exchange information on the delivery or consumption of data operates at byte granularity. As the shell typically accesses the bus at a larger granularity than a single byte, the same memory word may be stored simultaneously in the caches of different coprocessor shells. Thus, invalidate and dirty information must be handled in each cache at byte granularity.

For processing streaming data, several groups work on processors with special stream cache architectures to improve the data transport to/from memory. Eclipse combines the—traditionally separate—domains of stream caching and multiprocessor cache coherency. This section shows how the shells apply the Eclipse synchronization scheme to control cache coherency and prefetching, fully transparent to the coprocessor tasks. This results in a simpler and more efficient implementation than generic coherency mechanisms such as bus snooping [27]. In addition, it avoids the communication overhead of for instance a write-through architecture [79]. The shell's cache coherency mechanism builds on three key observations:

1. The access window on stream data, which is granted to a task port by a successful **GetSpace** action, is guaranteed to be private. Thus, **Read/Write** operations in this area are safe and do not require immediate intra-processor communication.
2. Additional **GetSpace** requests extend the access window, obtaining new memory space from a predecessor in the cyclic buffer. Data in the cache that corresponds to this new memory space possibly needs invalidation. A subsequent **Read** action on such a cache location then results in a cache miss, upon which the cache loads fresh valid data from the cyclic buffer.
3. Local **PutSpace** requests reduce the access window, leaving new memory space to a successor in the cyclic buffer. Dirty data in the cache that corresponds to the memory space in the reduction interval needs to be flushed to the cyclic buffer to make the local data available for other processors. Sending the 'putspace message' to another processor must be postponed until the cache flush is completed and safe ordering of memory operations can be guaranteed.

Figure 25 depicts the fixed-size cyclic memory space used as communication buffer. The rotation arrow in the center shows the direction in which a producing task *A* and a consuming task *B* move their access points ahead with each **PutSpace** action. A **PutSpace** action by the producer reduces the access window on empty room in the buffer, while the producer extends this access window with a successful **GetSpace** action. Equivalently, the consumer extends its access window on valid data through **GetSpace** actions, and reduces the access window by committing already consumed data.

The inner circle in Figure 25 depicts the memory words in the buffer. The producer moves its access point (its *write pointer*) from A1 to A2 by committing newly written data. The consumer may subsequently extend its access window into this new data range. To ensure cache coherency, the producer's write cache must have flushed memory words 4, 5, and 6 to memory, and the consumer's read cache must invalidate cached memory words with these same addresses.

Figure 25. Basic stream mapped to a finite cyclic buffer. Task B consumes the valid data left behind by task A. The inner circle shows a mapping of the byte-level data stream to memory words. Task A moves its write pointer from A1 to A2 when it commits newly written data, allowing B to read beyond A1. When B reads beyond A1, it must invalidate words 4-6 in its cache, as these are newly written by A. Moreover, task A must flush memory words 4-6 from its cache when it moves its access point from A1 to A2 to ensure that B reads valid data from memory when it reads beyond A1.

Generalizing the situation of Figure 25 leads to the following implementation. On a **PutSpace** action, the producer's write cache flushes all cached data words whose tag addresses overlap with the address range from the producer's previous write pointer to its updated write pointer after the commit action. On a consumer's **GetSpace** action, the read cache invalidates all cached data words whose tag addresses lie between the current write pointer and the write pointer at the last inquiry action. The consumer's shell computes the write pointers from its local buffer administration. Clearly, discrepancies in buffer administration between the producer and consumer shells due to synchronization messaging delays do not affect functional correctness.

5.2.3 Prefetching on dismissed cache locations

Processors generally apply caches to reduce the latency to access data in memory. For streaming applications, prefetch methods are deployed to predict upcoming I/O operations and further reduce memory access latency [154]. State of the art prefetch techniques address two problems. First, access to a stream must be recognized by matching the addresses of a series of I/O operations and extrapolate this to an expected future I/O access. This recognition of the stream access pattern is troubled by interleaved I/O operations that do not belong to the stream. Second, once a prediction is found that is not available in the cache, the corre-

sponding block of data needs to be fetched from memory. When the prefetch data enters the cache, it may replace data that is still valuable. Therefore, prefetching may be the cause of further cache misses.

Literature on prefetch methods focuses on solving the first problem of predicting future I/O operations [23][50]. Little attention has been given to the second problem of selecting cached data to be replaced. This section proposes a technique that reverses the traditional prefetching approach by predicting cache locations for which the cached data is not expected to be further used. The cache subsequently initiates prefetch actions to fill precisely these cache locations—without overwriting valuable cached data.

The Eclipse coprocessor shell implements this approach by carefully selecting when to execute a prefetch. Instead of predicting future I/O operations, the shell predicts dismissing of cached data that is not expected to be further used. Subsequently, it fetches data to replace the dismissed data in the cache. Thereby, the shell reduces the risk of overwriting cached data that is still needed in the cache.

Prefetching in the coprocessor shell is initiated by read and invalidate requests from the coprocessor. Apart from sporadic random access within the acquired window of valid data—through usage of the offset parameter in `Read` requests—Eclipse coprocessors are expected to access data in a streaming fashion. Thus, we assume that subsequent reads belonging to the same stream address a contiguous range in memory in linear order. This streaming behavior of Eclipse coprocessors allows the shell to cost-effectively embed prefetching caches.

If a read action within a stream buffer accesses the last data word in a cache block, the shell assumes that all data of the block has been read and can be dismissed. At this event, the shell prefetches a data block from a new location in memory that fits to the cache location of the dismissed cached data. The shell prefetches the next higher address from the address of the dismissed data that fits the cache location. The direct-mapped cache organization ensures that the prefetched data will be stored at the location of the dismissed data. As stream accesses occur in linear order, the stream is expected to access the prefetched data in the near future.

Invalidate requests on cached data to control cache coherency are triggered by coprocessor `GetSpace` requests. These invalidate events mark locations in the cache to be considered as empty. Invalidates are caused by a task that produces new data on the stream. By calling `GetSpace`, the reading coprocessor indicates that it expects to access this new data in the near future. Therefore, the Eclipse coprocessor shell issues a prefetch for those cache locations marked invalid.

Additionally, the shell prefetches new data for all cache locations within the selected cache row that fall outside the range of valid data in the stream buffer. The latter is of special importance to reduce the latency of updating a cache row immediately after a task switch in case that cache rows are shared between stream buffers of different tasks. For these dismissed cache locations, the shell prefetches

from the memory addresses that fit the dismissed cache location and are closest to the current point of access.

The write cache applies a similar strategy by flushing cached data as soon as it predicts that this data will not be accessed anymore. The write cache initiates such a *preflush* when a write access moves to a next word. With a streaming write behavior, the previous word will not be further accessed, and its cache location can be made available for expected future.

5.3 Task scheduling

The shell's task scheduler decides which task a coprocessor must execute and the times it must execute the task to attain proper application progress. The target granularity for processing steps in the Eclipse architecture ranges from 10 to 1,000 clock cycles. Typically, a processing step's duration is data dependent and can vary within this range. The number of processing steps needed to complete an application milestone (such as an MPEG frame), as well as the number of produced and consumed data items per processing step, may also be data dependent. Consequently, Eclipse must perform task scheduling at runtime to manage the highly data-dependent workloads cost-effectively.

Figure 26. Task scheduling algorithm. On a *GetTask* request from the coprocessor, the scheduler checks if the currently active task still has sufficient budget (**active_budget** > 0), is not disabled by control software (**enabled**) and has more than zero buffer space (**runnable**) on all relevant data streams. If the answers are positive, the coprocessor may continue executing the same task. If not, the scheduler walks in round-robin fashion over all configured tasks. It selects the first enabled (by control software) task that has more than zero buffer space on all relevant data streams and sets the active budget to the task's budget value configured by control software (**active_budget** = **budget[task_id]**). The active budget decrements with a fixed clock to keep track of the task's execution time.

The task-switching rate is too high (10 to 100 kHz) for runtime scheduling in software. Therefore, Eclipse implements task scheduling and synchronization in dedicated hardware in the coprocessor shell. Moreover, task scheduling is distributed to contribute to the coprocessors' autonomy, thereby increasing scalability and cost-effectiveness. Thus, the task scheduler in each shell runs independently from task schedulers in other shells. We chose a weighted round-robin scheme to select tasks because it ensures fairness and can be efficiently implemented in the generic shell hardware. Figure 26 gives a high-level overview of the task selection algorithm.

To improve the system's robustness, we endowed each task with its own *task budget*—a guaranteed minimum number of cycles that a task may continuously execute, irrespective of the other tasks' resource requirements. A task may continue to execute until it depletes its budget—implemented in Figure 26 as the decreasing `active_budget` cycle count that starts from the configured budget value when the task is first selected. The tasks mapped to the coprocessor are configured in the shell's *task table*, which contains among other things the resource budget per task.

Due to the data-dependent behavior of Eclipse tasks, the task scheduler cannot determine in advance whether a task can complete a processing step. Therefore, the scheduler performs a best guess by considering the available data and room in the stream buffers, as well as previously denied data access. This is depicted as the 'runnable' condition in Figure 26. The required information on the status of the stream buffers is locally available in the shell. The runnability check allows the task to select the right tasks in most cases and recover with a limited penalty otherwise. Section 5.3.2 further details the runnability condition.

5.3.1 Robust resource sharing through task budgets

The requirements for Eclipse are derived from the characteristics of the application domain of high-end media processing. This implies many applications running concurrently, each with a high degree of irregularity. In practice, the ratio of worst-case versus average-case load can be a factor of 10 or higher. Under these circumstances, architecting the system for worst-case scenarios is not competitive.

The direct consequence of running more applications than that the system can handle in worst case situations is that the system must be able to handle temporary overload situations in a robust way. Eclipse applications are configured by creating appropriate tasks on multi-tasking coprocessors. Robustness is implemented by assigning each task a guaranteed minimum of compute resources, which we term as a *budget contract* [14]. The budget is set at some level between average and worst-case task resource requirements, which means that usually the task has sufficient resources available and only sometimes not. The contract guar-

antees that each task can use the resources that are assigned to it by its contract, independent of the possible excess resource requirements of other tasks. Thus, system robustness is implemented by endowing the system a separation of concerns with respect to resource assignment per task. This separation of concerns is termed *justice*.

Usually, temporary overload occurs only for one or few applications concurrently, while others remain at or below average. Per task budgets are assigned somewhere between average and worst case, which means that in the average case not all hardware resources are used. If, in spite of the fact that some tasks have excess resource needs, the total load on the system still falls within the available resources, then reallocating surplus resources to tasks in need raises the cost-effectiveness of the system. Note that this reallocation is done only if the resources would remain otherwise unused.

Assignment of task computation budgets is done at system level, based on resource requirements and relative importance of each application with respect to the current system load [14][99]. Every application undergoes an acceptance test to verify that the cumulative application resource requirements do not exceed the system capacity. After acceptance, the system assigns budgets to the application. These system-level application budgets are translated into budgets per task for use in the ‘operating system’ defined by each Eclipse shell.

The shell must support a *policing* strategy to ensure protection of task budgets, such that a higher authority can implement justice. This responsibility of the Eclipse shells corresponds directly to the US police adage ‘to serve and protect’. Task budget protection is a shared responsibility between a coprocessor and its shell. In order to guarantee non-starvation, the length of the processing steps of each task mapped on the coprocessor must be bounded. For instance, as the coprocessor may perform a busy wait on a `GetSpace` call, the coprocessor must implement a time-out period after which the coprocessor performs a state save and ends the processing step (Section 4.2.1).

Task budgets are specified in units of processing time. Assignment of such task budgets has two opposing aspects that need to be balanced:

- In the acceptance test, the system must consider the additional resource requirements of non-preemptive task scheduling in the Eclipse shells; whenever a task depletes its budget, the task overruns this budget by the remaining duration of its processing step. The relative overhead of this budget overrun can be minimized by assigning a high budget value in relation to the (worst-case) duration of a processing step.
- The absolute budgets of tasks in a coprocessor determine the running time of these tasks, and therefore the task switch rate of the coprocessor. In turn, the task switch rate of the coprocessor relates to the buffer sizes for all its streams. A lower task switch rate means a longer sleep time for tasks, leading to larger buffer requirements and latency.

Eclipse task switch rates are fairly high, in the order of 10-100 kHz. Task budgets typically range from 1000 up to 10,000 clock cycles (10—100 processing steps). Assigning budget values that are relatively high with respect to processing step duration makes the system behave more like a preemptive scheduled system.

5.3.2 Addressing dynamic workload through task runnability

Eclipse streaming media tasks have a dynamic workload. They can be data dependent in execution time, stream selection, and/or packet size. This data dependency influences the design of the scheduler, as it cannot determine in advance whether a task can make progress or not. The aim of the scheduler is to improve the utilization of coprocessors, and schedule such that tasks can make as much progress as possible. Due to the data dependent operation of the tasks, it cannot guarantee that a selected task can complete a processing step.

Data dependent packet size

Task runnability is based on the available workload for the task; all streams associated with a task should have sufficient input data or output room to allow the completion of at least one processing step. Data packet sizes may vary per task and packet size can be data dependent. The shell, including the task scheduler, does not interpret the media data and has no notion of data packets. Therefore, the scheduler does not have sufficient information to guarantee success on `GetSpace` actions since it has no notion of how much space (data or room) the task is going to request on which stream.

The scheduler issues a ‘best guess’ by selecting tasks with at least some available workload for all associated streams, regardless of how much space is available or required for task execution, *i.e.* $space > 0$, with the *space* parameter holding the available data or room in the stream, updated at run-time via the `PutSpace` primitive. Checking if there is some data or room available in the buffer—regardless of the amount—suffices for the completion of a single processing step in the cases that:

- The consuming task synchronizes at an equal or lower grain size than the producing task. Therefore, if data is available, this is at least the amount of data that is necessary for the execution of one processing step.
- The consuming and producing tasks work on the same logical unit of operation, *i.e.* the same granularity of processing steps. For instance, if there is some but insufficient data in the buffer, this indicates that the producing task is currently active and that the missing data will arrive fast enough to allow the consuming task to wait instead of performing a task switch.

In current practice, including the MPEG-2 decoding and encoding applications, all coprocessor communication is covered by the above two cases.

Data dependent stream selection

If there is at least some available space in all streams of the task, then a task is runnable. The selection of input or output streams however can depend on the data being processed. This means that even if $space = 0$ for some of the streams associated with a task, the task may still be runnable if it does not access these streams in the upcoming processing step. In the task runnability test, the scheduler simply does not check the space value for streams for which it is unclear whether or not the task is going to access the data. This information is stored in the ‘schedule’ flag in the stream table. The space value of a stream only affects task runnability if the stream’s schedule flag is set.

Blocking on communication

Association of streams to a task is programmed through entries in the stream table. To assess task runnability, the stream synchronizer (SS in Figure 19) computes runnability of each stream on each **GetSpace** or **PutSpace** from the coprocessor and on accepted ‘putspace’ messages from the synchronization network. A task j is runnable if all streams i that belong to the task are runnable:

$$runnable_j = enable_j \wedge \prod_i [(space_{ij} > 0) \wedge \overline{schedule_{ij}}] \wedge \overline{blocked_{ij}}. \quad (12)$$

If the task is runnable and is configured to be active (‘enabled’ in Figure 26), it can be selected for execution on the coprocessor.

Task runnability also helps the task scheduler to select tasks that can progress in the case that coprocessor stream I/O selection or packet size is data dependent and cannot be predicted by the scheduler. If a task cannot make progress due to insufficient space, the **GetSpace** inquiry on one of its streams must have returned false. In this case, the stream is ‘blocked’. This information is stored in the ‘blocked’ flag in the stream table, as shown in (12). The blocked flag is taken into account in assessing task runnability to ensure that a task is not selected in a next scheduling round until its streams have sufficient data. The shell clears blocked flag of a stream when the it receives new space via an incoming ‘putspace’ message. Note that after a failing **GetSpace** request, the active task can also thereby reset the blocked flag by issuing a second **GetSpace** inquiry for a smaller number of bytes.⁶

Thus, on a **GetSpace** inquiry, the shell sets the blocked flag of the indicated stream to the inverse of the **GetSpace** return value. Additionally, the shell resets

⁶ Explicitly resetting the blocked flag by performing a **GetSpace** with a *zero* size argument after a failing **GetSpace** may lead to undeterministic behavior. If the coprocessor commits data (on a second stream) after a failing **GetSpace**, the ordering of data packets in the data streams may depend on the order in which the producing and consuming are scheduled. This is for instance the case when a coprocessor merges two input streams into a single output stream, and continues with the second input stream if a **GetSpace** inquiry on the first stream fails. Such use of the **GetSpace** primitive opposes the deterministic property of Kahn Process Networks and is therefore strongly discouraged.

the blocked flag when an external ‘putspace’ increases the space for the blocked stream. This blocked flag therefore improves the scheduling accuracy, in the sense that it avoids repeated activation of a blocked task.

5.4 Performance measurement support

Eclipse supports performance measurement (profiling) through hardware in the shells. Measurements include buffer filling, coprocessor utilization, and data access latency. Product engineers can use these measurements to optimize application behavior, and quality-of-service resource management software [14][99] can use these measurements to provide runtime control in the final product.

The shells accumulate measurement data in the stream and task tables. The shell collects measurement results per stream and per task rather than per coprocessor port and per coprocessor. Therefore, measurement results are available at the application level—that is, for each task and stream rather than for each coprocessor. All shell tables are memory mapped and accessible to a system-level CPU or an Eclipse-level DSP via a control (peripheral interface) bus. Thus, a configuration manager (configuration management software) can collect measurement data at regular time intervals—for example, once per MPEG frame. However, accumulating measurements every cycle for a complete MPEG frame requires a significant amount of memory in the shell. To reduce the hardware costs of measurement support, a separate process in the shell takes measurement samples at regular intervals. Because the storage space for measurements is limited, the configuration manager must balance the duration of these intervals with the duration of the total measurement.

5.5 Dynamic reconfiguration support

Eclipse supports dynamic (re)configuration of application graphs by allocating buffer space in shared memory and programming stream and task table entries in the shells. Thereto, virtually all internal registers of the shells are visible for read and write access to any control processor connected to the control bus. We developed a low-level control software API for configuring applications on the control processor. The underlying control software translates the generic API calls into read and write requests on the shell registers.

We present the API through the reconfiguration example of Figure 27, in which task B is removed from the task graph and task C is subsequently connected to task A. Removing a task from a running application is a typical example of quality of service resource management, where task B performs a quality enhancement function such as dynamic noise reduction. Removing task B dynamically reduces memory and coprocessor resource requirements at the cost of a decrease in output

quality. Task B is removed at the end of a frame to avoid an abrupt change in output quality during the display of the previous frame. Note that in the above example, the tasks that directly connect to task B must be suspended before changing the topology of the application. Clearly, when the application topology does not change—for instance when reconfiguration only affects task B’s functionality—tasks A and C can continue during the reconfiguration of task B.

In contrast, Webb et al. [147] describe a mechanism of removing a task without suspending the connected producer and consumer tasks, effectively by interrupting the synchronization of data exchange between the tasks. In Eclipse, the granularity of execution is relatively short with respect to duration of a reconfiguration action. Hence, the latency induced by suspending producer and consumer tasks does not outweigh the complexity of keeping these tasks alive during the reconfiguration process.

Figure 27. Dynamically removing task B and buffer X from the application graph in (a) and reconnecting buffer Y to task C as shown in (b).

Figure 28 gives the control code for the reconfiguration action of Figure 27, using a subset of the control API primitives. For a detailed description of the control API, see Walters [148]. These primitives are designed to hide architecture details from the application programmer, yet allowing full control of all capabilities of the architecture. The interface is designed in such way that it does not call its own primitives; each primitive implements independent functionality. High-level programming interfaces and control software—for instance for quality of service resource management or resolving artificial deadlocks by dynamically increasing buffer sizes [8]—can be built on top of this API. The API is implemented in ANSI C to facilitate compilation on low-cost DSP processors.

In the pseudo code of Figure 28, tasks A, B, and C each execute on a different coprocessor, although the above code is insensitive to such a mapping (the creation of tasks on coprocessors is not part of the example). The sample code dynamically removes task B at a logical point during the processing of the data streams, in this case at the end of an MPEG frame. Note that while these tasks are temporarily suspended, other tasks can continue to execute on the coprocessors.

```

// Suspend tasks A, B, and C when each recognizes the
// END_OF_FRAME location ID in their input data stream.
unsigned int location_id = END_OF_FRAME;

// Program the location in the shell of coprocessors A, B, C.
TaskSuspendLocation(task_a, location_id);
TaskSuspendLocation(task_b, location_id);
TaskSuspendLocation(task_c, location_id);
// Program task A to generate the location ID.
TaskGenerateLocation(task_a, location_id);

// Wait for a location match interrupt from task B.
TaskWaitLocation(task_b);

// Disconnect task A and B from channel X and delete channel X:
// - Free the corresponding stream administration
//   in the shells of coprocessors A and B;
// - Free the allocated buffer for channel X in shared memory.
Disconnect(task_a_p1, channel_x);
Disconnect(task_b_p0, channel_x);
ChannelDelete(channel_x);

// Wait for a location match interrupt from task C.
TaskWaitLocation(task_c);

// Disconnect task B from channel Y and delete task B:
// free the task B admin in the shell of coprocessor B.
Disconnect(task_b_p1, channel_y);
TaskDelete(task_b);

// Connect task A to channel Y:
// write a new stream administration in the shells of
// coprocessors A and C.
// Note that channel Y is empty: tasks B and C stopped at the
// same location.
Connect(task_a_p1, channel_y);

// Restart tasks A and C:
// - Optionally remove the programmed location ID
//   in tasks A & C;
// - Enable tasks A and C.
TaskResume(task_a);
TaskResume(task_c);

```

Figure 28. Control software to remove task B from a pipeline of three tasks and reconnect tasks A and C.

6 Eclipse coprocessor design

The Eclipse template furnishes many system-level services that normally burden coprocessor design. Nevertheless, designing reusable yet cost-effective coprocessors for an Eclipse set of applications is a challenging task. For example, the designer must not only identify the DCT as a reusable function but also carefully tune the data granularity (e.g. blocks, macroblocks, frames) on which a DCT coprocessor operates with respect to the desired synchronization rate and size of communication buffers.

Clearly, choosing fine granularity for a coprocessor function (e.g. to compute a sum of absolute differences in MPEG motion estimation) facilitates reuse. However, the overhead of the infrastructure (e.g. to transport and synchronize data access) becomes relatively large. Therefore, fine-grain hardware acceleration is sometimes embedded in RISC [9] or VLIW cores [133] in the form of complex function units. The Molen project [144][133] worked this concept out for MPEG encoding/decoding functions. However, the overall performance increase and reduction in power consumption with such compile-time scheduled function units is limited. A coarse-grain coprocessor such as an MPEG-2 decoder [32] on the other hand incurs significantly less overhead, but is difficult if not impossible to reuse in similar applications.

Balancing coprocessor reuse versus overhead incurred in a generic infrastructure demands a thorough understanding of the application domain and architecture at hand. Oftentimes, application information is only available as complex and ill-documented C-code, e.g. obtained as reference implementation from a standardization body. Extracting reusable computation kernels eligible for coprocessor implementation requires extensive restructuring of the application. This software development trajectory of mapping an application onto a hardware architecture requires a considerable amount of software engineering effort and calls for a structured approach.

Section 6.1 of this chapter presents a design trajectory to transform sequential application C-code into a reusable multi-threaded version.⁷ The multi-threaded code is subsequently mapped onto a target architecture with maximal source code reuse. The design trajectory concurrently addresses a set of applications to identify common compute kernels that qualify for generic coprocessor implementation. Thereby, the trajectory results in the definition of a set of multitasking coprocessors that can be applied to various applications within a chosen application do-

⁷ This chapter is based on Van Balen and Rutten [7] and Rutten et al. [128].

main. In Sections 6.2 and 6.3 we apply this design trajectory to define MPEG coprocessors from sequential C-code for MPEG-2 encoding and decoding.

6.1 Design of reusable Eclipse coprocessors

Eclipse adopts Kahn process networks [80][85] as the model of computation. Ptolemy [19][88] provides an integrated framework for specification and analysis of, among others, applications modeled as Kahn process network. Modeling languages such as SystemC [60] allow a gradual transformation from sequential code and Kahn-style models towards a hardware implementation. However, how to do this transformation process remains implicit. Eclipse addresses this issue with a structured design trajectory. Moreover, the proposed design trajectory focuses on a novel aspect: identifying common computation kernels for designing reusable hardware implementations.

Figure 29. Eclipse coprocessor design trajectory. Application C-code is transformed into a (YAPI) process network that is largely independent of the target architecture. The ‘generic process network’ of a set of applications is restructured to extract common compute kernels that are eligible for coprocessor implementation. The tasks in the ‘Eclipse-tailored process networks’ are then redesigned into Eclipse coprocessors. The redesign mostly concerns the control part of the task code that handles the formatting of data streams into packets, explicit synchronization of data access, state save and restore actions, etc. The bare-bones compute kernels (‘function’ in the figure) aspects may be almost fully reused from the original process networks.

Figure 29 gives the trajectory that is followed to gradually transform a set of applications written in a high-level programming language (such as C) into software modules and simulation models of Eclipse coprocessors. Oftentimes, particular applications are transformed more than once to different hardware architectures over time. During this software-mapping trajectory, the code is restructured significantly on behalf of optimization. Optimized code has little or no reusability over different hardware platforms. Therefore, our software development trajectory is designed to promote reuse at various levels of optimization.

6.1.1 Generic process network

The first step is making parallelism and communication explicit in the application. The application code is transformed into a generic process network of an application, written in YAPI [85]. A YAPI application is specified in C/C++ as a set of concurrent processes that communicate through buffered communication channels, similar to a Kahn Process Network [80]. For communication, each process calls read and write primitives on their input and output ports that connect to communication channels. With the read primitive, the process consumes typed data in FIFO order from the channel or blocks when no data is available. The process produces data into the channel through the write primitive. This style of structuring applications fits well with the streaming and data-dependent nature of the targeted media-processing applications of Eclipse.

The target hardware template influences the choice of the algorithm and its partitioning into functional entities. However, a ‘generic’ model of an application is independent of other applications that may run on the target hardware, and has minimal dependency on the target hardware architecture. Thus, generic YAPI models can be reused in other architecture-definition projects.

Once the generic model is available, this becomes the foundation for any further application development. This is similar to having ‘reference C-code’ as a basis for optimizing code towards a programmable platform. Moreover, the step of transforming an application into generic YAPI provides the insight into the structure and behavior of the application that is needed for the next step.

6.1.2 Eclipse-tailored process network

The second transformation step towards a hardware implementation contains the rewriting of the generic YAPI model into the architecture-tailored process network of the application. Here, the generic YAPI model is restructured such that subsequently it can be mapped onto the target architecture. Note that when an application is later mapped onto other architectures, the generic YAPI is again restructured to match the characteristics of the architecture at hand. For Eclipse, several applications are restructured concurrently to make the generic computation kernels explicit. These computation kernels form the candidates for coprocessor implementation. This may involve changing the actual application algorithms to extract common application kernels, as illustrated in Section 6.2.2

In the decision of how to partition the application into YAPI tasks that are eligible for coprocessor implementation, the architect can execute the YAPI models to analyze bandwidth requirements on the data streams. Design-space exploration of different partitionings is typically skipped due to the (often large) effort of manually partitioning the applications. However, once a partitioning is made, DSE environments may allow the architect to further analyze the behavior of a YAPI task

mapped onto an abstract model of a target processor, for instance to decide between a mapping to a programmable DSP or Eclipse coprocessor.

6.1.3 Eclipse coprocessor design

The architecture-tailored application model forms the starting point for hardware development. To aid the process of modeling the target hardware blocks, the YAPI code is separated into two parts: a control part with the YAPI read and write commands, and a part with bare functions. The bare functions provide the functionality of the target hardware, stripped from communication-dependent control aspects. The bare functions (e.g. the computation of a DCT) are reused as is in the hardware definition, while the control part must be rewritten to match the specific communication architecture of the target. For instance, communication channels between Eclipse application processes are mapped onto data streams with communication buffers allocated in shared on-chip memory. The coprocessor designer needs to explicitly decide on the granularity of synchronizing access to these buffers when rewriting the YAPI read and write primitives into the Eclipse coprocessor-shell interface primitives of Section 3.2.

The design of coprocessor control includes the design of the Eclipse data streams. For the Eclipse architecture to be flexible, tasks running on the function-specific coprocessors have to support being connected into various graphs. The possibilities for reconnection increase dramatically if all streams are based on a uniform syntax that all coprocessors understand. A packet formatting of the data streams provides a common way to handle (dynamically) variable-size data chunks.

Following the example of Section 4.3.3, Eclipse data streams may be formatted as a stream of variable-length packets, consisting of a 2-byte header field next to a 0 to 255-byte payload. The header field gives the size of the payload and indicates the packet type. Using the header information, the coprocessor can decide to process the packet as basic media data, interpret it as meta-data and update the task state, or forward the packet to the subsequent coprocessor in the application pipeline. With such a packet formatting, the coprocessor designer may choose to interleave packets from different application streams in a single stream to reduce buffer-memory requirements and synchronization overhead.

The process of rewriting the YAPI primitives into Eclipse primitives may be aided by code transformation tools to minimize the effort of rewriting C-code. Furthermore, state-of-the art DSE environments [109] support trace transformation techniques that allow the architect to execute the YAPI task as if it were an Eclipse coprocessor on an abstract model of the system. Such environments make it possible to analyze different coprocessor implementations up-front, e.g. to make an educated decision between a pipelined versus non-pipelined implementation. In the MPEG case study of the following sections, we did not apply DSE

tools as these were not available at the time and relied on the strong background of the Eclipse team in MPEG for the partitioning and mapping.

6.2 MPEG application analysis

The following sections present a case study of the coprocessor design trajectory outlined in this chapter for a video (MPEG-2) coding Eclipse subsystem. This section shows the step from generic YAPI models towards Eclipse-tailored YAPI models of an MPEG-2 decoder and encoder. Section 6.3 details the design of the reusable MPEG coprocessors as a result of this analysis.

6.2.1 Generic decoder and encoder models

Figure 30 depicts the generic MPEG-2 decoder YAPI model [149], derived from C-software of the UC Berkeley MPEG decoder. The creation of this generic YAPI model involved an extensive restructuring of the functions and data structures in the sequential C-code to make parallelism and dependencies explicit.

The process Tvld parses an MPEG bitstream under control of a process Thdr. The Thdr process distributes the retrieved sequence and picture properties to other processes. The Tvld process sends motion vectors into a functional pipeline that retrieves the prediction data for the reconstruction of macroblocks. The coefficient data for the error blocks is sent into a second functional pipeline for run-length decoding, inverse scan, inverse quantization, and inverse DCT. The grain size of this coefficient data is a macroblock. A memory manager process Tmemman controls the access to the frame memories.

Figure 30. Generic MPEG-2 decoder YAPI model.

Figure 31 depicts the generic MPEG-2 encoder model, derived from reference C-code of the EMPRESS encoder [15]. The depicted encoder accepts raw picture data as input. The difference of the input data with motion compensated prediction data is sent through a functional pipeline of forward DCT, quantization, run-length encoding, and finally variable-length encoding. A second functional

pipeline performs inverse quantization, inverse DCT, and motion compensation to create the predication data for the next input. Both pipelines operate at a macroblock granularity. The memory manager is responsible for synchronizing access to the frame memories.

Figure 31. Generic MPEG-2 encoder YAPI model.

6.2.2 Eclipse-tailored decoder and encoder models

The choice of Eclipse coprocessors is influenced by the targeted performance of the MPEG-2 decoding and encoding applications. The first Eclipse instance targets concurrent decoding of two high-definition (HD) MPEG-2 bitstreams. Moreover, the coprocessors should also be capable of simultaneous encoding and decoding standard definition (SD) MPEG-2 streams.

A typical hardware-software codesign problem is the trade-off between load on the control processor (CPU) and the complexity of coprocessor control as implemented in its hardware. The load on the control processor for coprocessors operating on a macroblock or sequence granularity is clearly too large for simultaneous decoding of two (worst-case) HD sequences. Therefore, the coprocessors are designed to run independently for at least an entire MPEG frame. This decision implies that macroblock and slice properties must be provided to the coprocessors through regular input streams. Picture and sequence properties may be provided through separate auxiliary input streams, thereby removing the dependency on the Thdr process of Figure 30.

For the DCT, scan, and quantization, both the forward and inverse functions can be implemented with exactly the same hardware, only requiring different constants and quantization tables. For the DCT function to be reusable in the decoder as well as in the forward and inverse path of the encoder, it needs to be implemented a stand-alone coprocessor. However, the run-length, scan, and quantiza-

tion functions may be combined into a single coprocessor to keep the communication between these functions local in the coprocessor.

The MPEG decoder of Figure 30 implements the sequence of run-length decoding, inverse scan, and inverse quantization. Since run-length encoding is lossless, and the silicon area for run-length encoding/decoding is very small, run-length encoding in the forward path of Figure 31 can be done before forking off the inverse path. The encoder model needs to be restructured to implement this behavior. However, this new structure significantly reduces communication buffer requirements of the encoder as the stream buffer between the forward and inverse path remains in the compressed domain.

6.3 MPEG coprocessor design

This section describes the internal architecture of the coprocessors for MPEG decoding and encoding. We decided upon a 4-coprocessor model: 1) variable-length decode and encode (VLD), 2) run-length, scan and quantization (RLSQ), 3) discrete cosine transform (DCT), and 4) motion compensation/estimation (MC/ME). Figure 32 depicts the mapping of tasks onto these coprocessors to decode a single MPEG-2 stream. The RLSQ, DCT, and MC/ME coprocessors should be effective for both MPEG-2 decoding and encoding. Variable-length encoding will be handled in software.

Figure 32. MPEG-2 video decoding tasks mapped onto a video subsystem consisting of four coprocessors: VLD, RLSQ, DCT, MC.

6.3.1 Variable-length decoding and encoding (VLD)

The VLD coprocessor performs the parsing of an MPEG-encoded video bitstream. From this bitstream it extracts both the basic video data to be passed to the RLSQ coprocessor, as well as control information contained in macroblock and picture headers, such as quantization parameters. All this information is decompressed and sent into the appropriate stream buffers. This control information will change some of the internal state of the coprocessors later in the pipeline.

Sending this information as packets through the buffered streams is an important prerequisite for the concurrent (pipelined) operation of all MPEG coprocessors.

Figure 33. VLD coprocessor architecture.

The application throughput requirement for dual stream high-definition decoding requires dedicated hardware to achieve sufficient speed in the innermost loops of the decoding process. However, the parsing and decompression of less frequent slice and picture headers can be handled in software. If the hardware for the inner-most loops is equipped with programmable tables, then the combination with software processing of the outer parts allows for multi-standard decoding, ranging from MPEG(-1, -2, -4) to DVC decoding. The YAPI model of Figure 30 indicates the partitioning of the parsing and decompression process between a higher level—implemented in software on a small, dedicated RISC core—and the dedicated lower-level hardware as the Thdr and the Tvld processes. Figure 33 sketches the hardware architecture. We assume an SoC-level DMA engine or DSP will place the bitstream into a buffer in the Eclipse memory.

The interfaces between the RISC core and the VLD engine are such that the VLD engine replies on commands from the RISC core. Such commands are like: “provide me with the initial bits currently heading the bitstream”, “advance the bitstream by n bits”, “send these provided bits into the output stream”, “decode the block-data in the bit-stream to the output and reply with a ready-code”.

For the best programmability, the VL decode engine performs the Huffmann decode at block level with a rate of one symbol (run-level pair) every two clock cycles based on a down-loadable table. A MIPS core performs all higher-level decoding in software. Total coprocessor area is estimated 2 mm², in CMOS18, including a few kByte of caches for the MIPS. The VLD coprocessor reads the incoming bitstream from the Eclipse network.

6.3.2 Run-length, scan, and quantization (RLSQ)

The RLSQ coprocessor combines run-length decoding and encoding, zigzag scanning, and quantization functions. In case of encoding, the input to this coprocessor is a DCT coefficient block and output is run-level pairs. For decoding, the input is run-level pairs and output is a (possibly scaled/smaller) coefficient block. The order of execution of scanning and quantization is different for encoding and decoding. However, the same implementation can support both encoding and decoding since the order of execution does not change the outcome of these opera-

tions. The run-length encoding (RLE) function and run-length decoding (RLD) functions are instantiated separately (Figure 34).

Figure 34. RLSQ coprocessor architecture.

The RLSQ coprocessor operates on the granularity of DCT blocks and can switch tasks after producing one such DCT block. However, to enable multi-tasking, the coprocessor must take care of the internal state of each task, formed by the quantization and scanning tables. The VLD writes these tables in a stream buffer in the on-chip memory. At the start of a new task, the RLSQ coprocessor (re)loads the tables from this buffer into its local table memory. The previous tables do not need to be saved upon a task switch. The separation of data transport and synchronization in the Eclipse coprocessor-shell interface (Section 3.2) allows the tables to remain in the stream buffer until the coprocessor explicitly frees the memory space by synchronizing with the Thdr process.

The RLSQ area is estimated at 0.3 mm^2 in CMOS18: less than half the area required for DCT, as RLSQ requires significantly less computations and intermediate storage.

6.3.3 Discrete cosine transform (DCT)

The DCT coprocessor includes inverse and forward DCT functions as required for the MPEG-2 encoder and decoder. This coprocessor operates on the DCT block level and does not require any knowledge of the picture properties or sequence properties. It supports both scaling and block compression.

We chose the LLM algorithm [97] to implement both forward and inverse DCT functions. The LLM algorithm shares the same constants for multiplication in both inverse and forward DCT. Therefore, a DCT coprocessor can use the same data path for the implementation of inverse and forward DCT. Moreover, implementing dedicated multipliers with fixed constants provides high throughput at minimal hardware cost [86][133]. The internal state of this coprocessor is void after processing a DCT block. Consequently, the coprocessor can switch tasks on a block-level granularity without requiring additional hardware for state save/re-store.

The DCT area estimation of 0.9 mm^2 in CMOS18 is based on an implementation using A|RT designer [86], with added area for the connection to the coprocessor shell.

6.3.4 Motion compensation/estimation (MC/ME)

To efficiently compress video signals, MPEG exploits the temporal correlation between successive pictures. In a linear-wise scanning of the pictures from the left to the right and top to bottom, successive blocks of 16x16 pixels (macroblocks) are predicted from previously decoded pictures. Because these HD reference pictures are too large for on-chip storage, they are located in off-chip SDRAM memory. Consequently, the MC/ME coprocessor requires a connection to this off-chip memory to fetch the prediction data and to write the reconstructed pictures that are used as reference for the decoding of successive pictures.

Figure 35. MC/ME coprocessor architecture.

Figure 35 depicts the internal architecture of the MC/ME coprocessor. To share the hardware resources for prefetching and interpolation, motion estimation and motion compensation are joined in one coprocessor. The prefetch and write units of Figure 35 embed sufficient buffering to hide the large access delay of the SoC infrastructure. The prefetch unit can work ahead of the MC/ME unit by processing motion vector data provided by the VLD in a separate stream.

The prefetch unit copies the prediction data from the cache to its buffer. Although caching of stream-oriented data generally does not make much sense, the use of an SDRAM-based memory introduces some temporal locality due to its inherent transfer overhead. This can be exploited to reduce some of the scarce memory bandwidth. Jaspers [76] shows that a relative small direct-mapped cache of 6 kByte per task can reduce memory bandwidth on average with 30%. Any further improvement would significantly increase the cache size, e.g. a 24 kByte cache results in a 33% bandwidth reduction.

Computational complexity of the MC/ME coprocessor is limited (2D half-pixel interpolations). The area is dominated by buffers needed to operate on local data. For block-level operation, this is roughly 6 blocks, or 1 kByte. Additionally, we assume a cache to reduce traffic on the SoC infrastructure of about 5 kByte. In CMOS18 technology, 6 kByte of KFSRAM takes 0.45 mm² area. Total MC/ME and cache size is estimated at 0.6 mm² in the same technology.

7 Discussion

System-on-chip subsystems range from fully flexible subsystems that contain one or more programmable processors (e.g. ARM, TriMedia) to highly efficient dedicated hardware, such as a video-display engine. The choice of flexibility versus hardware cost, performance, and power constraints depends largely on their impact at SoC level. Fully programmable solutions are around a factor 100 more expensive in silicon area than a hardwired solution. However, today's full software audio (post-)processing implementations are negligible in the total bill of material of a multimedia chip, eliminating the need for a more efficient hardwired implementation. In contrast, video processing—such as MPEG-2 and MPEG-4 coding—constitutes a significant part of the overall SoC cost and power consumption. For the coming decade, video functions still require a hardwired implementation to compete in the high-volume consumer-electronics market.

Eclipse targets SoC subsystems in the media-processing domain and is designed as an architecture template to allow designers to quickly instantiate new SoC subsystems tuned to the targeted applications. In the previous chapters, we discussed three key aspects in the Eclipse template that allow such rapid subsystem development:

- *Shell services.* The coprocessor-shell interface greatly simplifies coprocessor design by providing services for data transport, synchronization, task scheduling, and configuration.
- *Distributed synchronization and scheduling.* As a result of Eclipse's distributed synchronization and distributed task scheduling, Eclipse can easily accommodate new coprocessor/shell combinations without necessitating changes to the rest of the system. The shared memory—centralized for the flexibility of buffer allocation and cost reduction—is a potential scalability bottleneck. However, the access to memory is fully abstracted by the coprocessor shell and can therefore easily be changed.
- *Parameterization.* The fully synthesizable RTL of the Eclipse infrastructure is highly parameterized (approx. 80 parameters in the coprocessor shell alone) to quickly instantiate a new Eclipse subsystem without necessitating redesign.

With the introduction of a architecture template for SoC subsystems, Eclipse effectively introduces a new hierarchical level in the design of SoCs, as shown in Figure 5. State of the art SoC platforms such as Philips Nexperia and TI OMAP do not extend the platform approach to the subsystem level yet. However, as SoCs become more complex, these platforms will be forced to introduce a next layer of hierarchy. The lack of subsystem templates combined with the expected

need for such an approach allowed us to apply for worldwide patents on many aspects of our approach.

The recent OpenMAX [106] standardization effort is a clear step in the direction of defining subsystems with standardized interfaces for streaming and a well-defined interface for configuring streaming applications towards the SoC. The Eclipse interface for streaming and application control—defined around the year 2000—fits remarkably well with the OpenMAX concepts, to the point where one may wonder if OpenMAX infringes on Eclipse patents. As such, Eclipse seems a good fit for a hardware implementation of the OpenMAX standard.

Media processing systems become increasingly dynamic to accommodate for late changes during development. While the design time of a new product increases due to the sheer development complexity, new products replace their predecessors in the market at an increasing pace. Therefore, the system requirements are often not known in advance. Flexibility is the key to accommodate the resulting late changes. With respect to fully hardwired solutions, Eclipse SoC subsystems increase application flexibility in three distinct ways:

- *Flexibility of interconnecting medium-grain tasks.* The coprocessor shells fully hide the interconnection of tasks from their coprocessors, allowing any task graph to be specified. The shells further extend this with support for multicast transparent to the coprocessors. The shared memory allows the allocation of an arbitrary number and size of buffers.
- *Flexibility of instantiating task graphs.* The separation of data transport and synchronization enables the design of coprocessors that have no restrictions in the number of task states they can support. The shell's task scheduler allows any number of tasks to be mapped onto a coprocessor.
- *Flexibility of processing.* With some added programmability, Eclipse coprocessors may accommodate late changes in application requirements. In principle, Eclipse coprocessors may also be deployed in new applications that were not considered during coprocessor design. Here, the designer must carefully weigh the benefits of highly application-specific hardware with the cost and performance penalty of additional coprocessor flexibility. Tensilica's Xtensa [151] is a good industrial example of a configurable processor that can be deployed as Eclipse coprocessor.

In the Eclipse coprocessor-design trajectory, similar applications are transformed simultaneously to extract generic compute kernels that are candidates for coprocessor implementation. A key issue is the excessive manual effort involved in restructuring the application into a form that is suitable for coprocessor implementation. The Compaan/Laura tools [134] automate this transformation to a large extent, by extracting process networks from the application specification and transforming the individual processes to RTL. Compaan/Laura tools transform a single application at a time into coprocessor hardware. To extract more flexible coprocessors that can be deployed in a range of applications, the tool

would need to be extended to automate the full Eclipse coprocessor design trajectory by transform multiple applications in parallel.

Eclipse is designed for soft real-time, memory-to-memory functions, where large (e.g. frame) buffers in off-chip memory at the inputs and outputs reduce load variations. We discern two approaches to deal with Eclipse's real-time requirements.

- *Resource-aware applications.* Applications adapt their behavior to the availability of resources. For instance, in an MPEG decoder, we may conceal a missed deadline by redisplaying the last frame, and recover by skipping the decoding of the next frame. Eclipse supports such application behavior through performance measurement and dynamic reconfiguration services.
- *Static end-to-end guarantees.* While the execution of application tasks is determined at run-time, the designer computes a realistic worst-case execution bound for the overall application that guarantees that all deadlines will be met. Both the application and architecture must allow the computation of such bounds, for instance by providing (statistical) upper bounds on dynamic behavior [47].

Considering Eclipse's flexibility requirements and the dynamic nature of the targeted media applications, designing Eclipse for worst case is not cost-effective. Therefore, Eclipse was designed for resource-aware applications that gracefully degrade output quality in overload situations. However, many key aspects in the architecture—such as distributed synchronization, round-robin arbitration of single-word bus requests, and processing-time budgets in the task scheduler—cater for static end-to-end predictability, provided the targeted application allows for a cost-effective worst-case bound. To what degree Eclipse can support statically predictable and/or dynamic resource-aware applications is subject to further research.

In recent years, networks-on-chip (NoC) [4][58] have gained a high interest in both research and industry as a means to improve the real-time behavior of inter-processor communication. The Eclipse communication network could easily be replaced with an NoC, as the shell hides the details of the communication network from the coprocessor. Even though this may increase predictability of communication, the inherently unpredictable nature of multimedia applications still requires a graceful degradation mechanism to cope with overload situations. Moreover, while an NoC provides clear benefits at the SoC level, deploying an NoC at the subsystem level may result in an overly complex solution.

Part II

Eclipse simulation

The Dutch Open Telescope (DOT), on the rim of an extinct volcano on the Canary Island La Palma, is an innovative telescope for high-resolution studies of the solar atmosphere [130]. The author visited the DOT in 1996 when it was being assembled. Over the past decade, the DOT became the leading producer of movies that vividly portray the dynamic behavior of the solar atmosphere with unprecedented image sharpness.

The open design exploits the favorable conditions at the site. The trade wind from the ocean blows strongly upslope over the volcano rim and confines the turbulence caused by solar ground heating to a thin layer well below the 15-m high telescope platform, so that it does not spoil the image quality. The wind also flushes the mirror and telescope structure, suppressing internal turbulence within the telescope.

The remaining turbulence in the higher atmosphere above the telescope is removed through speckle reconstruction, a numerical technique using bursts of 100 images to derive a detailed model of the wavefront perturbations along the line of sight. The DOT employs a 70-CPU parallel speckle processor in a nearby building. Every CPU and power supply is individually

water cooled with heat storage in a large water tank and release at dusk when dissipation into the air does not jeopardize observations.

The DOT's six synchronous CCD cameras sample the solar atmosphere from the visible surface up to the high chromosphere seen in the hydrogen Balmer alpha line—which causes the beautiful purple color of the chromosphere during a total eclipse.

8 Multiprocessor architecture simulation

The Y-chart [84][85][5], visualized in Figure 36, depicts a fundamental separation between *function* (what the system is supposed to do) and *architecture* (how it does it) [83]. Clearly, this is not a separation between hardware and software in the more traditional hardware/software codesign terminology, as software is merely the form a function takes when it is ‘mapped’ onto a programmable processor. The function is represented by an *algorithm*, a specific recipe for realizing the desired function. For instance, a de-interlacing function can be implemented by straightforward copying of missing odd or even lines from the previous field, or by a more complex algorithm such as a median filter. In contrast to the function, the algorithm reveals to some extent how the function executes on the architecture. Therefore, an algorithm is always coupled to a certain style of architectures onto which the chosen algorithm provides the most efficient execution of the desired function.

Figure 36. Y-chart design methodology. A system function is represented in the form of an algorithm. The algorithm aims to be as independent as possible of the system architecture. Subsequently, the algorithm or algorithms are mapped onto the architecture, and the resulting implementation is analyzed e.g. through simulation. From the analysis, the designer may derive improvements on the algorithm or the architecture and repeat the process. Ultimately, the designer may change the system function, e.g. by reducing output quality to reduce hardware cost or complexity.

The mapping of function to architecture is an essential step from conception to implementation. The mapping process determines where the functions to be implemented are assigned to the components of the architecture. After mapping, the performance and cost of the function-architecture combination can be evaluated, potentially resulting in modifications in the architecture, the algorithm of the de-

sired function, or in the mapping itself. Clearly, modifications to the desired function can only be done when the function is not dictated by standards or similar external requirements.

Figure 37. Cost of redesign for architectural decisions taken at different abstraction levels in the design pyramid [84]. The cost of a design change at the top is small but decisions at this level have a large impact on the final result. As the design incorporates more details while progressing down along the pyramid, a design change quickly becomes very costly. Hence, it is important to make key design decisions as early as possible in the design trajectory. However, at the upper half of the pyramid, the architect has only a high-level view of the system and must rely on experience to make well-informed design decisions. A flexible simulation environment that evolves as the design moves down the pyramid can aid the architect with early feedback on design decisions.

We employ the Y-chart approach throughout the Eclipse design trajectory. Figure 37 depicts the so-called *design pyramid* in which an architectural choice at a certain abstraction level determines the range of architecture options at the next lower level of abstraction. As the architecture specification matures from an initial concept, we develop increasingly complex models of the architecture to validate our design decisions. This is reflected by the amount of design data (files, documents, etc.) needed to specify and validate the design. For instance, the abstract description of the task scheduler in Section 5.3 takes four pages of text, compared to the 2428 lines of VHDL code (including comments) needed to specify the task scheduler at the RTL level. Therefore, the cost of coming back to an earlier architectural decision increases drastically while proceeding from top to bottom through the design pyramid. Thus, early feedback on performance and cost are needed to allow the architect to reduce the cost of redesign by validating architectural choices early in the design trajectory. This allows an iterative design cycle where architectural decisions are taken as early as possible in the design trajectory from an abstract description to an increasingly detailed architecture specification.

During the exploration phase of a new architecture, an architect typically relies on experience, backed up by back-of-the-envelope calculations and implementation case studies to make high-level architectural choices. A typical example is the

choice between distributed or shared memory architectures. For instance, we performed transistor-level simulations of the local Eclipse bus early in the exploration phase to back up our estimates that a bus-based shared memory architecture could provide sufficient bandwidth. Once the architectural choices get more detailed, such as the choice of a cache replacement strategy, the architect typically needs more extensive and reliable methods of validating design decisions. With the tremendous complexity of current media-processing architectures, the architect can rely only on simulation for validating a design.

Immediate feedback on architectural choices throughout the design trajectory calls for a flexible simulation environment that allows gradual refinement of abstract models of the architecture into cycle-accurate simulation and RTL models. In addition, such an environment can support early development and evaluation of application and control software—before the actual silicon is available. This chapter describes the requirements for such a simulation environment and sets the stage for the introduction of the Eclipse simulation technology in Chapter 9.

8.1 Application development

In the design of media processing systems, the development of application software is too often postponed until the first hardware instance of the system has become available. As system design is traditionally done by hardware architects, the complexity of developing application software is often underestimated. Early development of application software—before the hardware instance is available—serves two important purposes:

- *Reduced time-to-market.* When application development starts before the hardware architecture (RTL, layout) is completed, applications can be developed in parallel with the hardware design. Ideally, application software development completes at the same time the hardware becomes available. A clear example of the benefits of parallel application and hardware development is the development of game consoles. The number of games available at market introduction significantly influences the success of the console.
- *Hardware-software codesign.* Parallel development of application software with the hardware architecture drives the architects to provide a clean programming model that matches the requirements of the application programmers, following the Y-chart design methodology of Figure 36. The ease with which application programmers can exploit the available parallelism in the architecture largely defines the actual application performance. For the game console example of the previous paragraph, applications are typically developed by third parties. The first games for the Sony Playstation 2 could not exploit the architecture to its full potential because of the difficulties in programming the Emotion engine [103] (e.g. application programmers had to manually take care of memory allocation and synchronization). The PC-style

programming model of Microsoft's X-Box makes the X-Box easier to program, reducing development cost and decreasing the time needed to develop new games. Therefore, the first X-Box featured more advanced games than Playstation 2, even though the Playstation's Emotion engine provides more intrinsic processing power than the X-Box architecture.

Parallel application and architecture development calls for a simulation environment on which application programmers can develop and verify application software before the hardware is available. This gives two main requirements for the simulation environment. First, the application software must execute as is on both the simulation environment and the actual hardware. Secondly, the simulation environment must be sufficiently fast to allow verification of the application software at an acceptable speed.

Any Eclipse application software consists of two parts: the configuration software to dynamically set up and manage the application graphs executing on a control CPU (or the media processor) and the software Eclipse tasks running on the media processor. The requirement that application software should execute *as is* on both the simulation environment and the actual hardware means that the simulation model of the architecture must define all registers visible to the control software. Consequently, the coprocessor shells must be modeled at a low level of abstraction for development of control software—close to the hardware micro architecture.

8.2 Architecture development and verification

The gap between the conceptual specification and the system implementation is rapidly becoming the most important bottleneck in the design process. There is a need for a formal and executable specification hand-over from system designer to system implementer, and an environment that allows gradual refinement of the designer's specification into a specification understood by the system implementer (e.g. VHDL for hardware implementation and C for software implementation).

A high-level simulation model of the architecture serves as an executable specification for the system implementer. In our experience, building an executable specification forces the architect to address important details that otherwise only show up during hardware design. Clearly, an executable specification in the form of a simulation model must match the level of abstraction corresponding to a design phase of Figure 37. As the design evolves from the top to the bottom of the design pyramid, the specification of the architecture will uncover more and more details of the architecture. The executable specification must follow this process of gradual refinement.

During the design trajectory, different parts of the architecture will be modeled at different levels of abstraction. The design is slowly refined in small sections to

add the necessary hardware and timing constructs to produce a good design. This encompasses the refinement of the internal structure of the model up to the detailed micro architecture, the refinement from abstract data types to bits and bit vectors, and the refinement of interfaces to wires and protocols. Using this refinement methodology, the designer can more easily implement design changes and detect bugs during refinement. There are four distinct levels of abstraction in modeling an architecture listed below, with the corresponding Open SystemC Initiative (OSCI) [137] added between brackets.

1. *Abstract behavioral models (programmers view, PV)*. At this level, hardware functionality is modeled in an abstract way. The micro architecture of the hardware module is not modeled, and the model provides only a rough timing estimate. Typically, the functionality is modeled as a sequential process, without expressing the parallelism in the hardware. On its interfaces, the model communicates through remote-procedure call mechanisms.
2. *Abstract near cycle-accurate models (programmers view with timing, PVT)*. At this level, the model approximately reflects the timing behavior and parallelism of the hardware module. The functionality is bit-accurate. The micro architecture of the hardware is not modeled. Typically, the functionality is modeled in separate threads to mimic the parallelism in the system. On its interfaces, the model communicates through remote-procedure call mechanisms.
3. *Cycle-accurate models (cycle-callable model, CC)*. Timing is nearly cycle accurate although the micro architecture of the hardware module is not modeled in detail. However, the model typically implements all functionality, architected registers, and interface signals bit-accurate.
4. *RTL models*. At this level, the detailed micro architecture of the hardware is modeled including all parallelism, registers, and signals. The hardware can be synthesized from this model.

Ideally, an executable specification can gradually be refined from the behavioral to the RTL level within a single environment to reduce modeling and verification effort and facilitate design reuse.

8.2.1 Design verification

Clearly, simulation models must be thoroughly tested before they are used to evaluate architecture design decisions or serve as a specification for hardware design. The test bench used to test the executable specification can be refined or used as is to test the implementation of the specification. As models are refined from abstract behavioral models to RTL, the number of verification tests increases. However, tests developed for abstract models can be reused as baseline test bench for more detailed models. Ideally, the verification tests for cycle-accurate simulation models can directly be applied as a first test bench for RTL design.

For the Eclipse template, it is impossible to exhaustively test functionality and timing for all settings of architecture parameters. Therefore, we advocate a two-stage approach. All modules are tested individually for corner cases, or ‘everything that could possibly break’ [78]. The interaction between modules is verified by executing demanding applications for a limited number of parameter settings. Next, when an Eclipse subsystem is delivered to a customer, the architectural parameters are fixed and the modules are subject to an exhaustive test.

A complex architecture design such as Eclipse typically includes a large number of verification tests. The simulation environment must support fast and automated execution of test benches to enable an efficient design process. This requirement is essential to facilitate integration of architecture models designed by independent parties. We have hitherto not encountered a test environment supporting regression tests on functionality as well as timing of parameterized SystemC modules. Therefore, we mainly performed validation tests of the overall Eclipse simulation model.

While the development of Eclipse models was mainly restricted to a single site, we relied on verification tests to share model ownership between developers. This allowed us to adopt Extreme Programming practices [78] where the overall software architecture of the simulation models is continuously refactored to accommodate change, resulting in an efficient and modular software architecture [48].

8.2.2 Design reuse

Many CAD vendors are investigating the possibilities of architectural modeling based on a library of reusable architecture models. A designer can quickly construct new architectures by instantiating and interconnecting reusable models. Reusable simulation models are often highly parameterized to increase reusability of the modeled function, while modeling the hardware with sufficient accuracy. This permits the designer to tune an instance of the model to the architecture at hand. Parameters are memory size, bus width, arbitration strategy, etc.

Similar to the development of reusable hardware architectures, reusable architecture models must separate communication from function to increase reusability. While refining a simulation model from a high level of abstraction to a cycle-accurate model of the actual hardware, the boundaries between communication and function change. For instance, at an abstract level, the interconnection between processors can be considered as communication, while at a concrete level, a DTL concentrator to connect processors is considered functionality. The more detailed functionality and communication are defined, the more vague the boundary between communication and function becomes. It is therefore important to focus on the *interfaces* of a module, rather than on the specific implementation of the communication protocols. A well-defined interface may allow a module to connect to both abstract communication channels as well as to specific communication

means, such as a DTL concentrator or AMBA bus [3]. Current simulation environments typically provide constructs to enable such a separation of communication and function. Here, we apply the following definitions [137]:

- a *module* implements a certain function of the architecture;
- a *port* is specified with an *interface*, which declares a set of methods that a module can use through the port. Ports are therefore part of a module;
- a *channel* implements one or more port interfaces and serves as a container for communication and synchronization.

In the system design community, there have been many standardization activities to enable reuse of IP-blocks. For instance, AMBA [3] standardizes SoC bus interfaces, facilitating reuse and easy integration of third party SoC subsystems. Currently, there is a strong trend to extend such standardization activities to architecture simulation models. For instance, the Open Core Protocol (OCP) [65] shows a combined effort to extend standardization of data transport interfaces between IP-blocks to the SystemC simulation environment.

The separation between communication and function through well-defined interfaces allows a designer to refine the modeling of communication and function independently. For instance, a designer can connect an abstract model of a processor to a communication channel—that explicitly models a DTL protocol—to communicate with an RTL model of a second processor. Alternatively, a designer may connect several detailed processor models via an abstract communication channel to increase simulation speed. To this end, the OCP defines four layers of abstraction in modeling of communication:

1. *Message Layer*: direct communication channels, no arbitration protocols;
2. *Transaction Layer*: arbitration and communication protocols, no explicit signals. Timing and protocol specifics may be included, but not necessarily cycle accurate;
3. *Transfer layer*: protocol refined in a cycle accurate way, explicitly with signals, allowing co-simulation with RTL;
4. *RTL layer*: protocols and signals modeled at the levels of gates and wires, including gate and wire delays.

Clearly, these layers show a correspondence with the abstraction levels for architecture modeling introduced in the beginning of this section. In Figure 45 on page 116 for example, the communication of the behavioral model of the Eclipse co-processor with its shell is modeled at the transaction layer. The wrapper refines the `Report` and `GetTask` requests to the transfer layer to allow communication with an RTL model of the task scheduler.

8.3 Architecture design space exploration and validation

To handle the complexity of developing an architecture template such as Eclipse, architects need feedback on design choices as early as possible in the design trajectory of Figure 37. During the exploration and specification phases, feedback from an executable specification of the architecture is essential. Due to the irregularity and data-dependency of media-processing applications, it is often impractical to develop theoretical models of the media-processing architectures. Here, architects resort to simulation techniques to obtain early feedback on design choices.

Exploring the architectural design space early in the design trajectory is only feasible if the simulation environment is sufficiently efficient and flexible to evaluate a wide range of architectures. Designers must balance the effort required to model chosen points in the design space with the accuracy of quantitative feedback from simulations. To reduce modeling effort, designers model the architecture at a high level of abstraction and aim to reuse models of architectural elements such as memory and processors over different architectures.

Typically, the interconnection and interfaces of reusable models must be fixed during instantiation of the models, while model parameters such as cache size and cache replacement strategy can be varied at run-time in a high-level architecture description [64], often in the form of a *machine description file* (MDF). Automated design space exploration (DSE) requires retargetable simulation to tune architectural parameters once the high-level architecture has been decided.

Typically, these parameters span an immense design space of which only a small part actually defines sensible alternatives. For example, Hekstra et al. [66] show that the design space of allocating function units to a five issue-slot TriMedia processor [36] amounts to 10^{15} alternatives. A select number of carefully chosen experiments allowed them to prune the design space to less than 6000 alternatives. As the number of viable alternatives remains very large, simulations must execute extremely fast to allow the architect to explore a significant part of this design space.

8.3.1 Trace driven simulation

Trace-driven simulation separates function from architecture in the simulation model. The ‘application’ models the function, while the ‘architecture’ models the timing behavior and structure of the hardware. An application may encompass parallel processes that communicate via abstract communication channels. Application processes map onto processors, and the communication channels map onto buses and wires in the architecture. The application executes in isolation from the architecture model. Executing the application generates traces of activity, such as

read and write actions to communicate between parallel processes, or executions of specific computations (Figure 38). The architecture model interprets these traces and models the timing associated with the trace actions.

Traces are explicitly mapped to modules in the architecture, according to the mapping phase in the Y-chart of Figure 36. The communication of traces from application to architecture occurs via files, sockets, or procedure calls in a co-simulation between the application execution environment and the architecture execution environment.

A key benefit of trace-driven simulation is the separation of trace generation and trace consumption, effectively decoupling the application from the architecture. This decoupling allows simulation of several architectures from a single application trace, or a co-simulation with an application that is supplied in binary format by an external developer. Moreover, the independent execution of application and architecture allows the use of non-functional architecture models that only model the timing of application traces. In such a setup, trace-driven simulation is frequently applied to explore the memory subsystem of an architecture based on address traces. Here, the architecture only models the behavior and timing of the memory subsystem, while the actual data does not pass through the architecture. Such a non-functional architecture model greatly increases simulation speed, and thereby permits the designer to cover a larger area of the design space.

Figure 38. Application traces interpreted by the architecture model [110]. The application executes independently from the architecture model. Application execution information, such as read and write actions on data streams, is captured in trace buffers. The architecture models the time required to execute the application actions in the traces and reports, e.g. timing and utilization information.

8.3.2 Execution-driven simulation

In execution-driven simulation, the execution of the application and architecture are interwoven. For hardwired processors, there is no clear distinction between

application and architecture; application execution is handled by the architecture models. Execution is based on discrete events, such as a rising clock edge.

Execution-driven simulation supports a gradual refinement trajectory from abstract to RTL-level models. In trace-driven simulation, such a refinement is not possible for the architecture models that actually interpret the traces (the coprocessors in Eclipse). Eclipse applies execution-driven simulation to support a gradual refinement from abstract to RTL-level Eclipse models.

Execution-driven simulation is beneficial when the architecture directly influences the behavior of the application. In Eclipse, the response to a `GetSpace` request depends on the global dynamic behavior such as the differences in execution speed between a producing and consuming task on a data stream. On a failing `GetSpace` request, the application task—that is, the coprocessor functionality—behaves differently from when the `GetSpace` succeeds. Similarly, the answer provided by the architecture on a `GetTask` request defines what functionality the coprocessor will execute next. This in turn depends on the execution time of the current task (budget), and the filling of communication buffers in the architecture.

In trace-driven simulation, such non-deterministic application behavior can change the application traces for different architectures. Trace-driven simulation techniques may address this problem by generating the traces on the fly in a co-simulation between application and architecture. Application threads are suspended upon a global event (`GetSpace`, `GetTask`) and resumed when all threads have reached the same point in simulated time as the suspended thread. Clearly, when global events occur often, such explicit synchronization decreases the advantage in simulation speed of trace-driven simulation.

8.4 Multiprocessor architecture simulation environments

This section gives a brief overview of simulation environments for multiprocessor architectures and discusses the application of these environments to model and analyze the Eclipse architecture. Table 4 and Table 5 give a comparison of the discussed simulation environments. A more extended overview of simulation environments is found in [31], [153], and [95].

8.4.1 Architecture design-space exploration environments

This section outlines a number of architecture design-space exploration (DSE) environments, summarized in Table 4. The summary is not intended to be complete; it presents DSE environments that were in the scope of the Eclipse project

within Philips Research. Gries [59] presents a more detailed survey of DSE environments.

Table 4. Architecture design-space exploration environments.

	Spade	Sesame	Cosy-VCC	OCAPI-xl
Technique	Trace-driven simulation	Trace-driven simulation	Execution-driven simulation	Execution-driven simulation
Application model	KPN	KPN	KPN	Communicating processes
Application implementation	YAPI	C++ run-time environment with application processes in C, C++, or Java. YAPI supported via trace transformation.	YAPI imported in VCC	Communicating C++ processes, annotated with timing behavior
Architecture model	Cycle-based	Discrete event	Discrete event (VCC), Cycle-based (TSS)	Refined application processes containing OCAPI-xl objects, executed in a discrete-event simulation
Architecture implementation	Library of generic TSS modules	Pearl models, Wrappers for SystemC	Library of VCC modules, Cosy communication modules to refine YAPI transactions	Generation of ANSI-C, or VHDL/Verilog from OCAPI-xl objects
Level of abstraction	Near cycle-true	Behavioral to near cycle-true	Behavioral (VCC) to cycle-accurate (TSS)	Behavioral to cycle-accurate
RTL support	VHDL, Verilog co-simulation through TSS	Include ISS/RTL models via ‘trace calibration’ [138]	VHDL, Verilog co-simulation through TSS	Support for custom wrappers to external modules
DSE support	Netlist, TSS viewports, mapping file	YML with GUI for application/architecture modeling and mapping, design-space pruning using analytical models and genetic algorithms	Graphical editors for application, architecture, mapping.	-
Analysis support	TSS viewports, TCL scripts		Visualization of traces	VCD, Debug traces

Artemis: Spade and Sesame

The Artemis simulation workbench [110] provides two complementary simulation trajectories for system architecture design-space exploration: Spade and Sesame. The Artemis modeling and simulation environment facilitates the performance analysis of embedded-systems architectures in a way that directly reflects the Y-chart approach, separating application and architecture models. Applications are modeled as Kahn process networks and mapped onto architecture modules in a trace-driven co-simulation of application and architecture.

Spade [93] provides a small library of architecture modules, such as a CPU, generic bus module, memory module, and interfaces. These modules are imple-

mented in TSS (Section 8.4.2). Traces from the application model written in YAPI are mapped onto the TSS modules through trace queues. The TSS modules execute the traces and provide the timing behavior for read, write, or execute actions in the trace. The modules only model the timing behavior, but do not perform the actual computations or data transfers.

Sesame [24] builds on the work of Spade and Mermaid [108] and employs an XML-based language—called the *Y-chart modeling language* (YML) [24]—to specify the application and architecture structure, as well as the mapping of application event traces to the architecture models. YML is similar to Ptolemy’s MoML [89], but only supports a subset for the Kahn process network domain. However, YML adds built-in scripting support, allowing the construction of YML templates that are instantiated with different parameter settings. The Sesame architecture models are implemented in the Pearl discrete-event simulation language [101]. In contrast to Spade’s cycle-based TSS simulation, Pearl provides easy construction of abstract architecture models and fast simulation.

A key aspect of trace-driven simulation is the refinement of application traces into operations that match the computational model imposed by the underlying architecture. Spade proposed a transformation of application `read` and `write` traces into operations that fit the underlying architecture more closely [92], for instance by translating the blocking read into an Eclipse `GetSpace`, `Read`, and `PutSpace` combination. This is implemented in Sesame with an intermediate mapping layer between application and architecture models that also handles the synchronization and scheduling of traces mapped onto the architecture [109]. The mapping layer supports parallel or pipelined execution of trace events (both communication and computation) generated from a single application task onto parallel function units in the architecture.

The system-level behavior of distributed synchronization and distributed task scheduling is the main target for a design-space exploration of the Eclipse template. The trace transformation techniques implemented in Sesame provide powerful means to model Eclipse coprocessors. The transformations allow to explore different coprocessor implementations, e.g. pipelined versus non-pipelined. However, Eclipse coprocessors often depend on the timing of synchronization and task scheduling performed in the architecture, as discussed in Section 8.3.2. Sesame does not provide mechanisms to synchronize the application execution with such time-dependent feedback from the architecture.

Cosy-VCC

The Cadence Virtual Component Co-Design (VCC) environment [21] is based on the Berkeley POLIS [5] and Cadence Felix initiatives [98]. VCC targets design-space exploration through graphical representations of behavior and architecture components. The behavior (i.e. the application) is defined by interconnecting functional blocks, imported from C, C++, SDL, MatLab, and behavioral HDLs.

The architecture is modeled by interconnecting architectural modules, such as CPUs, DSPs, RTOSes, buses, and memories. VCC provides an extensive library of such architectural elements, modeled at a high level of abstraction. The mapping editor enables system designers to map behavioral models onto architectural elements. Behavior and architecture models are executed together in the VCC discrete event simulation kernel.

The Cosy-VCC flow [16] relies on YAPI for the application model. The YAPI model is captured in VCC, and is mapped onto architecture models within the VCC environment for design-space exploration. Once architectural choices are assessed in VCC, the design can be exported to TSS for cycle-accurate co-verification. The focus of Cosy is on communication refinement. Within VCC, the YAPI read and write transactions are refined into system transactions that map onto software or hardware components [18]. Communication modules [17] unravel the system transactions into more detailed interfaces, such as the virtual component interface (VCI) to interface to physical bus protocols. These communication modules also provide for delay models that allow to assess the impact of communication parameters on performance.

OCAPI-xl

OCAPI-xl [143] is the second generation of the OCAPI [131] environment developed at IMEC that focuses on exploring the hardware/software partitioning throughout the design trajectory.

Applications are modeled as concurrent processes that communicate through a set of communication primitives (messages, semaphores, or shared variables) that can be mapped onto operating system calls for software implementations or translated into hardware protocols. Time is an integral part of the specification, e.g. each process must be annotated with its execution time. In the time-annotated application model, the functionality inside each process can execute in a foreign language such as Matlab or C routines, captured from the initial application specification. The resulting concurrent timed-system model is used to explore different hardware/software partitioning.

Next, the designer (incrementally) refines the functionality and timing of the high-level processes. Thereto, the designer decomposes each process into a set of actions implemented in OCAPI-xl objects. The objects perform expressions on specified-length integers, as well as control flow statements. Every process is refined in the same, implementation independent way. The path from a fully refined model down to implementation is completely automated, thanks to the OCAPI-xl objects and communication primitives that target both hardware and software. It results in plain C-code for software components and synthesizable VHDL or Verilog for the hardware parts. Since the OCAPI-xl objects used by the designer are independent of the (event-driven) simulation kernel, OCAPI-xl can execute on top of the core library of SystemC, using only the SystemC kernel and data types.

Although unavailable during the design of Eclipse, the OCAPI-xl methodology of capturing the functionality of the initial specification in processes that communicate through hardware/software-independent primitives fits to the approach of Section 6.1. Unfortunately, OCAPI-xl provides only minimal support for performance analysis and debugging.

8.4.2 Architecture modeling and verification environments

This section outlines a number of architecture modeling environments, summarized in Table 5. The overview is not intended to be complete, it reflects the modeling environments that were in the scope of the Eclipse project during the design of Eclipse.

Table 5. Architecture modeling and verification environments.

	TSS	SystemC 2.1	ConvergenSC	Ptolemy II / Teepee
Execution model	Cycle based (FSM)	Event based multi-threading	Event based multi-threading	CSP, CT, DE, DDE, DT, FSM, PN, SDF, SR ²
Implementation	C library, TCL user interface	C++ library	C++ library with tcl/tk scripting and visualization	Java packages, XML-based graphical user interface
Level of abstraction	(Near) cycle-true	Mixed: behavioral to RTL	Mixed: behavioral to RTL	Mixed: behavioral to cycle-true
RTL support	VHDL/Verilog co-simulation	Native and VHDL/Verilog co-simulation	RTL cosimulation through 3 rd party tools	-
Retargetability	Netlist, viewports	-	Generation of buses, interface wrappers, and module connections	XML-based Netlist
Analysis support	Viewports, VCD ³	VCD ³ , debug traces	Extensive analysis views: tasks, processors, buses, memory	Signal plotting

2) *Mix of models of computation: communicating sequential processes (CSP), continuous time (CT), discrete events (DE), distributed discrete events (DDE), discrete time (DT), finite state machines (FSM), process networks (PN), synchronous dataflow (SDF), synchronous/reactive (SR).*

3) *Value change dump file to display waveforms of communication signals.*

TSS

TSS (Tool for System Simulation) is a Philips in-house architecture modeling and simulation framework. TSS was the de-facto standard at Philips during our Eclipse development. Since then it has been replaced by SystemC. In TSS, an architecture is a network of interconnected modules, which are modeled in C. TSS

modules communicate via ports. Modules are instantiated and their ports interconnected at run-time from a netlist file. The netlist allows a designer to instantiate a system from modules available as libraries in C, VHDL, or Verilog.

TSS provides a TCL command-line interface to communicate with the modules through their *viewports*. A viewport is a special port on a module that does not exist in the actual hardware, and can be used to set parameters in the module at run-time, or to access performance measurement data. User-defined TCL scripts provide a powerful mechanism to read out viewports and collect performance data for a complete simulation run.

The requirement for easy application development contrasts the cycle-based execution of TSS. The lack of threads and abstract communication channels inhibits mixed execution between abstract coprocessor models and detailed models of the Eclipse shell. Therefore, we developed a custom C++ based simulation environment on top of a lightweight thread package [82]. TSS was used to enable co-simulation with RTL-models. Thereto, the simulator refined the abstract communication channels between modules into TSS ports interconnected via the TSS netlist at run-time. When SystemC 2.0 became available, the Eclipse C++ models were ported to SystemC to comply with NXP's SystemC design flow.

SystemC

Based on the results of the Scenic project [61], *SystemC* [60][137] is a C++ class library of objects such as processes, ports, channels, signals, etc. for system-level modeling. SystemC employs a discrete time model upon which a broad range of computational models can be implemented. Some models of computation that can be easily modeled in SystemC are static data flow (SDF), Kahn process networks (KPN), or discrete event models used for RTL hardware modeling. The basic mechanisms to model communication and synchronization in SystemC—interfaces, channels, and events—were inspired by similar constructs in SpecC [31]. Channels, interfaces, and ports are built on top of the event-based simulation kernel. SystemC provides an extensible library of primitive channels for FIFO buffers, semaphores, hardware signals, etc. In addition, SystemC relies on C++ operator overloading to allow a gradual refinement of abstract data types to fixed-point representations or hardware signals.

SystemC aims at providing a C++-based modeling and simulation environment to gradually refine abstract behavioral modules into RTL hardware descriptions. To this end, the SystemC community is in the process of defining modeling guidelines, standardized interfaces, and libraries of generic models such as a bus arbiter. The open source SystemC initiative has become the de-facto industry standard for behavioral and cycle-accurate architecture models. However, the industry still tends to rely on existing RTL languages such as VHDL and Verilog for RTL modeling and synthesis, as this is deeply ingrained in their design flows.

This may change in the future, as industrial-strength SystemC synthesis tools such as Cynthesizer [29] are gradually adopted.

The multi-threaded and event-based model of SystemC closely matches the requirements for modeling the Eclipse architecture, which allowed us to port our custom C++ models with relative ease to SystemC 2.0. For RTL-level modeling and synthesis, Eclipse relies on VHDL and deploys the Cadence NC-SystemC design flow to co-simulate VHDL modules with abstract models in SystemC.

CoWare N2C and ConvergenSC

The CoWare N2C environment [116] addresses three major requirements for SoC specification: *reuse* through separation of functional and communication behavior; *modularity* by splitting the design into communicating processes; and *design abstraction* by linking different design environments together. CoWare modules communicate through remote procedure calls (RPC), i.e., one process can trigger the execution of a procedure in another module. From the designer's perspective, different modules can be programmed using the C, C++, or VHDL language, communicating with the external world via explicit RPC calls. This enables mixing different levels of abstraction, for example, an abstract module written in C communicating with an RTL-level VHDL module.

To facilitate the RPC mechanism, CoWare's interface synthesis tools generate all the logic needed to interconnect modules [146][94]: protocols, address decoders, bus arbiters, and bridges, as well as software drivers for memory-mapped I/O and interrupt service routines. Interfacing hardware and software builds on CoWare's Symphony tool [146], which defines a parameterized processor template to interface programmable and hardwired processors using a library of processor models and I/O wrappers. Communication is based on Hoare's model of rendez-vous as defined for CSP [68].

In 2004, CoWare released ConvergenSC as the SystemC-based successor of the N2C environment, built around a proprietary implementation of the SystemC kernel adding scripting and user interfaces in TCL/TK. Similar to Cadence NCSysC, ConvergenSC defines an integrated environment for instantiating and connecting modules, debugging, performance analysis, and co-simulation. Co-simulation with HDL models is provided through inclusion of 3rd party co-simulation tools, such as Cadence NC-SC. ConvergenSC's performance analysis API and corresponding visualization tools for task execution, bus load, memory accesses, etc. in particular could furnish valuable extensions to the Eclipse SystemC model.

Ptolemy II

Ptolemy II [88] defines a Java framework for heterogeneous modeling and design of concurrent systems. Ptolemy is not governed by a specific model of computation but rather supports a heterogeneous mix of computational models encom-

passed in Ptolemy *domains*. To this purpose, the Ptolemy framework heavily relies on object-oriented techniques such as inheritance and polymorphism to allow components from different domains to interact with each other. Currently supported domains are among others synchronous dataflow (SDF), discrete event (DE), and Kahn process networks (PN) [57]. Ptolemy incorporates a visual editor framework (Vergil), which in turn relies on an XML-based file format (MoML) to describe Ptolemy models.

The Mescal [100] project extends Ptolemy with architectural domains for modeling VLIW-like processors, memory hierarchies, and inter-processor communication structures. Thereto, the Mescal implementation—called Teepee—provides libraries of parameterizable architectural components. Mescal aims to implement the Y-chart methodology for design-space exploration through the deployment of formal computational models in different views (application, mapping, micro architecture, compiler, etc.) of the architecture.

The mix of computational models in Ptolemy proves to be a powerful means of modeling media applications, allowing for instance to intermix Kahn process networks with finite-state machines to model reactivity to external events in a streaming application. However, Ptolemy traditionally focuses on synchronous and statically scheduled approaches, which do not fit with the data-dependent and dynamic nature of Eclipse applications. Although partially addressed by Mescal, the lack of a path to RTL-level languages makes Ptolemy unsuitable to address the co-verification requirements of the Eclipse model.

Metropolis

Metropolis [6][56] defines a metamodel with precise semantics that designers can use to capture functionality, architecture, and the mapping of functionality to architectural elements. Metropolis builds on the idea of separation between functionality and architecture pioneered in POLIS [5] and VCC [21]. The metamodel allows modeling various computation and communication semantics in a uniform way. The metamodel is used to represent the function of a system being designed at all levels of abstraction, to generate executables for simulation, and as input to formal methods for both synthesis and verification in various design stages.

The Metropolis metamodel bears similarities to Ptolemy [19], SystemC [60], and SpecC [31], since all share the notion of a network of concurrent processes, communicating through channels and port interfaces. Process execution is defined as a sequence of events, which are a program's entries or exits to some piece of code. To represent an implementation's efficiency, each event can be annotated with a value that indicates the event's cost, such as the memory access delay on a read event. Associated with the channels is a set of constraints—written in logic formulas—that define the set of legal executions. Adapters can be inserted as part of the mapping specification to glue interfaces of different system components. This is conceptually similar to the architectural services in Cadence's VCC 2.0.

Metropolis aims to integrate a rich set of tools that each apply to a particular subset of the metamodel. Thereto, Metropolis defines a parser that reads metamodel designs and an API that lets developers analyze and modify those designs. For each tool integrated into Metropolis, a back end uses the API to generate required input by the tool from the design's relevant portion. Metropolis seems a promising approach for the design of heterogeneous multiprocessor architectures such as Eclipse, provided it integrates appropriate simulation (e.g. SystemC) and verification (e.g. RTL co-simulation) tools.

9 Eclipse simulation technology

The Eclipse architecture is modeled as a hierarchy of SystemC [137] *modules*, referred to as the *Eclipse simulation model* (Figure 39). The techniques presented below are not limited to Eclipse technology but are applicable to a wide range of architectures and can be regarded as extensions to the SystemC style of architecture modeling and simulation. The simulation model aims to serve as an environment for Eclipse application development and coprocessor design, as well as an executable specification of the Eclipse architecture for validation and hardware development. This chapter discusses generic mechanisms in the Eclipse model to realize this objective.⁸ Chapter 10 deploys the simulation model for a video decoding subsystem to validate key aspects of the Eclipse infrastructure.

Figure 39. High-level structure of the simulation model of an Eclipse instantiation. This example depicts two clients, each consisting of a coprocessor with its shell. The clients are connected to the transport network that includes a model of the shared memory. An abstract model of the host CPU executes the application control code that configures the shells.

Figure 39 shows the modules available in the Eclipse simulation model, namely client modules, the transport network module, and a CPU module. Each client consists of a behavioral model of a coprocessor and an instance of the detailed shell model. The transport network module contains an abstract model of the read/write bus arbiters and shared memory. The abstract CPU model executes application control software, and interfaces to the shells through an abstract DTL concentrator, the control target module. All modules communicate with each other through *channels* [137]. The channels encompass communication protocols and largely define the overall timing behavior. The media processor and the interconnection to the SoC environment are modeled at a high level of abstraction.

⁸ This chapter is largely based on Rutten [118]. Section 10.2.1 is reworked from [127]. Section 10.3 is based on [129].

9.1 Retargetable simulation

To support template instantiation, the Eclipse simulation model is reconfigurable in terms of the number and types of coprocessors, the sizes of stream tables, etc. Parameters are specified in a machine description file—here referred to as the *architecture setup file*—that is parsed at run-time by the simulation model. This allows system integrators to instantiate the Eclipse template without having to delve into the internals of the simulation models or recompile simulation code. Moreover, run-time retargetability allows system integrators to easily explore the design space spanned by the Eclipse template parameters.

Often, module parameters depend on each other or can only be initialized in a specific order. In order to make these ordering relations explicit, the Eclipse model discerns four distinct setup phases. The following list gives the setup phases in the order at which they are executed in each module.

1. **Construction.** The Eclipse module hierarchy of Figure 39 is (partly) constructed, and class instantiation factories register themselves (Section 9.1.3). Construction of modules that depend on architecture setup parameters is deferred to phase 2.
2. **Architecture setup.** In this phase, the Eclipse architecture template is instantiated, similar to when a generic architecture template is tailored to make a single point solution. The coprocessors with their shells are created, as their creation depends on the type and number of coprocessors needed in a specific Eclipse instance. The architecture setup file specifies the Eclipse instance (coprocessors, bus width, size of caches, etc.). The top-level module reads the architecture setup file and passes these setup parameters down the module hierarchy, allowing each module to parse the parameters for their own hardware settings. The architecture setup phase is implemented as repeated function calls, where each function call concerns only a single parameter.
3. **Initialization.** After the architecture setup phase, all modules and parameters are available. Parameter settings are compared to identify possible errors in the setup files. When all parameters are correct, the simulation modules are initialized, e.g. allowing them to create internal stream tables, etc. In addition, all Eclipse modules are connected via communication channels. Each hierarchical module connects its children. The connection of Eclipse modules is hard-coded in the simulator as part of the Eclipse template knowledge.
4. **Execution.** The simulation starts executing from simulation time 0.

This section shows how the modular setup of the Eclipse model is exploited to dynamically instantiate and configure modules in the architecture setup and initialization phases.

9.1.1 Distributed parameter parsing

The modules in the Eclipse simulation model of Figure 39 are highly parameterized. To easily maintain, extend, or reuse modules, we designed each module to be maximally independent from other modules. In the module hierarchy of Figure 39, each module is responsible for setting and maintaining its own parameters from the architecture setup file. However, to reduce complexity of the simulation model, we only parse the architecture setup file at one location in the code. Setting parameters in the Eclipse model is separated in

1. Lexical analysis of the architecture setup file;
2. Forwarding parameters through the module hierarchy;
3. Setting a parameter in the designated module.

Figure 40. Architecture setup file syntax description.

The parser is a simple lexical analyzer that does not interpret the meaning of the setup file. The parser generates name-value pairs called *tokenpairs*. The name or *token identifier* is a concatenation of the module name in the module hierarchy and the name of the setup parameter; the *token value* signifies the value of the identified parameter. The architecture setup file contains a list of such tokenpairs, as indicated in Figure 40.⁹

The simulator passes each tokenpair through the module hierarchy of Figure 39. For instance, the tokenpair

```
Dct.Shell.Ts.Tasktable.0.Enable : 1
```

sets the enable flag for the task at index 0 in the task table of the task scheduler in the shell of the DCT coprocessor. The token ID addresses the module that is supposed to absorb it. Each module absorbs its tokenpairs and forwards tokenpairs that are addressed to its children. Figure 41 shows such a sequence where the top-level Eclipse module passes a tokenpair to the DCT client module to set the size of the stream table (**NStreams**) in the stream synchronizer (SS). The DCT client consequently forwards the tokenpair to the coprocessor shell. The stream syn-

⁹ We chose to couple the format of the architecture setup file directly to the tokenpair format. To comply with standard practice, the architecture setup file can be specified in XML. However, switching to such a generic notation increases the verbosity of the architecture setup file as well as the complexity of the lexical scanner.

chronizer in the shell finally accepts the tokenpair and initializes its stream table to hold data for 8 streams. Thus, parsing of tokenpairs is fully distributed. Parse errors are generated in the same distributed way; each module signals an error when it does not recognize a token ID or when the value of the tokenpair is out of range. Defining a new architecture setup parameter only entails the addition of the parameter to the parsing of tokenpairs inside the module—other modules are not affected, not even the lexical scanner. Clearly, this interdependence between modules greatly improves the maintainability of the simulation model.

Figure 41. Setup sequence for setting the size of the stream tables in a DCT shell.

9.1.2 Shared template parameters

The Eclipse simulation modules are highly parameterized. In addition, the modules depend on many parameters that are derived from the parameters passed in the architecture setup file. Each module has its separate parameter class that is responsible for accepting parameter settings from the architecture setup file through the tokenpair mechanism, computing derived parameters, and verifying parameter settings wherever possible. As these activities are only needed in a simulation environment, the parameter classes are fully separated from the actual model of the hardware inside the modules. This separation also ensures that the modules cannot introduce errors by accidentally modifying parameter settings. To this end, modules only access their parameters as (compile time) constants during the execution phase.

Many parameters are shared by a set of modules. For instance, the width of the read bus needs to be known by both the memory and all shells, while other parameter values may be different for each instance of the shell. Sharing parameters ensures that each parameter is only declared once, and derived parameters are computed only once. Obviously, avoiding multiple declarations of the same parameter improves the maintainability of the code. In addition, having only a single definition of each parameter reduces redundancy in verifying parameter settings.

Figure 42 shows the implementation of shared parameters for the Eclipse, client, shell, and DTR modules. Eclipse parameters are shared over all clients (coprocessor-shell pairs), whereas client parameters are shared only between a specific co-

processor instance and its shell. Likewise, shell parameters are shared between all modules in the shell, while the DTR module in the shell defines parameters that are local to the DTR module only.

Figure 42. Sharing setup parameters: class diagram (left) and object diagram example (right).

```
class DtrParamC : public ShellParamC
{
public :
    unsigned int m_NLinesCache;    // public DTR parameter

    // Construction: obtain pointer to parent
    DtrParamC(const ShellParamC * const parent) :
        m_Parent(parent)
    {}

    // Architecture setup: parse tokenpairs
    void Setup(TokenIdC token_id, TokenValueC token_value) {
        if ("NLinesCache" == token_id)
            m_NLinesCache = token_value;
    }

    // Initialization: copy parent parameters and verify
    void Init(void) {
        m_NPorts = m_Parent->m_NPorts;    // Shared shell parameter
        assert(m_NLinesCache <= m_NPorts);
    }

private :
    const ShellParamC * const m_Parent;    // Read-only reference to parent
};
```

Figure 43. Pseudo code of the DTR Parameter class.

Parameters are declared as public variables of the parameter class at hand (Figure 43). This reduces the maintenance problem of updating both the parameter and its access function when a parameter definition changes. A module initializes its parameter object only after all parameters are set from the architecture setup file. During initialization, each parameter object copies the content of its parent parameter object. Thereto, each parameter object (e.g. `dtr_param`) obtains a read-only reference to its parent (`shell_param`) during creation. Inheriting a parameter

class from its parent class ensures that modules can access shared parameters without having to specify the actual location (DTR, Shell, Client, or Eclipse) of the parameter definition.

9.1.3 Dynamic addition of coprocessor modules

For a provider of independent SoC subsystems based on the Eclipse template, Eclipse coprocessors are typically developed by third parties and integrated by the subsystem provider. Coprocessor designers construct simulation models at various levels of abstraction before developing the actual hardware. This reduces the risk of redesign by validating coprocessors designs in a complete subsystem as early as possible. To ease validation of coprocessor modules, coprocessor designers should be able to plug in and execute their coprocessor modules with CPU, shell, and transport network modules without having to touch the internals of these Eclipse modules or modify the simulator.

Coprocessor modules are implemented as C++ classes. Instantiating a coprocessor means that the simulator must instantiate the specific coprocessor class. Normally, a C++ class called **MyCopro** is instantiated as follows:

```
MyCopro *my_copro = new MyCopro();
```

To instantiate the **MyCopro** class, the simulator must include the **my_copro.h** header file that contains the full class declaration of the coprocessor. Both for including the header file and for adding the coprocessor class name, the top-level Eclipse module code needs to be changed and recompiled. This section outlines a design pattern that avoids any changes or recompilation of this simulator code when adding a new coprocessor class.

Figure 44. Class diagram of the dynamic coprocessor creation pattern.

To allow such flexible addition of coprocessors, the construction of coprocessors is not hard-coded in the top-level Eclipse module. Instead, it recognizes a ‘**create**’ token ID from the architecture setup file, on which the Eclipse module uses a coprocessor creation *factory* (a class that provides an interface for creating instances of an associated class [51]), to instantiate a desired coprocessor module. Figure 44 illustrates the design pattern for the creation of coprocessors. The pattern allows the addition of coprocessors via the architecture setup file without the need to recompile the Eclipse simulation model. Note that this pattern is similar to the instantiation of components in Microsoft COM.

The Eclipse model is independent of the implementation of the coprocessors by accessing them only via the coprocessor interface functions of the **Coproc** class. The **Coproc** class defines a generic interface that is implemented by each specific coprocessor. Each coprocessor is instantiated as follows:

```
// Return pointer to generic Coproc interface
Coproc *my_coproc = new MyCoproc();
```

This is similar to the generic coprocessor-shell interface in the Eclipse architecture. The shell only accesses the generic **Coproc** interface, and has no knowledge of the specific **MyCoproc** implementation.

However, for instantiating the coprocessor classes, the Eclipse model still needs the coprocessor specific class definition from the **my_coproc.h** header file. This is circumvented by having a single factory (**DctFactory**, **RlsqFactory**, etc.) for each coprocessor that knows the implementation of the coprocessor and can create instances of the associated coprocessor via its **CreateCoproc()** function.

Each coprocessor model is compiled into a shared library. The simulator dynamically loads the coprocessor libraries specified by the user in the architecture setup file. The coprocessor factories are created automatically when the simulator loads the library. This is implemented in C++ with a static instance of the factory class in the coprocessor source (.cc) file (see also the ‘registry of singletons’ on page 130-131 of the singleton design pattern [51]). Each factory registers itself to a registry by name of the coprocessor when this static instance is constructed. This allows the top-level Eclipse module to access each factory by the name of the coprocessor via the **GetCoprocFactory()** function. The Eclipse model is separated from the details of each factory by only accessing them via the virtual **CreateCoproc()** function of the **CoprocFactory** interface:

```
CoprocFactory *f = CoprocFactoryRegistry::GetCoprocFactory("Vld");
Coproc *my_coproc = f->CreateCoproc();
```

This way, the Eclipse model is completely independent of the coprocessor and the factory implementation, and can instantiate multiple instances of a coprocessor by only referring to its name in the registry.

9.2 Mixed-level simulation

To comply with an iterative design approach as shown in Figure 36, the architect must be able to intermix architecture models of various levels of abstraction. In the design of the Eclipse architecture template, we developed near cycle-accurate models of the generic infrastructure early in the project. To evaluate this essential part of the architecture template, we created abstract behavioral models of a set of coprocessors for MPEG-2 decoding. Using behavioral models of the coprocessors reduced both development effort and simulation time with respect to cycle-accurate models. Thus, to efficiently execute a detailed specification of only a part of the architecture, the simulation environment must support mixed-level execution of models at different levels of abstraction. Figure 45 gives an example mixed-level simulation setup for Eclipse.

This section details our way of modeling Eclipse coprocessors and generic infrastructure at different levels of abstraction, followed by the introduction of interfaces and channels to enable mixed-level execution of Eclipse models for functional verification, performance analysis, and verification of RTL.

Figure 45. Mixed-level Eclipse simulation with an abstract behavioral model of a coprocessor and a near cycle-accurate model of its coprocessor shell. The task scheduler (TS) within the shell is modeled at the RTL level.

9.2.1 Architecture modeling

For architecture validation, the simulator provides timing information such as the execution time of a function or the delay in a communication protocol. A key aspect of intermixing architecture models is the communication between the modules that are modeled at different abstraction levels. To communicate between modules, an initiator typically sends a request and waits until the target responds with an acknowledge. There are various ways of implementing this waiting behavior in a module:

- Implement a full state machine in the target module that is called every clock cycle. While state machines are a good match for modeling communication protocols, writing a coprocessor that communicates with its shell as a state machine is complex and error prone.
- Apply lightweight multi-threading [82][137] to write sequential code instead of writing a full state machine. While this reduces the simulation performance, it allows the use of blocking statements in sequential code. This is done in all coprocessor models, allowing fast development of coprocessor models without being forced to generate complex state machines to handle the communication with the shell.

The use of multi-threading enables a mixed execution of abstract models and detailed models in a single environment. An abstract module can for instance model computation time of a certain operation by simply waiting for a parameterizable number of clock cycles and then execute the operation in zero simulated time [110]. To improve timing accuracy, a module can be gradually refined into a hierarchical module to reveal the internal structure and parallelism. The concept of ports and interfaces in SystemC allows such a transformation without affecting other modules. However, as this typically requires more threads and more simulation overhead in switching between the threads, such a refinement must be done only when necessary to significantly improve overall timing behavior. An event-based simulation environment [137] that suspends and resumes threads on a specified signal—such as an acknowledge—is crucial to avoid the simulation overhead of actively polling for the signal on each clock cycle.

9.2.2 Coprocessor modeling

Coprocessor development—that is, the development of simulation models of the application specific parts of an architecture—can start from a YAPI description, as outlined in Section 6.1. However, the translation from such a purely functional model to a model that includes timing and hardware constraints, such as multitasking with state save and restore, or synchronization of data accesses, is far from trivial.

The Eclipse simulator enables early simulation when such translation is yet to be done. Thereto, the Eclipse simulator provides a ‘YAPI coprocessor’, which encapsulates existing YAPI processes into an Eclipse coprocessor module (Figure 46). With the YAPI coprocessor, abstract application models written in YAPI can execute within the Eclipse simulator to obtain a first order estimate on for instance the performance of the architecture for a new application.

The YAPI coprocessor is multitasking, executing multiple different YAPI processes in their own thread of control. Thereto, each YAPI process is encapsulated in a lightweight thread called a *fiber* [82]. Assigning a thread to each YAPI process allows the application programmer to initially leave the responsibility of sav-

ing and restoring internal state of the process upon a task switch to the thread mechanism. The task scheduler in the coprocessor shell selects between tasks mapped on the YAPI coprocessor, similar to the behavior of a real Eclipse coprocessor. Therefore, the YAPI coprocessor assigns a task ID to each YAPI process mapped onto the coprocessor. The task ID returned on a `GetTask` request identifies the thread that the coprocessor must execute to resume the corresponding YAPI process.

Figure 46. YAPI coprocessor with forward in inverse DCT tasks.

The YAPI coprocessor translates the data structures communicated through YAPI write calls into unformatted sequences of bytes. Subsequently, such a byte sequence is encapsulated in a number of Eclipse packets. The YAPI coprocessor sends these packets to its shell in multiple Eclipse `write` requests. In addition, the coprocessor surrounds each YAPI write with `GetSpace/PutSpace` synchronization requests at the granularity of the requested data structure. On a read from a YAPI process, the coprocessor reads Eclipse packets from its shell and transforms these into the requested data structure. The coprocessor also recognizes and forwards location packets (Section 4.3) that may be present in the data stream, without affecting the YAPI process.

Evidently, more accurate simulation models are obtained by refining the YAPI read and write primitives into the Eclipse primitives to make synchronization and context switches explicit. Moreover, mixed execution of YAPI processes with Eclipse coprocessors necessitates a rewrite of the YAPI processes that communicate with the Eclipse coprocessor to match the communication behavior of the coprocessor. Eclipse coprocessors often multiplex data packets of different nature over a single stream and operate on a data granularity that does not match the granularity of input and output data of the YAPI process. Thus, the *part* of the YAPI process that communicates with the native Eclipse coprocessor must be transformed. Allowing such a gradual refinement closes the design gap from Eclipse tailored YAPI to the design of Eclipse coprocessors of Figure 29.

9.2.3 Performance analysis and functional verification

To cope with the requirements for performance analysis and verification of RTL models, the Eclipse simulation model uses SystemC threads [137] to model parallelism inside the modules and handshake (bi-phase) protocols to model inter-module communication. A significant part of the execution of the Eclipse model is spent on switching between threads and waiting on bi-phase protocol communication. However, for functional verification of coprocessor tasks and control software, timing behavior of the Eclipse instance is irrelevant and parallel threads and communication protocols can be largely avoided. Moreover, reducing the overall complexity by restricting functionality and parallelism facilitates debugging the Eclipse modules.

Despite the potential speedup in simulation time, we chose not to maintain separate models and code bases for application and hardware development. Clearly, maintaining separate models of considerable size (each around 10 to 30 thousand lines of C++ code) increases modeling effort and endangers integration of application software when extensions and bug fixes are made to only one of the two models. Therefore, a separate simulation mode within the code base of the Eclipse simulation model supports application development. The resulting two simulation modes are referred to as *sequential mode* for application development and *timed mode* for performance analysis and verification of RTL models.

If possible, functionality that is only needed in timed mode, such as bus arbitration, is bypassed in sequential mode. Although this requires additional code inside the modules to accommodate both modes, this can be minimized by a proper partitioning of the functionality into mode specific and generic functions. Moreover, the sequential mode bypasses all communication protocols by relying on interface method calls [137] between modules. The concept of interface method calls allows a module to be completely unaware of the inter-module communication implemented by the channels. The Eclipse model features both sequential and timed channels, as depicted in Figure 47, and Figure 48.

Figure 47. Sequential channel between coprocessor and task scheduler.

All modules communicate through ports. A port has an interface dedicated for the type of communication. For instance, a coprocessor communicates with the task scheduler in its coprocessor shell as follows:

```
task_id = TsPort->GetTaskBlocking(&task_info);
```

The port connects to a channel that implements the `GetTaskBlocking` function to handle the communication of the `GetTask` primitive with the task scheduler. This is a blocking call; the function call only returns when the task scheduler replies with a new task to execute on the coprocessor.

Figure 48. Timed channel between coprocessor and scheduler, split into coprocessor and task scheduler parts that are interconnected through signals.

In the sequential mode, the channel function directly triggers the task scheduler and blocks the coprocessor until the task scheduler replies. The task scheduler executes on the thread of the coprocessor. In timed mode (Figure 48), the channel function raises a request signal at the next rising clock edge. The request signal triggers the task scheduler. Here, the task scheduler runs on the thread of the `Ts-Coproc` channel. When the task scheduler replies with a next task, its channel only raises the acknowledge signal at the next rising clock edge. Upon receiving the acknowledge signal, the `GetTaskBlocking` function returns to the coprocessor.

Table 6. Simulation execution time for sequential and timed mode MPEG-2 video decoding.

Application	Sequence	Simulated clock cycles	Sequential execution	Timed execution
Single MPEG-2 decode	Tennis	3.7 M	123 sec.	329 sec.

Table 6 gives the overall simulation time for an MPEG-2 video decoding application in both timed and sequential mode. The competitive simulation speed of approximately a video frame per minute [69][93] in timed mode—comparable to the trace-driven simulation of the same MPEG-2 decoder on the TriMedia architecture by van der Wolf et al. [149]—stems largely from the use of abstract models for the coprocessors. The speed of the sequential simulation is relatively slow compared to state-of-the-art DSE environments. This is largely due to the detailed modeling (even in sequential mode) of the coprocessor shell. See also Chapter 11 for a discussion of the simulation speed.

9.2.4 RTL verification

While SystemC allows modeling the architecture at the RTL level, the industry standards for RTL-level modeling are VHDL and Verilog, driven by the availability of high quality synthesis tools. We chose VHDL to implement the Eclipse architecture at RTL level. Compared to Verilog, VHDL offers a wider range of options in using *generics*: design time parameters needed to implement the highly parameterized shell template.

Figure 49 shows the use of timed channels (Figure 48) to connect an abstract SystemC module to an RTL module in a co-simulation of SystemC and VHDL using the Cadence NCSysSystemC environment [20]. In such a co-simulation, all communication between the modules must be made explicit through SystemC signals at the level of individual wires. Therefore, the timed channel in Figure 49 provides individual ports for the request, acknowledge, task ID, and task info signals. While these ports are necessary for co-simulation with RTL modules, the timed channels of Figure 48 can omit these ports. This results in a single channel between coprocessor and task scheduler that contains an abstract—yet cycle accurate—implementation of the timing of the bi-phase protocol. We chose to explicitly implement ports and signals for both timed execution and RTL co-simulation to minimize modeling effort at the cost of a reduced simulation speed in timed execution.

Figure 49. Timed channel connects an abstract coprocessor module with an RTL-level task scheduler module through signals.

The main reason to provide co-simulation of SystemC modules and RTL modules is to be able to verify a newly developed RTL module against the SystemC module. Ideally, the unit tests of the SystemC modules can be reused as unit tests for the RTL modules in a co-simulation. While not all internals of the RTL module can be verified this way, the possibility to hook up an RTL module into a model of the complete system before all RTL modules are available greatly increases the chance of getting the RTL ‘first time right’.

10 Eclipse simulation experiments

This chapter presents a number of related experiments to validate the Eclipse template. The Eclipse template centers around the generic infrastructure formed for a large part by the coprocessor shells. Therefore, the experiments validate key aspects of the generic *infrastructure* described in Chapter 5. The detailed design and simulation of an Eclipse subsystem focuses on the *coprocessors*, which is outside the scope of the Eclipse template. Hence, this thesis does not use the simulation model to validate complete Eclipse subsystems. However, Part III of this thesis details three case studies of Eclipse subsystems, showing the applicability of Eclipse for three very different applications. These subsystems are evaluated in the typical way architectures are benchmarked in an industry setting: through back-of-the-envelope and spreadsheet calculations.

A secondary objective of the Eclipse architecture template—namely reuse of coprocessor hardware across subsystems with respect to hardware and development cost—is obviously not validated in this thesis. While the MPEG coprocessor study of Chapter 6 and the multitasking experiments of Sections 10.2 and 10.4 demonstrate the feasibility of reusing coprocessor hardware *within* one particular instance of the template, the value addition of reuse in a generic architecture template can only be proven once Eclipse is embedded in NXP’s SoC subsystem development process.

Figure 50. Eclipse simulation toolchain.

The generic infrastructure is validated with an MPEG video decoder. The complexity of the application—due to e.g. the extreme data dependencies, execution of multiple streams at different resolution and frame rates, and the different granularity of operation per coprocessor—make it an ideal test bench. The timing of

the simulation models itself is verified by hand against the detailed design of the coprocessor shells. In addition, the experiment of Section 10.2.1 is fully computed by hand to verify the simulation model. The complexity of the other experiments allows only result estimation to assess the correctness of the simulations.

Figure 50 depicts the Eclipse simulation toolchain used in the validation experiments of this chapter. The Eclipse simulation model, depicted as a solar eclipse, takes the architecture setup file as input to instantiate the parameterizable Eclipse template. The application mapped onto the Eclipse instance is fed into the simulation as application control software executing on a model of the control processor. Additionally, the user can control the simulation through command-line options, and select the different simulation modes. Outputs of a simulation run are textual debug traces and waveform traces of communication wires to debug and analyze the simulation execution in detail. Additionally, the simulator outputs a trace file for performance measurement. This is fed into an independent viewer—aptly named *Telescope*¹⁰—to visualize the dynamic behavior of the tasks, coprocessors, and data streams [53][44].

10.1 MPEG-2 video decoding behavior

This chapter analyzes the behavior of the MPEG-2 video coding Eclipse instance, as described in Section 6.3. The coprocessors in the simulation model can only decode MPEG-2 video. The extension of the coprocessor models to include encoding and coding of other formats is postponed to a follow-up project in cooperation with a lead customer.

The Eclipse coprocessors and memory are assumed to execute at a modest 150 MHz. With a frame rate of 25 Hz, this results in 6 million clock cycles available to decode one frame. The simulation model mimics the behavior of an output display unit with a dual frame buffer. Whenever both entries in the frame buffer are full, the decoding pipe stalls. When a frame is not decoded in time for display, the display unit repeats the previous frame. In this simulation, the coprocessors are sufficiently fast to decode each frame before its display deadline. Only the first two frames are decoded immediately, all other frames are only decoded after the display unit frees space in the frame buffer. Figure 51 depicts three MPEG-2 video sequences that will be used for as simulation input throughout this chapter:

teeny.m2v: 50 frames/second interlaced, 720 pixels by 576 lines. This sequence has an open GOP structure with N=10 and M=2. The bitstream starts with the following pictures: **IPB**BPBPB**IPB**BPBP, etc. (the frames that are analyzed later in this chapter are typeset in bold). Note that the first group-of-pictures (GOP) does not contain a B-picture after the first I-frame, as the B-picture de-

¹⁰ Telescope is superseded by TimeDoctor. The TimeDoctor performance visualizer is implemented in the open source Eclipse framework (not to be confused with the Eclipse architecture of this thesis) and freely available on <http://www.sourceforge.net/projects/timedoctor>

depends on the last P-picture of the previous GOP. The pictures that are typeset in bold will be used for analysis later in this chapter. Teeny is a standard-definition sequence shot from a stationary camera, while the girl is moving her head downwards. The background is blurred with the colors shifting over time. The low amount of detail and motion makes this a relatively easy scene to decode.

tech.m2v: 50 frames/second interlaced, 720 pixels by 576 lines. This sequence has an open GOP structure with $N=12$ and $M=3$, i.e., a bitstream starting with IPBBPBBPBB**IBBPBBP**, etc. Tech is a standard-definition sequence with the camera panning from right to left while a cross-country skier moves from right to left in the video. The panning combined with the detail in the images makes this a difficult stream to decode.

oslo.m2v: 50 frames/second interlaced, 1920 pixels by 1088 lines. This sequence is a high-definition sequence. The sequence is pans diagonally from the left bottom to the right top. The individual movement of the people in the audience combined with the detail in the flowers in the foreground makes this a highly difficult stream to decode.

Figure 51 Simulation MPEG-2 video sequences. From left to right: Teeny, Tech, and Oslo.

All three sequences are encoded using a variant of the EMPRESS encoder [15]. The encoder requires several (around five) pictures to build up a consistent motion vector field. To avoid these run-in effects of the encoder, we analyze the first pictures of the second GOP in the Teeny and Tech sequences. The Oslo sequence is only used in the budget analysis of Section 10.4. Moreover, focusing on the second GOP allows the Eclipse model to reach a steady state during the decoding of the first GOP.

10.1.1 Decoding behavior for the Teeny sequence

To get a global insight into the behavior of the system, Figures 52 through 56 show the input buffer fillings of each coprocessor task for decoding the Teeny sequence. These simulations are performed with prefetching caches in the coprocessor shells, which are subject of Section 10.2. The pictures show the buffer filling as known by the shell of the respective coprocessor. The buffer information is updated by local **PutSpace** actions and incoming synchronization messages from the

remote shell. Buffers are sized as follows: VLD input: 512 byte, RLSQ inputs: 512 byte each, DCT input: 1024 byte, MC pixel block input: 1024 byte, and 512 byte for the other MC inputs.

Figure 52 shows that the DCT task forms the bottleneck in the system for I-pictures; the DCT input buffer is almost full while its output is close to empty for the duration of the video frame. No motion compensation is needed for I-pictures, hence, the MC does not have a large workload. The DCT simulation model pipelines the read stage and execution of the horizontal DCT. The subsequent vertical DCT and write stages are also pipelined. The throughput of the DCT is largely defined by the read and write stages. The DCT reads 8 by 8 blocks of 12-bit DCT coefficients. The DCT blocks may also be smaller than 8 by 8, due to dynamic block compression in the RLSQ coprocessor (Section 6.3.2). This variable input block size causes the jagged edges of the DCT input buffer filling in the picture.

Figure 52. Input buffer fillings (in byte) for the intra-coded (I) picture in the second GOP of the Teeny bitstream. The horizontal axis gives the simulated time (in clock cycles) between the start and completion of decoding the picture, as measured at the MC coprocessor. From top to bottom: VLD bitstream input; RLSQ run-length pairs input; RLSQ picture properties input; DCT coefficient input; MC pixel block input; MC sequence properties input; MC macroblock properties input.

The VLD input is always full. In the simulation setup, the input data is read from file by an input coprocessor with zero execution delay. The VLD outputs macroblock headers to the MC, and run-level pairs to the RLSQ. The RLSQ blocks

on writing DCT coefficients to the DCT, causing the RLSQ input buffer to fill up and block the VLD.

On average, the DCT input buffer contains 870 samples, corresponding to approx. eight to nine DCT blocks (one DCT block is $8 * 8$ coefficients * 12/8 byte plus a three-byte header; in total 99 byte). The DCT operates at a block granularity, while the MC operates at a macroblock granularity. The DCT output buffer contains on average approx. 370 samples, corresponding to approx. five blocks of 8 by 8 pixels of 9-bits each ($8 * 8$ pixels * 9/8 byte plus a 3-byte header = 75 byte). The MC typically requires six coded blocks to process one macroblock. Hence, the MC regularly stalls on reading pixels blocks from the DCT, which causes the macroblock properties input buffer to fill up and occasionally block the VLD.

MC macroblock headers for the MC have a fixed size of 50 byte. The MC macroblock properties input contains at least 4 to 5 macroblock headers. The MC's progress is limited by the availability of macroblock data from the DCT.

Both RLSQ and MC only commit reading of picture and sequence properties when the properties change. During processing of (macro)blocks, the tasks leave the information in the respective input buffers for later reference.

Figure 53. Coprocessor input buffer fillings (in byte) versus the simulation time (in clock cycles) for the first predictive-coded (P) picture in the second GOP of the Teeny bitstream.

Figure 53 depicts the input buffer fillings for a predictive-coded (P) picture in the second GOP of the Teeny sequence. Apart from intra-coded macroblocks in I-pictures, P-pictures mostly contain inter-coded macroblocks with or without motion vectors. This increases the workload on the MC coprocessor to fetch macroblocks from reference pictures in off-chip memory via the SoC infrastructure, and combine these with the difference blocks from the DCT.

For intra-coded macroblocks, the MC input contains a 50-byte packet with macroblock properties as well as a second 50-byte packet with the motion vector. At cycle 66,350,000, the MC property input contains four macroblocks, directly obtained from the VLD. The DCT output contains five to six pixel blocks, with around nine DCT blocks at its input. The RLSQ adds roughly two macroblocks to the two macroblocks in flight at the DCT. Thus, the total of roughly four macroblocks in the RLSQ and DCT pipe matches the four macroblocks present in the path from VLD to MC. Despite the close coupling between the tasks, the system is well balanced for this frame, with the largest load shifting between VLD, DCT, and occasionally the MC.

The DCT input shows 36 peaks in this frame. The peaks correspond to the start of a new MPEG slice (576 pixels per line with 16 pixels per macroblock gives 36 slices per frame). At the start of a new slice, the VLD must decode slice properties, in close cooperation with the MPEG header task on the coprocessor that models the CPU. This causes a very brief underflow in both the RLSQ and MC macroblock properties input. The lack of run-level pairs in the RLSQ input ripples through to the DCT input.

Figure 54. Number of coded blocks per macroblock (top) and the run-level packet size per macroblock (bottom) as output by the VLD for all macroblocks in the first P-picture of the second GOP of the Teeny bitstream.

Figure 54 shows the number of coded blocks per macroblock as output by the VLD, as well as the combined size of the run-level packets transmitted from the VLD to the RLSQ. The number of coded blocks has a clear correspondence with the behavior at the DCT input in Figure 53. With little coded blocks, the workload of the DCT is relatively low. The pattern in the run-level packet sizes is re-

flected in the small variations superimposed on the large fluctuations in DCT input of Figure 53. Here, the embedded block compression reduces the DCT load for blocks with little detail.

The first 10-14 macroblocks and the last 1-2 macroblocks in each slice have little coded blocks, corresponding to the blurred background on the left and right of the girl's head in the Teeny sequence. MPEG-2 encoders typically search for candidate motion vectors in the upper-left region of the current macroblock, with a search space of around 128 pixels. For the upper-left macroblocks in the picture, the picture boundary limits the search space. Although the lack of spatial references may result in a low coding efficiency, the minimal number of coded blocks (Figure 54) shows that the temporal references for the blurred background of the sequence seem to provide a good match.

Figure 55 shows a P-picture of the Teeny sequence in the *first* GOP. Compared with the P-picture in the second GOP, the first GOP places a higher load on the coprocessors. The EMPRESS encoder takes several pictures to build up a consistent motion vector field. The resulting lack of good temporal references increases the number of coded blocks.

Figure 55. Coprocessor input buffer fillings (in byte) versus simulation time (in clock cycles) for a predictive-coded (P) picture in the first GOP of the Teeny bitstream.

Figure 56. Coprocessor input buffer fillings (in byte) versus simulation time (in clock cycles) of the second bidirectionally predicted (B) picture in the second GOP of the Teeny bitstream.

Figure 57. Characteristics of the second B-picture in the second GOP of the Teeny bitstream. Vertical axis: number of coded blocks per macroblock (top) and the run-level packet size per macroblock (bottom). Horizontal axis: successive macroblocks in the picture as output by the VLD.

Figure 56 shows the input buffer fillings for a bidirectionally predicted (B) picture in the Teeny sequence. The load on the MC—fetching two macroblocks from

reference pictures in external memory for each bidirectionally predicted macroblock—increases, as clearly visible in the DCT output.

On the other hand, the workload of the DCT decreases, with less coded blocks and smaller block sizes (Figure 57). A potential way to reduce the large variation in the coprocessor execution is to increase parallelism by increasing buffer sizes. The VLD to MC and VLD-RLSQ-DCT paths basically operate at a macroblock granularity. Increasing buffer size in the VLD to MC path may give the RLSQ and DCT more breathing space to operate independently.

10.1.2 Decoding behavior for the Tech sequence

Figures 58 through 61 give the input buffer fillings for I-, P-, and B-pictures of the Tech sequence. The differences with the Teeny sequence clearly show the data dependent behavior of MPEG-2 decoding. The increased complexity of the Tech stream reflects on the behavior of the coprocessors.

Figure 58. Coprocessor input buffer fillings (in byte) versus simulation time (in cycles) for the intra-coded (I) picture in the second GOP of the Tech bitstream.

Figure 58 gives the input buffer fillings for the second I-picture of the Tech sequence. The picture is decoded in approximately a factor 7.5 faster than real-time, as can be seen from the horizontal axis in the figure. The minima in the RLSQ input correspond to a high load on the VLD, having to decode a large number of run-level pairs. The dips correspond to image regions with a high level of detail.

The first minimum centered at cycle 60.4 million corresponds to the audience above the skier. The subsequent maximum corresponds to the snow and billboard in the image. The minimum from 60.5 to 60.75 million corresponds to the audience below the skier. The last slices match with the snow at the bottom of the image, and are relatively easy to decode. The length of the minima in the DCT and RLSQ input are extended by the dynamic block compression used in the RLSQ, giving the RLSQ and DCT extra time to keep up with the VLD's production of new run-level pairs.

Figure 59. Coprocessor input buffer fillings (in byte) versus simulation time (in cycles) for the first predictive-coded (P) picture of the second GOP in the Tech bitstream.

Figure 60. Characteristics of the first P-picture in the second GOP of the Tech stream.

Figure 59 depicts the input buffer fillings for the first P-picture in the second GOP of the Tech sequence. Figure 60 shows the characteristics of this P-picture. The large number of coded blocks reflects the complexity of the Tech sequence.

Figure 61. Coprocessor input buffer fillings (in byte) versus simulation time (in clock cycles) for the second bidirectionally predicted (B) picture in the second GOP of the Tech bitstream.

Figure 62. Characteristics of the second B-picture in the second GOP of the Tech bitstream. Vertical axis: number of coded blocks per macroblock (top) and the run-level packet size per macroblock (bottom). Horizontal axis: successive macroblocks in the picture as output by the VLD.

Figure 61 and Figure 62 give the input buffer fillings and characteristics of the second B-picture in the second GOP of the Tech bitstream. We clearly see the buffer filling of the RLSQ input and the MC macroblock properties input increase when the DCT cannot keep up. For macroblocks with a reduced number of coded blocks, the system is in balance. Surprisingly, the B-picture contains more coded blocks and run-length pairs than the P-picture that was used for the prediction.

10.2 Cache control

Caches are often the most complex modules of any processor design. The following analyses aim to verify the cache coherency and prefetching/preflushing strategy of the read and write caches of Section 5.2. Note that while the caches in the Eclipse coprocessor shell allow arbitrary access patterns, the simulation results are based on our multiprocessor MPEG implementation that exhibits a largely linear access pattern for inter-task communication.

For the analysis of the caches with the MPEG-2 video decoder application, we focus on the caches of the RLSQ and DCT coprocessors, as these pose the most stringent demands on the cache. To generate a high overall load, we focus on the first GOP of the Tech sequence. However, to minimize simulation time, we limit the cache simulations to the standard-definition Tech and Teeny sequences. The high-definition Oslo sequence is used in the budget analysis of Section 10.4.

10.2.1 Cache behavior for producer-consumer

In a first study into the behavior of the read and write caches, we analyze the ‘ideal’ producer-consumer setup of Figure 63. The setup concerns just one stream, transferred from a producer to a consumer task. The producer task runs on a DMA style coprocessor that produces data packets onto the Eclipse network. The consumer task runs on a similar coprocessor that reads data from the Eclipse network. These coprocessors only exist as simulation models to read or write data from/to file.

The producer task generates four data packets of different sizes and synchronizes via `PutSpace` requests at the end of each packet. The consumer task synchronizes its consumption of packets at the same granularity. In this example, the producer writes five packets of respectively 130, 130, 130, 98, and 2 bytes. Each packet consists of a 2-byte header, and a variable-length payload. The last 2-byte packet contains a location ID that indicates the termination of the data stream generated by the producer. In total, the producer writes data packets of in total 490 bytes in 125 writes (5 writes for packet headers plus 120 writes to write the packet payload in units of 4 bytes).

Figure 63. Producer and consumer tasks mapped onto a producer and consumer coprocessor, and communicating through a buffered data stream mapped into shared memory.

There is no bus contention; the producer coprocessor is the only client on the write bus, and the consumer coprocessor is the only client on the read bus. Furthermore, the producer and consumer strictly write and read in a streaming fashion. Both coprocessors have 32-bit data ports to their shell, on which they issue 16 and 32-bit **Read** and **Write** actions. It is the task of the shell to perform address generation, and the coprocessors are unaware of memory alignment issues; the 16- and 32-bit accesses might occur on an unaligned address. The memory bus is 128 bit wide. The shell caches are sized in this example to contain only two ‘memory words’ of 128 bit each. Thus total cache size per shell is 32 bytes.

The caches reduce the number of read and write operations on the shared memory and issue only aligned memory reads/writes. For synchronization, the coprocessors issue **GetSpace** and **PutSpace** requests to the shell. The shell hardware uses these to perform automatic cache coherency control and optionally prefetching and preflushing for streaming data. Prefetching and preflushing is intended to hide the access latency to the shared embedded memory.

With optimal prefetch/preflush behavior, and no additional transfers for cache-coherency, the application would induce no more than 31 (490 byte/16 byte memory words) bus transfers per task. Table 7 shows the actual cache behavior. Despite the tiny cache sizes, the (shell of the) consuming coprocessor experienced *not a single* read cache miss as result of the automatic prefetch, although some reads had to wait a few cycles for the prefetch to complete. As a result, the cache and its prefetching strategy clearly reduce the average read latency.

For the producing coprocessor, the cache allocates on a write-miss (does not fetch). Correspondingly, the preflush here empties dirty cache words to memory, so that the allocate *not once* had to delay for first flushing a dirty cache word. Table 7 also shows that the preflush only marginally improves write latency, as the pipelined implementation of the shell fully hides the latency of a cache flush upon a **Write** or **PutSpace** request. Even without a write cache, the write latency is minimal due to the write buffer in the shell that hides the bus arbitration latency.

The effect of the preflush will only become visible when there is a high load/contention on the bus.

Table 7. Producer-consumer copy simulation results.

Parameter	No caches		Caches, no prefetch/preflush		Caches, prefetch/preflush	
	Prod.	Cons.	Prod.	Cons.	Prod.	Cons.
# Read misses/ write flushes	-	-	26	33	0	0
# Read/write memory words	139	141	35	33	35	34
Read/write memory bytes	490	2256	490	528	490	544
Read/write latency (avg/max/min cycles)	3/4/3	10/12/10	3/5/3	5/12/3	3/4/3	3/11/3

Figure 64. Simulation results of producer and consumer actions without caches in the coprocessor shells.

Figure 64 depicts the behavior of the copy application when the coprocessor shells do not contain caches. The simulation output shows that the producer immediately starts writing data, as the stream buffer is initially empty. As shown in Figure 64, the consumer only starts reading data after the first commit from the producer; the task scheduler correctly blocks the consumer `GetTask` request until there is more than zero data in the data stream.

Figure 65. Simulation results of the producer task, using a write cache (without automatic preflush).

Figure 66. Simulation results of the consumer task, using a read cache without automatic prefetch.

Initially, the consumer reads data at a lower rate than the producer due to the large read latency. After writing three packets, the producer task stops on a failing **GetSpace** request. Only after the consumer task has freed up buffer space with a **PutSpace** request, the producer shell's scheduler reactivates the producer.

The read latency of 12 cycles occurs whenever the consumer issues a read request that spans two memory words. Here, the split-transaction memory bus allows the two read requests to be issued directly after each other to the memory arbiter. Similarly, the producer write latency of 4 cycles occurs when a write request from the coprocessor spans two write words.

Figure 65 and Figure 66 show the producer and consumer behavior with read and write caches in the shells, but without prefetch and preflush functionality. Compared to Figure 64, the figures clearly show reduction in memory access achieved by the read and write caches. The consumer empties the buffer faster than in the uncached situation. As a result, the producer does not stall on unavailable room in the buffer. The producer must wait for a write-back of a dirty cache word upon every fourth write action. Additionally, a **PutSpace** request by the producer causes the write cache to flush any partially filled cache word.

Figure 67 shows the behavior of the producer when deploying a write cache and enabling preflushing in the producer coprocessor shell. On each write request that extends beyond a single cache word, the producer's shell (pre)flushes the fully filled cache word to memory.

Figure 68 shows the consumer's behavior, utilizing a read cache with prefetching enabled. On the first **GetSpace** request, the read cache of the consumer shell prefetches two bus words (not visible in Figure 68 as the first read memory address is 0x0). For subsequent read requests, the read cache prefetches new data ahead of the current read pointer whenever the consumer reads the last byte of a cache word—and thereby effectively marks the cache location as unused.

When the consumer reads beyond the last cache word that lies fully within the space granted by **GetSpace**, the read cache may still perform a further prefetch. The consumer shell knows the amount of available data in the buffer, kept up to date via synchronization messages received from the producer shell. However, the read cache only samples the data availability information in the consumer shell upon a **GetSpace** request to restrict the effort of checking for invalidated cache words to **GetSpace** actions. The read cache uses this sampled data availability information for prefetching. The read cache does not start further prefetches when the prefetch address extends beyond the known available space (sampled upon the last **GetSpace** action), as the cache would fetch potentially invalid data. Figure 68 shows that read actions do not trigger new (prefetch) bus transactions just before every new **GetSpace** action.

Figure 67. Simulation results of the producer task, using a write cache with automatic preflush.

Figure 68. Simulation results of the consumer task, using a read cache with automatic prefetch.

A subsequent `GetSpace` request may cause the cache word (from which the coprocessor read last) to be invalidated. This last word may be filled only partially with valid data. For the remainder of the data, the read cache cannot determine whether this data is valid (written by the producer before the word was fetched), or invalid (the producer writes the data after the consumer has fetched the word). To keep the administration in the cache to a minimum, the cache invalidates the complete word.

For each packet, the consumer task first inquires two bytes space for the packet header, followed by a `GetSpace` on the complete packet size. Figure 68 shows that the first `GetSpace` causes the invalidate/prefetch action as the producer task produced new data while the consumer task consumed the last packet¹¹. The figure shows that two (prefetch) bus transactions are started upon the `GetSpace` action: one for the invalidated cache word, and one prefetch that was not started on an earlier read action because of insufficient data availability (based on the read cache's information from the previous `GetSpace` action).

The following `GetSpace` does not trigger any invalidates. The consumer task takes ample time to read the packet header: insufficient for the producer to write and commit new data.

The producer and consumer closely match in execution speed. The absence of data-dependencies allows the consumer task to keep a steady one-packet delay with respect to the generation of new data by the producer task. We conclude that the total of three invalidates are redundant as the distance between the write pointer (producer) and the read pointer (consumer) always is larger than one bus word. The design choice of a simple implementation in the shell by only invalidating cache words upon `GetSpace` actions may result in missed opportunities for prefetching and cause redundant invalidates. However, these are almost fully compensated by the additional prefetching upon `GetSpace` actions.

10.2.2 Cache behavior for single-stream MPEG decoding

To show the effectiveness of the caches as part of the generic coprocessor shell, we decode an MPEG-2 stream on the architecture depicted in Figure 76 on page 161. The reference pictures for motion compensation are accessed from off-chip memory and do not pass through the shell. The cache sizes vary with the number of input and output streams to the coprocessors. Input and output streams are each assigned a cache line of two bus words.

Table 8 gives the frame completion times for a standard-definition MPEG-2 stream (`tech.m2v`). The table shows that the prefetching caches—despite their tiny sizes—significantly reduce the overall execution time, with an average re-

¹¹ As can be seen by the monotonously increasing read address in Figure 68, the buffer can hold all transported data without wrap-around. Thus, apart from the last read data word, there will never be other invalid data words in the cache.

duction in decoding time of 27% for I frames, 36% for P frames, and 44% for B frames.

Table 8. Cache influence on frame completion times at the MC coprocessor for single-stream MPEG-2 decoding.

Nr.	Frame type	Frame start at MC	No caches	Caches, no prefetch / preflush	Caches, prefetch / preflush
1	I	0	1,670,398	1,252,992	1,219,931
2	P	Completion of frame 1	3,049,212	2,288,971	2,135,462
3	B	12,000,000	13,263,329	12,938,816	12,704,532
4	B	18,000,000	19,206,467	18,903,091	18,674,307
5	P	24,000,000	25,301,683	24,970,384	24,831,062
6	B	30,000,000	31,194,870	30,891,440	30,675,872
7	B	36,000,000	37,222,936	36,910,397	36,680,396
8	P	42,000,000	43,301,525	42,963,741	42,814,208
9	B	48,000,000	49,252,474	48,931,254	48,696,557
10	B	54, 000,000	55,246,427	54,924,816	54,693,687

Table 9. Single-stream MPEG-2 decoding simulation results for caches with prefetch/preflush enabled. The cache lines contain two words each.

Parameter	DCT		RLSQ		
	Coefficients input	Pixel output	Run-length pairs input	Picture prop. input	Coefficients output
# Reads/writes	1240102	2413504	1463926	558	1600727
# Read misses/write flushes	34	0	10162	0	0
# Read/write memory words	1437001	1514822	581031		1648334
Read/write bytes	22154554	20018876	7607992	4526	22154080
Read/write memory bytes	22992016	20018876	9296496		22154080
Read/write latency (avg/max/min cycles)	4/14/3	3/3/3	4/14/3	4/14/3	3/3/3

Table 9 gives detailed results for single-stream MPEG-2 decoding with prefetching caches. The results concern decoding of all 31 frames of the `tech.m2v` stream. Both the DCT and RLSQ encounter cache misses on the primary input. For instance, cache misses occur for instance in the following sequence of events:

- The coprocessor reads a 2-byte packet header. These two bytes are located at the end of the first cache word. As the last bytes of the cache word are read, this read triggers a prefetch to refill this cache word with future data.

- After reading the header, the coprocessor issues a **GetSpace** command to acquire space for e.g. writing the output packet. The **GetSpace** fails due to insufficient room in the output buffer.
- Upon the failing **GetSpace**, the coprocessor completes the processing step, after the subsequent next **GetTask** action, it re-starts by re-reading the 2-byte header. This causes a read miss, as the cache data is already overwritten by the above prefetch.

A second, more complicated sequence of events that leads to a cache miss is the following:

- The coprocessor reads a 2-byte packet header. These two bytes are located at the end of the first cache word. As the last bytes of the cache word are read, this read triggers a prefetch.
- After reading the header, the coprocessor issues a **GetSpace** command to acquire space for the payload of the input packet. In this case, the **GetSpace** causes an invalidate of the previously prefetched data in the first cache word. The **GetSpace** invalidate subsequently triggers a new prefetch for this location. The prefetch address is based on the current read pointer as maintained in the shell.
- The coprocessor reads the packet payload, using the **offset** field of the read command. Hereby, the coprocessor reads further than the shell anticipated when starting up the invalidate prefetch. In this situation, the read extends into the second cache word, without triggering a proper prefetch for the first cache word. This results in a read miss when a next read accesses the first cache word.

The above situation occurs when the coprocessor uses the **offset** field of the read command after a **GetSpace** action that is not directly preceded by a **PutSpace**. The **GetSpace** actions cause cache words to be invalidated when the buffer is almost empty, *i.e.*, the producer can hardly keep up with the consumer on the buffer.

Figure 69 depicts the distribution of read latency for all coprocessors, with prefetching caches in the coprocessor shells. For most coprocessors, the average read latency is 4 cycles. The differences in read latency are mostly the result of the pipelining of different sequences of coprocessor shell commands. We describe a number of scenarios in which the different cycle latencies occur to give a feeling for the interaction with the coprocessor and its shell.

3 cycles. This is the minimal possible read latency. The coprocessor read hits in the cache and accesses a single cache word. There are no pending prefetches or invalidate actions.

4 cycles. A 4-cycle read latency occurs typically when a **GetSpace** action precedes the read request. The shell checks if the read cache words require invali-

duction before accepting a new read request. The read requests hits in the cache.

6 cycles. A succession of read actions may cause this behavior. The first read action reads the last data from the first cache word and triggers a prefetch. The second read action hits in the next cache word. A third read action hits on the pending prefetch of the first cache word, resulting in a 6 cycle latency when there is no bus contention.

Figure 69. Read latency distribution (in cycles) for single-stream MPEG-2 decoding with prefetching caches in the coprocessor shells. From top to bottom: VLD bitstream input; RLSQ run-length pairs input; RLSQ picture properties input; DCT coefficient input; MC pixel block input; MC sequence properties input; MC macroblock properties input.

9 cycles. A read action reads the last data from the first cache word and triggers a prefetch. A subsequent read action accesses two cache words. One hits in the cache, while the second hits on the pending prefetch. Another situation is where the coprocessor triggers **GetSpace** actions on the input and output ports. The first **GetSpace** triggers a prefetch. A subsequent read hits on the pending prefetch of the first **GetSpace** action.

11 cycles. A **GetSpace** action triggers a prefetch. A subsequent read hits on the pending prefetch.

12 cycles. A **GetSpace** action checks for invalid cache words and prefetch candidates. The shell only accepts the next read command after the **GetSpace** is fully handled. The read misses in the cache and must wait for a memory transfer.

14 cycles. A **GetSpace** action succeeding a **GetTask** action triggers prefetches for both cache words in the cache line. A following read command hits on the pending prefetch for the second word, and hence must wait for bus access of the first prefetch action.

The above analysis shows that the design of the coprocessor strongly influences the timing behavior. The smallest read latencies can be obtained by long sequences of read commands, as implemented in the DCT coprocessor.

10.2.3 Cache behavior with shared cache lines

Section 5.2.1 details how the caches in the shell allow to share cache lines between different task ports. This is useful when a task only infrequently accesses a second task port, e.g. to read in configuration information. Sharing a single line between the regular streaming port and the configuration port reduces hardware cost, yet only minimally affects performance.

Table 10 and Figure 70 show the simulation results for the RLSQ with separate caches for each of the two inputs, and a single shared cache line for both inputs. The number of cache misses and read latency does not increase with the shared cache line. The coprocessor issues a **GetSpace** command before reading either run-length or picture property data. The shell invalidates cache words and starts up prefetches upon the **GetSpace** command, such that subsequent read actions will typically hit in the cache, but may have to wait for a pending prefetch.

Table 10. Simulation results of separate and shared cache lines for the RLSQ inputs.

Parameter	Separate cache lines		Shared cache line	
	Run-length pairs input	Picture prop. input	Run-length pairs input	Picture prop. input
# Reads	1,463,926	558	1,463,925	558
# Read misses	10,162	0	10,167	0
# Read memory words	581,031		581,092	
Read bytes	7,607,992	4,526	7,607,990	4,526
Read memory bytes	9,296,496		9,297,472	
Read latency (avg/max/min cycles)	4/14/3	4/14/3	4/14/3	4/14/3

Note that the picture property data and run-length data are much larger than the tiny cache line size. Thus, only the first read actions must wait for completion of the prefetch triggered on **GetSpace**. Subsequent reads on the same stream will have full benefit of the cache.

Figure 70. Distribution of read latency (in cycles) as seen from the RLSQ coprocessor. The shell deploys a single cache line that is shared for both input ports. Top: RLSQ run-length pairs input. Bottom: RLSQ picture properties input.

Table 11. Simulation results for dual-stream producer-consumer copy with prefetching caches.

Parameter	Producer		Consumer	
	Stream 1	Stream 2	Stream 1	Stream 2
# Reads/writes	125	74	125	74
# Cache misses/flushes	0	0	0	0
# Read/write memory words	56		55	
Read/write bytes	490	287	490	287
Read/write memory bytes	777		880	
Read/write latency (avg/max/min cycles)	3/4/3	3/4/3	3/11/3	3/11/3

10.2.4 Cache behavior for dual-stream producer-consumer

Table 11 gives the simulation results for a dual-stream producer-consumer copy application. The producer and consumer coprocessors each execute two different tasks in a time-shared fashion. The producer shell’s write cache allocates a 2-word cache line that is shared by the two output streams of both producer tasks. Equivalently, the consumer shell’s read cache contains a single 2-word cache line

that is shared by the two input streams of both consumer tasks. The table shows that the sharing of cache lines between tasks hardly affects the read and write behavior (i.e., compared with Table 7). Despite the sharing of cache lines between tasks, the number of cache misses/flushes remains zero.

Figure 71 and 72 show the behavior of the consumer tasks. The pictures show that the first `GetSpace` action after a `GetTask` invalidates the cache line and triggers prefetches to ensure that subsequent reads will hit in the cache.

10.2.5 Cache behavior for dual-stream MPEG decoding

To show the behavior of sharing cache lines among two tasks that execute on the coprocessor in a real-life application, we decode two independent MPEG-2 streams in parallel. The RLSQ, DCT, and MC coprocessors execute two tasks in time-shared fashion, one task for each of the two streams. The VLD coprocessor is instantiated twice as the simulation model does not implement multi-tasking with corresponding state save and restore for the VLD coprocessor. The VLD header tasks execute as separate tasks on a RISC CPU in the VLD architecture. In the simulation model, each header task is mapped to a separate instantiation of a dedicated MPEG header coprocessor.

Table 12. Frame completion times (clock cycles) at the MC coprocessor for dual-stream MPEG-2 decoding.

Nr.	Frame start at MC	No caches		Caches, no prefetch / preflush		Caches prefetch / preflush	
		Tech	Teeny	Tech	Teeny	Tech	Teeny
1	0	5,768,159	5,559,106	4,128,322	3,979,083	2,712,936	2,613,141
2	Completion frame 1	9,760,699	9,439,198	6,974,830	6,738,633	4,689,057	4,507,861
3	12000000	15,322,505	15,032,431	14,343,381	14,190,100	13,615,376	13,506,987
4	18,000,000	21896431	21,787,856	20,741,958	20,680,140	19,899,809	19,857,546
5	24,000,000	27,418,064	27,106,290	26,350,295	26,172,229	25,694,123	25,553,581
6	30,000,000	33,772,782	33,637,197	32,594,550	32,513,039	31,815,310	31,758,147
7	36,000,000	39,214,352	38,943,997	38,237,514	38,079,740	37,574,401	37,468,719
8	42,000,000	45,669,903	45,381,948	44,550,219	44,362,759	43,782,421	43,647,620
9	48,000,000	51,203,901	50,898,350	50,306,122	50,110,385	49,582,524	49,450,159
10	54,000,000	58,338,023	58,588,182	57,086,564	57,271,755	56,100,192	56,236,330

Table 12 shows the frame completion times of decoding of two standard-definition MPEG-2 streams. The frames of each stream are decoded at roughly the same time, which underlines the effectiveness of the medium function grain of the coprocessors. All frames are decoded well before the display deadline, set to 6 million clock cycles for both 25 Hz streams, assuming the coprocessors operate at 150 MHz. Table 12 shows that the prefetching caches consistently improve frame completion times.

Figure 71. Consumer task 1 behavior for dual-stream producer-consumer copy with prefetching caches.

Figure 72. Consumer task 2 behavior for dual-stream producer-consumer copy with prefetching caches.

Table 13. Simulation results for dual-stream RLSQ with prefetching caches.

Parameter	Run length pairs input		Picture properties input		Coefficients output	
	Tech	Teeny	Tech	Teeny	Tech	Teeny
# Reads/writes	853,046	446,142	1,906,218	1,905,948	963,756	627,623
# Read misses	1,641	162	0	0	0	0
# Read/write memory words	3,042,220				1,647,142	
Read/write bytes	4,573,828	1,812,137	15,461,546	15,459,356	13,332,735	8,600,051
Read/write memory bytes	48,675,520				21,932,786	
Read/write latency (avg/max/min cycles)	4/14/3	5/14/3	4/16/3	4/16/3	3/3/3	3/3/3

Table 13 details the simulation results for the RLSQ coprocessor. In this setup, the shell reserves separate two-word cache lines for each of the two RLSQ inputs. The simulation stops when the 19 frames of the second stream (Teeny) are decoded.

Compared with Table 9, we see that the dual-task RLSQ reads significantly more data from its picture properties input than in the single-task setup. Upon a task switch, the RLSQ starts with reading the picture properties for the stream of the newly selected task. By leaving the picture property data in the input stream, the RLSQ avoids having to store the picture properties internally in the coprocessor. The RLSQ only commits (**PutSpace**) reading of picture properties when the picture properties change.

Figure 73. Distribution of memory read latency (in cycles) as seen from the coprocessor shells in dual-stream standard-definition MPEG-2 decoding with prefetching caches.

Figure 73 gives the latency of a memory read as seen from the shell of each coprocessor. The figure gives an overview of the load on the read bus. The figure shows that the delay caused by bus contention is minimal. Here, the separation of read and write buses combined with the split-transaction behavior on the read bus

minimizes the latency of a read access. Figure 74 shows the latency of read requests as seen from the coprocessor tasks. As Figure 73 suggests, the distribution in read latency is only minimally influenced by the bus load, but largely caused by the sequence of actions triggered by the task.

Figure 74. Distribution of the latency (in cycles) of coprocessor read requests as seen from the coprocessor tasks in dual-stream standard-definition MPEG-2 decoding with prefetching caches.

Especially for the MC tasks, the read latency increases significantly from the single to the dual-task setup. The 10-cycle read latency occurs for example on the first read of macro block properties after a task switch. The preceding `GetSpace` triggers a prefetch for both cache locations to replace the data of the previous task in the cache. The read action must wait for the pending prefetch. Upon execution, the MC task starts reading macro block properties. The MC starts with reading the macro block properties to ensure sufficient time to access the reference pictures from the SoC infrastructure. Next, the MC reads the header of the pixel block data from the DCT. From the header, the MC task tries to read the pixel block data. Whenever there is insufficient data in this input buffer, the MC returns control to the shell's task scheduler, which dispatches the next task. Here, the mechanism to avoid state and re-read macro block properties at every new task execution causes a large amount of data switching in the read cache. In a product implementation of the MC coprocessor, the MC's behavior must be augmented to read only the information from the macro block properties that is needed to decide if the task can continue or must yield to the scheduler.

Figure 75. Dynamically scaling down horizontal resolution at an MPEG-2 frame boundary using location *lds*. The figure shows buffer filling (in byte) versus simulation time (in cycles).

10.3 Dynamic reconfiguration

This section presents a case study of the dynamic reconfiguration mechanism as described in Sections 4.3 and 5.5. Chapter 6 describes an Eclipse subsystem for video compression, supporting MPEG-2 decoding of multiple streams. The RLSQ, DCT, and MC coprocessors of Figure 76 on page 161 support horizontal and vertical downscaling during MPEG decoding to allow multi-window and picture-in-picture functionality. The case study shows a mode switch from full-screen to dual-window mode, by dynamically scaling down the horizontal resolution during decoding of a video stream.

Figure 75 shows the task behavior when the user switches from full screen to dual-window mode. To accomplish this mode switch for the main video path, we scale down the horizontal resolution by setting a scale factor in the RLSQ, DCT, and MC tasks. We use the location mechanism to effectuate the switch in each task just before decoding a new group of pictures (GOP) to ensure a smooth transition. Application control for the mode switch consists of the following steps.

1. Program the shell of the RLSQ, DCT, and MC coprocessors to suspend the active task when the coprocessor recognizes and reports a ‘start of GOP’ location to its shell.
2. Command the VLD task to generate a ‘start of GOP’ location ID at the start of the next GOP. This command can be issued to the VLD by setting a task parameter in the shell of the VLD coprocessor.
3. Wait for a location match interrupt from the RLSQ task.
4. Upon the interrupt, write the new horizontal scale factor as task parameter to the RLSQ shell. In this setup, we scale down horizontal resolution by a factor 2.
5. Enable the RLSQ task by setting the enable flag for the RLSQ task in its shell.
6. Repeat step 3 to 5 for the DCT and MC tasks.

Figure 75 shows the number of bytes read at RLSQ, DCT, and MC inputs for the Teeny sequence.

Cycle 7,327,435. The VLD task generates the ‘start of GOP’ location packet at all outputs.

Cycle 7,327,628. The RLSQ receives the ‘start of GOP’ location packet at its primary input and reports the location to its shell. The shell interrupts the control processor.

Cycle 7,327,679. The RLSQ task continues with the new horizontal scale factor received from its shell. It starts by re-reading quantization and scan matrices for the new frame from its auxiliary input. The delay in execution is caused by the VLD task that must first parse picture and sequence properties before sending new run-length pairs to the RLSQ task.

Cycle 7,329,133. The DCT task reports the ‘start of GOP’ location to its shell. The shell interrupts the control processor.

Cycle 7,329,184. The DCT continues with the new horizontal scale factor received from its shell. This is clearly visible from the size of the read requests at the DCT input.

Cycle 7,329,507. The MC task receives the ‘end of GOP’ location packet at its macroblock property input and reports the location to its shell. The shell interrupts the control processor.

Cycle 7,329,558. The MC task continues with the new horizontal scale factor received from its shell. The increased rate of processing after the mode switch reveals that the tasks need less time to complete a processing step for the new I-frame.

10.4 Budget-based task scheduling

This section analyzes the behavior of the budget-based task scheduling implemented in the coprocessor shells (Section 5.3.1). We decode three MPEG-2 video streams in parallel, one high-definition stream (Oslo), and two standard-definition streams (Tech and Teeny). In this use case, we model a situation where the high-definition stream is rendered on the main window, and the two standard definition streams are rendered as picture-in-picture windows. The high-definition stream must be rendered with the highest quality. This means that the decoder must decode all high-definition frames before the display deadline.

Table 14 gives the frame completion times for each stream with two budget settings in the shell. The simulation stops when all 19 frames of the Teeny stream are decoded. Shaded entries in the table indicate that the corresponding frame did not meet the display deadline. The table shows that this Eclipse instantiation does not have sufficient performance to decode two SD streams and one HD stream in real time at 150 MHz coprocessor speed. The instantiation was designed for real-time decoding of two HD streams. The coprocessors must be designed with pipelined read/write and compute stages to achieve the required performance. The simulation models do not implement coprocessor pipelining, resulting in a lower performance.

Table 14. Frame completion times (clock cycles) at the MC coprocessor for triple-stream MPEG-2 decoding with prefetching caches in the shells. Two budget settings: budget 1 to obtain round-robin scheduling for all tasks, and budget 7k cycles for the high-definition (HD) stream to ensure real-time behavior of the HD frames. The frames that are too late for the 6M cycle deadline are shaded.

Nr.	Budget 1 cycle/task for all tasks			Budget 7x1024 cycles/task for Oslo		
	Tech	Teeny	Oslo	Tech	Teeny	Oslo
1	3,772,631	3,627,178	9,324,982	7,170,038	6,899,272	5,319,633
2	6,845,864	6,326,890	15,072,655	12,080,633	11,465,452	9,927,874
3	14,764,235	14,188,766	23,291,040	21,637,847	20,774,814	16,216,948
4	20,822,851	20,634,807	29,226,296	28,631,003	28,345,439	22,412,224
5	26,777,019	26,192,950	35,857,171	33,629,578	32,807,551	28,867,159
6	32,741,023	32,502,725	41,103,780	39,645,598	39,308,371	34,354,430
7	38,570,355	38,071,286	47,249,106	45,882,174	45,226,760	40,235,027
8	44,814,425	44,361,472	53,687,349	51,657,965	50,976,031	47,332,759
9	50,594,601	50,065,803	59,668,134	57,591,787	56,742,517	52,206,678
10	57,010,354	57,422,157	65,404,592	65,734,821	66,601,727	58,469,949

The left columns in Table 14 show a default setup in which all tasks are dispatched in round-robin order. Each task gets an equal share of the coprocessor. This does not match with the workload differences of HD and SD streams, resulting in missed deadlines for the HD stream. The right column addresses this problem with a budget setting in which the HD stream gets a seven times larger share

of the coprocessor than the SD streams. The workload for an HD stream is approximately a factor 6 larger than for an SD stream based on the difference in picture resolution. A factor 6 difference in budget settings between HD and SD streams gives a proper distribution of coprocessor time in relation to the workload. However, as MPEG-2 decoding is inherently data-dependent, the workload varies per individual frame. To ensure timely decoding of all HD frames, we increase the HD budget to a factor 7 higher than the SD budget.

The absolute value of the budget controls the rate of task switching between streams. In principle, the choice of setting the budget in multiples of 1024 is a trade-off between decode times for each stream and the overhead of switching between streams. However, the budget of 1024 cycles per task is very small with respect to the overall execution time of an MPEG frame. Hence, this budget setting has no negative effects on the real-time behavior of the system. Table 15 shows the reduction of task-switch overhead with a larger budget setting than the 1-cycle/task budget of the two-stream setup of Section 10.2.5, Table 12. The larger absolute budget value clearly reduces the number of switches between streams. The larger budget reduces the amount of cache invalidate and state restore actions (e.g., re-reading of picture properties in the RLSQ coprocessor).

Table 15. Frame completion times for dual-stream MPEG-2 decode with prefetching caches. The table compares frame completion times with a budget of 1024 cycles/task and the 1 cycle/task budget of Table 12.

Nr.	Frame start at MC	Budget 1 cycle/task		Budget 1024 cycle/task	
		Tech	Teeny	Tech	Teeny
1	0	2,712,936	2,613,141	1,683,386	1,611,996
2	Completion of frame 1	4,689,057	4,507,861	3,016,415	2,753,535
3	12,000,000	13,615,376	13,506,987	13,238,124	12,983,012
4	18,000,000	19,899,809	19,857,546	19,265,993	19,158,063
5	24,000,000	25,694,123	25,553,581	25,230,960	24,978,896
6	30,000,000	31,815,310	31,758,147	31,257,848	31,126,478
7	36,000,000	37,574,401	37,468,719	37,213,372	36,960,991
8	42,000,000	43,782,421	43,647,620	43,269,197	43,045,964
9	48,000,000	49,582,524	49,450,159	49,233,994	48,949,721
10	54,000,000	56,100,192	56,236,330	55,370,154	55,490,209

11 Discussion

The irregular, data-dependent behavior of Eclipse applications calls for simulation. Eclipse simulation serves multiple purposes.

- *Design space exploration.* A key question for design-space exploration of Eclipse is the interaction between the distributed task scheduling of coprocessor and DSP tasks. To answer this question, the accuracy of the modeled timing behavior of the communication network (coprocessor shell, pipelined bus) is critical. The execution behavior, and therefore the timing behavior, of Eclipse tasks depends on the timing with which buffers are filled and emptied. For instance, a failed inquiry may cause a task to reload previously read data (Section 4.2.2). A similar dependency holds for the scheduling of Eclipse tasks by the task scheduler in the coprocessor shell. While trace-driven simulation seems best suited for design-space exploration due to its high simulation speed, trace-driven simulation cannot cope with dependencies between application and architecture. Moreover, the required effort to accurately model the timing behavior of the communication network—as well as a shift of focus towards coprocessor-only Eclipse subsystems—led us to omit high-level design space exploration and focus on a more detailed analysis of the Eclipse infrastructure.
- *Application development.* Three different aspects are important in the development of Eclipse applications. For functional verification, we need a highly efficient model that provides data transport, synchronization, and task scheduling services through the Eclipse primitives. To verify application control software, this model must also support Eclipse’s reconfiguration capabilities. For application timing analysis, we need a cycle-accurate model of the hardware. Finally, both models must support gradual refinement of application tasks from YAPI to the Eclipse coprocessors. The reconfiguration requirement led us to develop an abstract model for functional verification that resembles the structure of the hardware. To avoid maintaining separate code bases, we gradually augmented this model with timing information for cycle-accurate simulation.
- *Architecture validation.* In terms of validation, the Eclipse simulation model was developed as a proof of concept for potential customers. The key question was the behavior of the distributed synchronization and distributed scheduling approach. The experiments in Chapter 10 validate these aspects of the Eclipse infrastructure. Feasibility of complete Eclipse subsystems is subject of Part III.
- *RTL verification.* An abstract cycle-accurate simulation model can be used to verify RTL modules in a system context. For verification of RTL modules, co-

simulation can only address errors visible at the interfaces of the modules. We verified the shell's RTL design using RTL-level test modules. We only briefly exercised a SystemC-RTL co-simulation using SystemC-based coprocessors and the shell's RTL model. Nevertheless, the development of the simulation model clearly helped to improve the design of the shell and verify the functional correctness. Given the complexity of the shell with its caches and high level of pipelining, the simulation model proved essential to guarantee functional correctness and refine the design.

Eclipse simulation focused on application development and architecture validation. Additionally, the simulation model aimed to serve as executable specification for RTL design. However, the largest impact of building an abstract, yet near cycle-accurate simulator was that it forced us to fill in the details of the Eclipse architecture from a system-level perspective. This resulted in a set of detailed specification documents of the architecture, as the basis for development of RTL and device drivers.

A conclusion from the development of the Eclipse simulation model is that creating cycle-accurate, yet abstract models is a daunting task—particularly due to the amount of parallel and pipelined processing in the communication network. The risk is that the designer cannot abstract from the internal structure of the modules, and is effectively writing RTL-like code. Especially for the caches in the coprocessor shell, this resulted in excessive modeling effort and low simulation speed.

We conclude that for complex architectures such as Eclipse, the simulation model should not constitute the specification. While it accurately models the interfaces of the architecture modules, the abstract implementation of the module's internal structure cannot represent the module's actual structure and timing. For complex modules—such as the read and write caches in the shell—the architect must equip the RTL designer with a specification of the internal structure and timing behavior to ensure a correct and efficient implementation.

SystemC 2.x is rapidly becoming the de-facto standard for system-level modeling, with a highly active community of users and tool vendors. When the Eclipse simulation model was developed, SystemC 2.0 did not exist yet, and Philips Semiconductors design flow was based on TSS. The requirements for abstract modeling and retargetability led us to develop an event-based model with lightweight threads, modules, and communication channels that strongly resembles today's SystemC approach. With respect to SystemC 2.x, the main contribution of Chapter 9 is a user's perspective on how to construct parameterized modules and support mixed-mode simulation.

We developed a clean object-oriented software architecture that is generically applicable for simulation models with many parameter settings. Our approach complemented SystemC with dynamic instantiation of modules from a setup file. Hereto, we separated instantiation time from the actual execution of the models.

The separation in several phases during the setup of the models resulted in a clean design of the simulation models and a very high flexibility.

The highly parameterized design of the simulation model makes it suitable for design-space exploration (DSE) of a specific subsystem to tune cache and memory sizes, use of prefetching or not, etc. Further exploration of coprocessor designs, e.g. to analyze a pipelined versus non-pipelined coprocessor implementation, still requires a large effort to accurately model the coprocessors. While the ‘YAPI coprocessor’ of Section 9.2.2 closes the gap between YAPI and Eclipse coprocessors from a modeling and verification perspective, the automatic translation from YAPI to Eclipse primitives is too simple for an accurate timing analysis. State-of-the-art DSE environments are needed to enable the architect to quickly model and analyze different coprocessor implementations.

Section 9.2 discusses how we can combine a behavioral and cycle-accurate model in a single code base by replacing communication channels and setting computation delay values. This approach works well when we model the functionality of each module in a highly abstract way, for instance by instantly computing the result of a request but delaying the answer with the estimated compute time. However, when we need to model pipelines for correct timing behavior, this adversely affects the complexity and simulation speed of the behavioral model. As a result, the simulation speed of the behavioral model is relatively slow with respect to state-of-the-art DSE environments.

Chapter 10 discusses simulation experiments with the MPEG subsystem as defined in Chapter 6. The MPEG execution analysis shows that the Eclipse coprocessors can already execute a well-balanced schedule with as little as 5 kByte of buffer memory for the Eclipse streams. The cache experiments show that the explicit synchronization primitives allow very efficient use of a micro cache that is specifically designed for streaming. The dynamic reconfiguration experiment shows that even though the coprocessors are hardwired, the coprocessor shells allow a large flexibility in the properties and structure of the graphs. For multiple MPEG streams of different resolution, the task scheduler in the Eclipse shells offers sufficient hooks to control the execution of the coprocessors for the different streams.

The experiments also have taught us that the detailed behavior of the coprocessors—and therefore the accuracy of their simulation models—strongly influences the system performance. Similar to tuning a program to the capabilities of the compiler, the coprocessors can exploit the pipelining in the coprocessor shell by a smart ordering of shell operations. While the prefetching mechanism in the shell reduces average read latency, the chosen implementation negatively affects the potential worst-case read latency. Deploying the Eclipse caches in applications with more stringent real-time constraints than targeted by the Eclipse design requires a careful review of the Eclipse caching concepts. Such analysis is outside the scope of this thesis.

Part III

Eclipse case studies

Rob Rutten's toes, as published in full color in the astronomy journal Zenit. Martijn's head provides the necessary contrast to view the crescent sun in projection onto the beach of Salina Cruz, Mexico, during the solar eclipse of July 11, 1991.

12 Video coding subsystem

This chapter describes a video coding subsystem that served as the initial driver for our design of the Eclipse template. This subsystem targets SoC platforms for high-definition television functions such as NXP's Nexperia digital video chips. The original requirements formulated by the Nexperia SoC architects defined an accelerator subsystem attached to a TriMedia processor. The subsystem should be able to simultaneously decode two high-definition (HD) MPEG-2 streams, or encode one standard-definition (SD) MPEG-2 stream in parallel with decoding multiple SD MPEG-2 streams. Various combinations should be possible, such as decoding one HD stream and decoding two SD streams in parallel, or transcoding to implement time-shift functionality. To be commercially competitive, the total area of the subsystem should not exceed 10 mm² in CMOS18 technology.

Figure 76. Eclipse instance for MPEG decoding and encoding.

12.1 MPEG coding subsystem

Figure 76 depicts the MPEG coding subsystem. Section 6.3 details the design of the coprocessors. The coprocessors are multitasking and weakly programmable. Thus, a single DCT coprocessor can time-share both the forward and inverse DCT functions of one or more MPEG encoding applications and the inverse DCT of one or more decoding applications. Equivalently, the run-length scan and quantization (RLSQ) coprocessor performs the run-length decoding, inverse scan, and inverse quantization of the MPEG-2 decoding network shown in Figure 3, as well as its encoding variant: quantization, zigzag scan, and run-length encoding. The motion compensation/motion estimation (MC/ME) coprocessor has a dedi-

cated connection to the system bus to access MPEG reference frames in off-chip memory. Similarly, the variable-length decoding (VLD) coprocessor fetches the incoming compressed bitstreams from off-chip memory.

The DSP-CPU executes audio decoding, variable-length encoding, and demultiplexing in software. In addition, the DSP-CPU configures the set of applications executing in the subsystem by programming the stream and task tables in the shells through the control bus (not shown in Figure 76).

12.1.1 Memory bandwidth

Van Eijndhoven et al. [34] estimate the required memory bandwidth for MPEG-2 video decoding and encoding, and DV decoding. They show that the targeted applications allow the use of a single on-chip memory (SRAM) for communication buffering with a wide data path (128 bits) to provide the necessary bandwidth. Figure 77 gives an estimate of the bandwidth requirement per stream for single-stream HD decoding (MP@HL).

Figure 77. MPEG-2 video decoding task graph.

For single-stream HD MPEG-2 video decoding, we estimate a total bandwidth requirement on the local bus of 611 MByte/sec and minimally 324 MByte/sec on the SoC infrastructure.

- s0* The VLD accesses the MPEG bitstream directly from the SoC infrastructure. We assume 10 MB/sec as a worst-case number of the average bandwidth of an HD MPEG stream. The peak bandwidth for this stream may be significantly higher.
- s1* This stream contains the quantization matrices and sequence, picture, and macroblock properties. We estimate a worst-case bandwidth requirement of less than 1 MByte/sec.
- s2* We estimate 50 MByte/sec for the run-length pairs.
- s3* This stream contains the motion vectors and sequence, picture, and macroblock properties. A single HD stream has 8160 macroblocks per frame (1920×1088 pixels/frame / 16×16 pixels/macroblock). We estimate a bandwidth of 8 MByte/sec based on 32 bytes per macroblock (30 frames/sec * 8160 macroblocks/frame * 32 bytes/macroblock).
- s4* This stream contains the DCT coefficients. Worst-case, each macroblock contains 6 coded DCT blocks. At 30 frames/sec, this gives approximately 1.5M DCT blocks per second (30 frames/sec * 8160 macroblocks/frame * 6 DCT

blocks/macroblock). This results in a bandwidth of 141 MByte/sec assuming 12 bit coefficients ($1.5 \text{ MBlocks/sec} * 64 \text{ DCT coefficients/block} * 12 \text{ bits/coefficient}$).

- s5 For the pixel blocks, we estimate a bandwidth of 106 MByte/sec assuming 9 bits/pixel ($1.5 \text{ MBlocks/sec} * 64 \text{ pixels/block} * 9 \text{ bit/pixel}$).
- s6 We optimistically estimate 220 MByte/sec for fetching the prediction data, assuming that the 17×17 luminance data and the 9×9 chrominance data of a macroblock can be fetched from memory without overhead. The actual bandwidth will be higher, depending on the memory organization for the frame data and the types of memory accesses supported by the memory interface. Caching in the MC may limit the average bandwidth requirement. For storage and display of the frame data, we estimate an additional 94 MB/sec ($1.5 \text{ MBlocks/sec} * 64 \text{ pixels} * 1 \text{ byte/pixel}$).

The Eclipse infrastructure, instantiated for this particular applications, uses a separate read and write bus of 128 bits each. The buses are pipelined, allowing a new request to be accepted every cycle. The coprocessors and buses operate at 150 MHz. The local memory operates at the double frequency of 300 MHz, allowing it to serve both the read and write requests every bus cycle. With 2.4 GByte/sec each, the buses can easily accommodate the bandwidth requirement for dual HD decoding of 611 MByte/sec per bus.

12.1.2 Coprocessor performance

The standard CCIR601 ML/MP MPEG-2 decoder is somewhat conservatively estimated at 3 GOPS based on software implementations on the TriMedia, MAP1000, Pentium, and SUN MAJC cores. The MPEG subsystem is designed to handle two HD streams concurrently ($1920 \times 1080 @ 30 \text{ Hz}$). These streams have about six times the SD resolution, and also about six times the input bitrate (25 Mbit/sec). For two such streams, the estimated computational performance is roughly 36 GOPS ($2 \text{ streams} * 6 * 3 \text{ GOPS per SD stream}$) on mostly 16-bit data items.

For the Eclipse coprocessor performance, the DCT poses the largest compute challenge. Per DCT block, the DCT coprocessor has 50 cycles available for dual-stream decoding at a 150 MHz clock frequency ($150 \text{ MHz} / (2 \text{ streams} * 1.5 \text{ MBlocks/sec})$).

A previous hardware implementation shows a full 8×8 two-dimensional DCT takes 64 cycles, or 32 cycles if the horizontal and vertical DCT are pipelined. Krishnan [86] gives more detailed performance figures for a DCT with embedded block compression. Assuming a best-case latency of 4 cycles to read a DCT row via the shell, reading a full DCT block takes 32 cycles ($8 \text{ rows} * 4 \text{ cycles/row}$). Clearly, this is not fast enough for worst-case dual-stream decoding. Thus, the

DCT coprocessor must pipeline the read and write stages with the compute stage to meet the required performance.

12.1.3 Silicon area

Our initial estimates indicate that the instance takes less than 7 mm² of silicon area in 0.18 micron CMOS technology. The shell area is based on a detailed spreadsheet in which we count all internal registers in tables and interfaces for each shell instance with proper parameter settings for stream and task tables, cache sizes, and data widths. Area for logic is assumed equal to the area occupied by the registers. The sum of all shells is 0.8 mm², assuming a layout efficiency of 80%. The estimate includes 1.7 mm² for a 32-kByte on-chip memory and 2.0 mm² for a programmable VLD coprocessor, but excludes the DSP-CPU. We estimate total coprocessor area at roughly 4 mm², as detailed in Section 6.3 (2.0 mm² VLD + 0.9 mm² DCT + 0.3 mm² RLSQ + 0.6 mm² MC).

12.1.4 Power consumption

All coprocessors will be synthesized for operation at 150 MHz. The on-chip SRAM operates at 300 MHz to support separate read and write data buses, each running at 150 MHz. We estimate total power consumption at less than 240 mW for simultaneous decoding of two HD MPEG streams, excluding the power needed for the off-chip data traffic to the MPEG reference frames. A strong contribution is expected from the dual bus that connects all shells to shared SRAM memory. A conservative estimate of 3 mm bus length with the target width of 2 times 128 bits has shown to easily meet the speed targets [54], and would consume about 180 mW when operated at 100% bus load, including bus drivers, wires, bus receivers, and assuming a worst-case signal transition rate. As the dual HD application consumes not more than half the peak bus bandwidth of 4.8 GB/s, this results in roughly 90 mW dissipation for worst-case bus traffic. A 32 kByte KFSRAM consumes about 100 mW when operated at 300 MHz, needed to support both the read and write bus each running at 150 MHz. For half the used bandwidth, 50 mW of power remains. For the MIPS processor, which dominates the VLD coprocessor area, the power specification is 30 mW at 75 MHz. For the internal shell operation and the other coprocessors (DCT, RLSQ, MC/ME), we do not have specific power figures yet. As estimation, we extrapolate the same power density of the MIPS. This results in an additional $(0.8 + 0.9 + 0.3 + 0.6) \text{ mm}^2 * 15 \text{ mW/mm}^2 = 39 \text{ mW}$. Extrapolating to the double clockrate of 150 MHz, assuming a high load on all coprocessors, results in about 70 mW. Total power consumption is summed up to $90 + 50 + 30 + 70 = 240 \text{ mW}$.

12.2 Conclusion

Eclipse, and its initial instantiation for MPEG coding, is designed for memory-to-memory functions, relaxing the real-time requirements to soft real-time behavior. The focus of the design is on flexibility while allowing execution of demanding applications such as dual-stream HD decoding. The shared memory allows a flexible allocation of communication buffers and hence supports many combinations of encoding and decoding applications with different buffer requirements. Separating the local memory from the SoC-level infrastructure enables high-bandwidth communication within the Eclipse subsystem by using a wide memory and memory bus.

In the original design of the MPEG subsystem, the TriMedia processor (DSP-CPU in Figure 76) could benefit from direct access to the Eclipse shared memory whenever the Eclipse subsystem was not in use. Communication between software tasks on the TriMedia and tasks on the Eclipse coprocessors requires large buffers to average the bursty nature of TriMedia execution. The choice of a single shared memory allows to allocate such large buffers at any part in the processing chain. Thus, it further increases flexibility by allowing software tasks to interact with any of the Eclipse coprocessor tasks.

While the Eclipse video-coding subsystem was originally designed to be an extension to the 64-bit TriMedia processor, the integration has not been worked out in detail. Subsequent studies focused on a more autonomous subsystem in which the MIPS in the VLD processor takes care of the control software to program the shells. Chapter 14 presents a recent approach for interaction between software tasks on a DSP and the hardware tasks of an Eclipse subsystem.

The video coding subsystem of this section forms a good benchmark for high-end applications. The area estimation shows the shells take up 0.8 mm^2 in CMOS18 technology, roughly 11% of the total subsystem area. Compared to a fully dedicated implementation of an MPEG2 decoder, the shells can be considered as the overhead of deploying Eclipse. Obviously this is still a pessimistic estimate, as even a fully dedicated implementation will need some infrastructure to interconnect parts. The benefit of the additional area for the shells is the added flexibility to intermix decoding and encoding of multiple streams at different resolutions.

The original customer of the Eclipse template was the TriMedia department within Philips Semiconductors. When the results of the Eclipse project became available, the TriMedia department had spun out into a separate company called TriMedia Technologies (TTI). Philips Semiconductors explicitly forbade transfer of Eclipse results to TTI, as Eclipse was seen as an important asset for Philips. Within Philips Semiconductors, video compression activities were split over encoding and decoding departments at different sites in Europe. Unfortunately, neither department was inclined to join forces to productize an Eclipse subsystem that could handle both encoding and decoding functionality.

13 Image processing subsystem

At present, televisions sport progressive displays. The television or storage systems (such as DVD recorders) must transform interlaced into progressive video. This transformation process, de-interlacing, can be done in many ways, for instance through incorporating motion and/or edge information. The regular nature of the pixel-based algorithms and the limited flexibility requirements allow for cost-effective implementations in dedicated hardware.

This chapter presents a business case for Edge-Dependent De-Interlacing (EDDI) for effectively removing jagged edges from de-interlaced video. The EDDI algorithm [63] is a relatively cheap algorithm since it does not require expensive frame buffers. As such, it is suitable for low-cost DVD recorders and similar systems.

The individual components of the algorithm can be reused for different purposes, such as noise filtering (peaking). We discuss an Eclipse subsystem implementing EDDI in efficient and flexible manner. The scalable Eclipse template allows the subsystem to be extended with other blocks—for instance for peaking and 2D scaling—while reusing the infrastructure and existing EDDI blocks.

13.1 Edge dependent de-interlacing

The simple median filter is a popular algorithm in the economy market for television and storage ICs offering de-interlacing [62]. The result of a median filter is adequate for image regions without edges. For slanted edges in the image, directional interpolation significantly improves image quality. With directional interpolation, the image data is filtered in the direction with the highest correlation, i.e., along edges, in an attempt to prevent blurring that results from interpolation across edges. The EDDI algorithm ensures that directional interpolation is only attempted in edge areas to avoid artifacts. For all other features in the image, we employ a vertical temporal median filter.

EDDI operates only on luminance data. We split the algorithm in three steps: edge filtering and gradient computation, edge detection, and edge-dependent de-interlacing. We improve the edge orientation detection by preprocessing the input video signal with a vertical high-pass filter. The edge filter is cascaded with a three-point horizontal averaging filter. The low-pass filter lowers the sensitivity of EDDI for steep angles and reduces the noise sensitivity. The filter also computes the gradient of the luminance signal from the image points around each pixel.

The edge detector searches for monotonously descending or ascending edges in successive image lines of the pre-filtered video signal. The position of the edge is determined by a zero crossing in the luminance values of successive horizontal pixels. The width of the monotonous region defines the width of the edge. For each edge point in a first line, the edge detector temporarily stores the edge position, span of the monotonous region, and the gradients. During edge detection in the second image line, the detector searches for matching edge points in both lines. If an edge is found, it outputs the edge position, span, edge width (defined as the number of pixels between the two edge positions), and direction of each edge.

The de-interlacer performs vertical temporal median filtering. For edge areas, the de-interlacer uses the edge information for directional interpolation along the edge. Shallow-angled edges give wide to interpolation regions. Artifacts occur if such a region crosses another object in the image. To avoid artifacts, the de-interlacer mixes interpolated edge pixels with the result of normal median filtering. The mixing factor depends on the difference in luminance values in the vertical and in the edge direction. For a true edge, the difference of the luminance values along the edge must be very small, and the difference in the vertical direction must be high in comparison.

13.2 Coprocessor design

Figure 78 depicts the task graph of the EDDI application. The application is partitioned into five coprocessors:

Input. This coprocessor reads the video data from (external) memory and outputs the data into Eclipse streams.

Filter. This coprocessor performs 3x3 filtering of luminance data to accentuate slanted edges. In addition, it computes the gradient of the luminance signal around each pixel.

Edge detection. This coprocessor detects edges in the luminance data and outputs a stream of edge information.

De-interlacing (DI). This coprocessor de-interlaces the Y and UV data using edge information.

Output. This coprocessor writes Eclipse video streams to the display output.

Figure 78. EDDI task graph.

The input, output, and filter coprocessors are generic, while the edge-detection and de-interlacing coprocessors implement more application-specific functionality. The following subsections detail the design of the data streams in Figure 78.

13.2.1 Input to filter and de-interlacer Y-data streams

Figure 79 depicts three possibilities to format the data stream(s) between the input and filter coprocessors. The 3x3 edge filter accesses three video lines in parallel. The three possibilities impact memory requirements, cache behavior, and coprocessor reusability.

- The three video lines are transmitted as separate streams. The filter has a single point of access to each stream, resulting in optimal cache behavior. The split of video data into three streams decreases the reusability of the input coprocessor. The communication requires three line buffers of Y data, one for each stream.
- The filter uses three different points of access (read pointers) into a single video stream. Access to the Y pixels of three video lines requires a stream buffer of slightly more than two line buffers. Thus, compared to a), the input coprocessor remains application-independent, and memory requirements are minimized. However, the parallel access points of the filter result in sub-optimal cache behavior.
- The filter uses the same setup as in b), but reserves different cache lines for each video line. Each cache line has a single access point, resulting in optimal cache behavior. The filter uses forking to synchronize the triple access to the stream buffer. While improving cache behavior, the shell's cost increases with the larger cache size and additional entries in the stream table.

Figure 79. Data stream options for the stream between input and filter.

Figure 80 extends the analysis with the consumption of Y-data by the de-interlacer. The de-interlacer reads the Y-data of two video lines in parallel. The two ac-

cess points of the de-interlacer follow the same analysis as for the three access points of the filter.

- The input provides separate streams for the filter and de-interlacer. The input coprocessor is extremely application specific. This setup requires a large number of line buffers to copy data to both consumers.
- The three video lines required for the filter are forked to the de-interlacer to reduce stream buffer memory. This setup negatively impacts the reusability of the de-interlacer by forcing the de-interlacer to work with three access points instead of two.
- The input coprocessor produces a single stream of Y-data, forked to both the filter and the de-interlacer. This setup ensures reusability of the input coprocessor, and requires minimal stream buffer memory.
- The filter and de-interlacer improve cache behavior at the cost of increased cache size and additional stream entries by forking the data stream to different cache lines.

Figure 80. Options for the Y-data streams from input to filter and de-interlacer.

The focus on cost and reusability of coprocessors leads us to choose option c). The sub-optimal cache behavior may be alleviated by increasing transport bandwidth. Alternatively, the coprocessors may incorporate local buffers to improve cache behavior by reading multiple pixels at a time from each access point. Section 13.3.3 shows that the coprocessors must operate at the same granularity as the bus and memory (e.g. 32 bits, accessing four pixels at a time) to achieve the required data throughput.

13.2.2 Input to de-interlacer UV-data stream

Figure 81 depicts four options to format the streams of YUV-data from the input coprocessor to the filter and de-interlace coprocessors. The de-interlacer reads YUV data from two video lines in parallel. The Y-data is needed by both the filter and de-interlacer, while the UV-data is only needed by the de-interlacer.

- The input coprocessor has separate outputs for Y- and UV-data streams. Here, the reusability of the input coprocessor is sacrificed.
- The input produces a single YUV stream; the filter skips UV data. In this setup, the filter must skip (via `PutSpace` commands) UV data in the input stream, which reduces the reusability of the filter. Moreover, the filter operating on YUV data leads to large memory requirements of 2+ line buffers of YUV data.
- The input coprocessor is duplicated, one outputs Y-data, while the second outputs UV-data. This leads to minimal memory requirements and does not impact the reusability of the coprocessors. However, the duplication of the input coprocessor increases cost.
- The input coprocessor executes two independent tasks, one for Y, and one task for UV data. This is similar to the setup of option c), but avoids the costly duplication of coprocessor hardware.

Option d) is the preferred solution, as it leads to minimal memory and hardware requirements, yet maintains reusability of the coprocessors.

Figure 81. Options for the Y and UV data streams from the input coprocessor to the filter and de-interlacer.

13.2.3 Stream of filtered pixels and gradients

The filter coprocessor not only filters the pixels, it also computes the gradients of each set of 3x3 pixels. The edge detector uses the gradient information to compare edge points of two successive lines. The gradients are stored in 4 bits per filtered pixel.

13.2.4 Stream of detected edges

The edge detector needs to maintain information on detected edges to correlate edges over successive lines. The local stream must buffer slightly more than one

line of detected edge points. The stream stores the pixel position (12 bit), span (8 bit), and gradients (4 bit) of each detected edge.

The edge detector can store edge information in an Eclipse stream outside the co-processor. Exposing this local data stream to shared memory enables a multi-tasking edge detector that allocates a different edge information stream for each task.

13.2.5 De-interlacer to output stream

Separating the de-interlacer and output coprocessors allows execution of multiple de-interlace tasks, each with their own output buffering. The separation also increases reusability of the de-interlacer, allowing it to connect to other filters (e.g. peaking). However, the large amount of data transmitted between de-interlacer and output coprocessors inflicts a significant bandwidth and memory requirement on the Eclipse infrastructure.

An alternative solution is to combine the de-interlacer and output coprocessors and implement only a small private output buffer. In the memory and bandwidth computations below, we explore the results for a separate output coprocessor as well as for the combined solution.

13.3 EDDI subsystem

Figure 82 gives the EDDI Eclipse instance, assuming the de-interlace and output coprocessors are combined. The Eclipse instance must be able to handle standard-definition (SD) streams at a rate of 30 frames/second input and 60 frames/second output. The input streams are encoded in YUV 4:2:0 or 4:2:2 format. Output format is in display format: YUV 4:4:4. For a competitive solution, the total memory area must cost less than five line memories of YUV data (approx. 10 kByte). Total area should be less than 100 kGates.

Figure 82. Eclipse EDDI instance.

13.3.1 Buffer size

The EDDI application requires less than 7 kByte of buffer memory for the streams indicated in Figure 78:

- s1* This stream buffers slightly more than two lines of luminance data. Each line contains 720 pixels. Eclipse requires buffer sizes aligned to a power of two, resulting in a 2 kByte buffer.
- s2* This stream contains the filtered luminance data as well as the gradients per pixel. The edge detector reads this in a linear fashion, requiring only a small buffer.
- s3* This stream stores edge points for one video line. Each edge point requires three byte of information. If we assume a worst case of one edge every four pixels, this results in a buffer of 1 kByte (720 pixels * 3 byte / 4 pixels, aligned to a power of two).
- s4* This stream contains the edge information for de-interlacing. The de-interlacer consumes this stream in a linear fashion, requiring only a small buffer.
- s5* This stream stores YUV 4:4:4 data, with 3 bytes per pixel. Half a video line should suffice to accommodate data-dependent processing in the edge detector. Alignment on a power of two results in a 2 kByte buffer.
- s6* This stream stores UV information required by the de-interlacer. As the de-interlacer operates on two successive lines, it requires slightly more than one line. Assuming worst-case 4:2:2 format, this results in a 1 kByte buffer.

The required 7 kByte memory fulfills the customer requirement that the total memory should be less than five line memories (approx. 10 kByte). An alternative option is to combine the de-interlacer and output coprocessors. In this case, stream *s5* is kept internal in the combined coprocessor. This results in a total minimal memory requirement of approximately 4.5 kByte.

13.3.2 Memory bandwidth

The in-line pixel rate of an SD stream is 14 MPixel/second. This is the pixel-rate that is required for real-time display of individual lines, excluding the vertical blanking and line blanking periods. For bandwidth calculations, we assume the worst-case situation of 4:2:2 input format, where two luminance (Y) values share one pair of chrominance (UV) values. Thus, a luminance stream requires 14 MByte/second, and a UV stream also requires 14 MByte/second worst case. Total required bandwidth on the Eclipse bus is 350 MByte/second:

- s1* Per pixel, this stream requires 1 write access by the input, 3 reads from the filter, and 2 read accesses from the de-interlacer, resulting in a bandwidth of 84 MByte/second (6*14MByte/second for SD streams).
- s2* Per pixel, this stream requires one write access by the filter, and one read by the edge detector. The stream contains pixel values plus 4-bit gradients per

pixel, giving a bandwidth of 42 MByte/second ($2 \text{ accesses} * 1.5 \text{ byte per pixel} * 14 \text{ MByte/sec}$).

- s3 Per edge point, this stream requires one read and write access. The stream contains 3 byte per edge point with worst case one edge point every four pixels. This results in a bandwidth of 21 MByte/second ($2 \text{ accesses} * 3 \text{ byte} / 4 \text{ pixels} * 14 \text{ MByte/second}$).
- s4 Per edge, this stream requires one read and write access. The stream contains 3 byte information per edge. Worst case, the detector identifies an edge once every four pixels for each set of three lines. The result is a bandwidth requirement of 7 MByte/second ($2 \text{ accesses} * 3 \text{ byte} * 14 \text{ MByte/second} / (4 \text{ pixels} * 3 \text{ lines})$).
- s5 Per pixel, this stream requires one read and write accesses of three byte for YUV 4:4:4 format. The stream has twice the rate of the incoming pixel stream, resulting in a bandwidth of 168 MByte/second.
- s6 Per pixel, this stream requires one read and write access of UV data. This results in a bandwidth of 28 MByte/second ($2 \text{ accesses} * 14 \text{ MByte/second}$).

The EDDI Eclipse instance uses a combined read and write bus to local memory, allowing back-to-back transfers with a new read or write operation every cycle. The parameterized design of the coprocessor shells allows us to explore different options for the memory and bus bandwidth. We decide for either a 32-bits memory and bus operating at 120 MHz, or 64-bits memory and bus operating at 60 MHz. Both scenarios provide a maximal memory bandwidth 480 MByte/second. This is sufficient for the worst-case required bandwidth of 350 MByte/second, resulting in a 73% bus load. In case the de-interlacer and output coprocessors are combined, stream s5 is not visible on the Eclipse bus. In this case, the total required bandwidth reduces to 182 MHz.

13.3.3 Coprocessor performance

We assume a best-case read latency of four cycles on a cache hit, based on the simulation results of Section 10.2.2. On a cache miss, the read latency increases to 12 cycles, assuming minimal bus contention and no prefetching in the cache. Thus, the shell can provide a maximal throughput of 120 MPixels/second ($480 \text{ MByte/second} / 4 \text{ read cycles per pixel}$) to its coprocessor for best-case execution with all accesses hitting in the cache. The worst-case available throughput (all cache misses) becomes 40 MPixels/second.

The compute requirements of the EDDI coprocessors are similar to each other. Hence, the system bottleneck is the coprocessor with the largest I/O requirements. The 3x3 filter has the most severe I/O requirements. The following analysis shows that the 3x3 filter can be designed to meet its real-time requirements.

The filter reads three pixels for every output pixels. We design the coprocessors to operate at the same width as the bus and memory towards their shells. This

yields the highest data throughput available to the coprocessor. In the scenario of a 32-bit memory interface, the 3x3 filter reads four pixels per shell access and thus must internally store maximally two pixels for each of the three video lines.

Reading 32 bit words from three video lines with a rate of 14 MPixels/second, the filter requests a throughput of 10.5 MPixels/second (3 read accesses * 14 MPixels/second / 4 pixels per word) from its shell. The worst-case available bandwidth of 40 MPixels/second at the coprocessor-shell interface easily fulfills this requirement.

The filter coprocessor itself requires worst-case 44 cycles per produced output word of four pixels: 3 reads * 10 read cycles worst-case + 2 cycles multiply-accumulate + 2*3 cycles write + 4 cycles synchronization + 2 cycles for the `GetTask` operation. Here, we assume the 4-bit gradient information per pixel is combined into one byte for every two pixels.

Assuming the coprocessor runs at the same speed as the memory bus, this results in an output rate of 11 MPixels/second (120 MHz * 4 pixels per word / 44 cycles per word). This is insufficient to meet the video requirement of a 14 MPixel/second in-line pixel rate. However, the communication buffers may relax this requirement at the filter. The line buffers at the input and output of the system are sufficient to average out the horizontal line blanking, and thereby reduce the required pixel rate. The vertical blanking period can only be used to lower the pixel rate requirement by adding significantly more buffering to the system.

Clearly, the coprocessor execution is dominated by the read latency. A pipelined implementation yields 30 cycles per output word of four pixels: 3 reads * 10 read cycles. This yields an output rate of 16 MPixels/second, which is sufficient for real-time execution at the display rate.

A separate output coprocessor cannot meet the required data throughput. For every four output pixels of three byte each, a pipelined output coprocessor requires 30 read cycles. The resulting data throughput of 16 MPixels/second is insufficient for real-time display of the doubled video lines. For the EDDI Eclipse instance, the customer preferred to combine the de-interlacer and output coprocessor. The combination minimizes the required bus bandwidth, memory size, and coprocessor performance at the cost of reduced flexibility.

13.3.4 Silicon area

Table 16 gives the estimated area of the EDDI subsystem, showing three options for the coprocessor shells. In all three options the de-interlacer and output coprocessor are combined. Option 1 and 2 define shells with individual cache lines for all access points of the filter and de-interlace coprocessors, according to the forking scenario of Figure 80d. In option 1, each cache line contains two 32-bits words; in option 2 each cache line has only one word. In both cases, the perfect streaming behavior at each access point will result in optimal prefetching. Option

3 shows the preferred scenario in which the filter and de-interlacer use only a single cache line for all accesses into luminance data. In this case, the cache prefetching mechanism will increase the number of misses and must be disabled. Without prefetching and the coprocessor operating at the same granularity as the bus and memory, the cache has no added value. Hence, the shells of the filter and de-interlacer do not have a read cache in option 3.

The analysis shows that all options easily meet the requirement that the design must be smaller than 100 kGates (excluding the buffer memory). Though consuming a relatively large portion of the total silicon area, the shells provide the flexibility and scalability to accommodate multiple streams or to add additional coprocessors for e.g. peaking and scaling.

Table 16. Estimated silicon area in CMOS90LP for the EDDI subsystem.

Parameter	1) Full flexible	2) Single word cache	3) No forking
Shells storage size	4829 bits	4101 bits	3072 bits
	29 kGates	25 kGates	18 kGates
Shells control size (gates)	29 kGates	25 kGates	18 kGates
Shells total size (gates)	58 kGates	49 kGates	37 kGates
	0.21 mm ²	0.18 mm ²	0.13 mm ²
2K*32 memory size	0.066 mm ²	0.066 mm ²	0.066 mm ²
Coprocessors size	20 kGates	20 kGates	20 kGates
	0.07 mm ²	0.07 mm ²	0.07 mm ²
EDDI module total size	0.35 mm²	0.32 mm²	0.27 mm²

The coprocessor area of 20 kGates is the total area of a previous dedicated EDDI implementation. The area of the coprocessor shells is estimated in a detailed spreadsheet in which all design parameters—such as the number of cache lines, stream entries, etc.—are tuned for the EDDI instance. We count the number of storage bits in the design and multiply this with the area required for a scan flip-flop. As a general rule of thumb, designs may require an equal amount of area for logic as for storage. We assume this to be true for the coprocessor shells. Furthermore, we assume a layout efficiency of 80 percent to assess the total shell area. Table 17 gives the technology parameters used. Previous synthesis results of the initial shell design have shown that these assumptions result in accurate area estimates.

Table 17. Used technology parameters for CMOS90.

Scan flip-flop size	18 μm ²
Standard gate size	3 μm ²
Layout efficiency	80%

The area required for the local memory is computed from a detailed estimator available for each memory type. Table 18 shows the memory sizes for the origi-

nal, non-Eclipse based, design of EDDI. The original design deployed four individual memories of 1 kByte, closely linked to the coprocessor. With Eclipse, all communication buffers are allocated in a single large memory. This matches with a 4.5 kByte memory in Eclipse. Table 18 shows that such a single centralized memory yields a small area reduction when compared to the original solution that employed multiple smaller memories. For CMOS18 and CMOS12 technologies (used in the original business case), the relative area improvement of a single centralized memory is even larger.

Table 18. Area of multiple small SRAMS versus a single large SRAM.

Number	Size	Area (CMOS90LP)
4	1 kByte	0.084 mm ² 11.2 k equivalent gates
1	4.5 kByte	0.066 mm ² 8.8 k equivalent gates

13.4 Conclusion

The EDDI subsystem discussed above shows that the Eclipse concept also enables cost-effective subsystems in the low-end market, in addition to the high-end video-coding subsystem presented in the previous chapter. The memory efficiency of Eclipse's centralized shared SRAM reduces the total area compared with traditional subsystems that instantiate multiple, smaller buffer memories with relatively large area.

On the other hand, the shared memory may complicate real-time analysis. While Eclipse is designed for soft real-time memory-to-memory operation, the EDDI analysis shows that Eclipse can also be deployed for hard real-time requirements. The caches in the coprocessor shells do not contribute to the worst-case performance but still consume a significant part of the shell area. However, removing them from the shell is possible thanks to the shell's parameterized design. Alternatively, a detailed analysis of the coprocessor and cache behavior (including prefetching) may further reduce the worst-case requirement.

The formatting of the data streams greatly influences the area, flexibility, and reusability of the subsystem. Eclipse shells provide a large flexibility to realize optimal coprocessor design. The EDDI subsystem demonstrates that the shell services allow the designers to exploit different trade-offs between performance, memory area, coprocessor complexity, and flexibility. The EDDI subsystem exploits shell services in new ways that were not anticipated during the design of the Eclipse shell. Examples are the use of forking to enable multiple (cached) access points to a single data stream, and multi-tasking to increase coprocessor reusability by distributing the production of related data streams over different tasks.

The complexity of the EDDI coprocessors is very low, with a total area roughly a factor 15 smaller than the MPEG coprocessors of Chapter 12. As such, the EDDI subsystem is a good benchmark for a low-cost subsystem. Due to the simple coprocessors, the total shell area of 0.13 mm^2 is relatively large compared to 0.07 mm^2 for a dedicated Philips implementation of the complete subsystem (excluding memory). Despite the relatively large area, the customer considered the Eclipse subsystem as a technically superior solution. The additional flexibility offered by the shells together with the reduced memory compared to the individual line buffers of the original solution was seen as an important benefit.

Despite the positive technical response, the customer decided to use an existing implementation of the EDDI algorithm, albeit with less features. The main reason for this choice was the high risk involved with adopting innovative technology as well as the perceived risk that the department responsible for Eclipse technology, in its new role as provider of subsystem infrastructure IP, would not be able to deliver on commitments.

14 Audio accelerator subsystem

Audio functions are sufficiently small in terms of memory footprint and performance requirements such that a software-only implementation is viable. Individual audio functions easily fit on today's programmable processors and do not require a further split-up in smaller sub-functions. In our discussion below, we divide the audio domain into two distinct types of processing: audio compression and audio post-processing.

Audio devices must support a range of audio compression standards (e.g. AAC+, Mp3, WMA). Every standard describes both encoding and decoding functionality. These functions largely consist of control code (e.g. if-then-else branches) and require a large memory (code and data) footprint. Audio compression requires flexible hardware to cope with the rapidly evolving audio compression standards. The above properties of audio compression suggest a mapping to a programmable RISC processor possibly with instruction and data caches.

Audio post-processing consists of more traditional DSP functions, such as sample-rate conversion, and equalization. These functions largely consist of regular signal processing code and operate on small data granularity. While the algorithms and ordering of functions are carefully tuned by each vendor, the compute kernels such as sample-rate conversion and FIR filtering hardly change. Other than for example digital television SoCs in which the video functions dominate the overall chip cost, audio post-processing functions in car radio and portable audio products have a significant impact on the overall (program) memory cost and power consumption.

This chapter describes a recent study to implement dedicated audio post-processing kernels to offload the programmable DSPs and reduce memory and power consumption for a car radio SoC.

14.1 Processing tiles

Figure 83 gives an overview of the audio subsystem in the car radio SoC [10][11]. Audio and radio functions are mapped to *processor tiles*, according to the Sea-of-DSP (SoD) concept. A tile consists of a programmable processor (DSP or CPU) with its program and data memory. Each tile executes in its own voltage and frequency domain. Tiles communicate with each other through asynchronous point-to-point links, termed ITC (inter-tile communication). Figure 83 introduces a new tile type, coined *accelerator tile*, based on the Eclipse template. The depicted accelerator tile for audio and analog radio processing clusters functions such

as a sample rate converter, FIR/IIR filter, and potentially a CORDIC [30]. The clustering of small accelerators in an accelerator tile not only allows efficient communication among the accelerators, but also allows to amortize the hardware cost of inter-tile communication over a larger design.

Figure 83. Audio SoC subsystem consisting of multiple processing tiles.

14.1.1 DSP tile

Figure 84 shows a simplified view of inter-tile communication among three DSP tiles. Processor tiles communicate with each other through asynchronous ITC links [132]. This setup has been deployed in an NXP car radio SoC subsystem that consists of four interconnected DSP tiles, executing analog radio, audio compression, and audio post-processing functions [10][11].

Figure 84. Inter-tile communication (ITC) links between three DSP tiles.

The SoD streaming setup is based on the Eclipse concept, with a similar software API on the host processor to instantiate and interconnect tasks on the DSP tiles. Hereto, the local memory of each tile is accessible by the host processor via a DMA interface. Task scheduling and synchronization of data transport is fully

implemented in DSP software. The **Read**, **Write**, **GetSpace** and **PutSpace** commands are implemented in software and operate on a word granularity, matching the word size of the DSP. To transport data from one tile to another, a producer task on tile 0 directly writes into the local memory of the corresponding consumer task on tile 1. The ITC block in a tile discerns between writes to the local memory or to a specific ITC link based on the write address. Per link, the ITC block defines a configurable mapping from a local address range to an address range of the same size in the remote tile.

As in the Eclipse shell, both the producer and consumer locally maintain the administration of all communication buffers that pertain to the tasks mapped on the tile. The buffer administration is synchronized by sending synchronization messages in the form of ITC writes to update the buffer administration in the remote tile's local memory. Equivalent to the buffer administration in the Eclipse shells, the latency of sending synchronization messages does not affect functional correctness. The consumer task may have a pessimistic view on the buffer filling when synchronization messages are still in-flight.

The SoD **PutSpace** command to send synchronization messages, as well as the SoD **GetSpace** command to check the local buffer administration are implemented in software. The buffer administration consists of a buffer base address and buffer size needed for cyclic addressing, and a read and write pointer into the buffer. A **PutSpace** action on the local tile only updates one of these pointers, while a **PutSpace** on the remote tile updates the other. By comparing pointers, the local tile has information on the buffer filling needed for the **GetSpace** operation. The separate read and write pointers avoid the need for atomic access by either tile.

Figure 85. I/O structure of the accelerator tile. The audio coprocessors within the tile are not shown. The I/O coprocessor communicates with DSPs through the ITC interface. It may also communicate to external devices via I²S.

14.1.2 Accelerator tile

Figure 85 gives the I/O structure in the accelerator tile. The addition of an I/O coprocessor allows to separate the address space of local memory in the accelerator tile from the address space supported by the connected tiles. The I/O coprocessor handles all off-tile channels, including the inter-tile communication and I²S com-

munication. The tasks on the I/O coprocessor handle precisely one logical communication channel.

Figure 86 depicts the architecture of the I/O coprocessor. The I/O coprocessor uses different ITC address ranges for remote read, write, and commit operations. Per channel, the I/O coprocessor maintains a read or write pointer, buffer size, and buffer base address of the remote channel buffer, as well as information on the selected ITC link and address of the remote buffer administration. The table also stores the packet size at which the remote tile operates.

Figure 86. Architecture of the I/O coprocessor.

The I/O coprocessor translates incoming ITC writes and synchronization commands to shell **Write** commands. The SoD **Write** call is augmented to write the data to a single address per channel in the coprocessor tile, instead of writing data directly in the buffer. Regardless of the size of the buffer in the coprocessor tile, each communication channel requires only a single address in the address space of the connected tile. Likewise, the SoD implementation of **PutSpace** is augmented to write the amount of committed data to the coprocessor tile instead of directly writing the buffers' updated read or write pointer.

The coprocessor issues **GetSpace** commands to its shell to check for availability of a next empty or full packet in the Eclipse buffer upon reception of an SoD **PutSpace** command. The data packet granularity is programmed in the coprocessor's channel administration table. If a **GetSpace** action fails, the coprocessor keeps checking for new space at every task invocation until the **GetSpace** request succeeds.

The I/O coprocessor actively copies newly produced data in the Eclipse buffers to the outgoing ITC links. Hereto, the I/O coprocessor task implements a data pump that reads data from the shell and copies this to the remote buffer. The I/O coprocessor task only commits copied data when a full packet has been copied. Upon a commit, the task updates both the local administration (via a **PutSpace** action to the shell) and the remote administration (via an ITC write of the write pointer in the SoD buffer administration).

Note that a task to copy data from Eclipse to SoD may be active in parallel with a task that handles incoming SoD commands. Eclipse's shell interface specifies the task ID on each interface primitive, originally designed for pipelined coprocessors. The task ID argument allows the I/O coprocessor to issue shell commands for a different task than the active task dispatched by shell's scheduler.

14.2 Coprocessor design

The SRC, FIR/IIR, and CORDIC coprocessors each handle up to four tasks, required for two stereo radio/audio streams. The I/O coprocessor is configured for eight input and eight output streams, thus handling eight tasks.

The CORDIC coprocessor has one input and one output stream per task. The CORDIC operation is programmed into the task info field via control software. The input stream contains the operands, the output stream holds the results. The sample-rate converter (SRC) has two input streams and two output streams per task. The SRC uses one private stream to store the upconverted input signal.

The IIR/FIR coprocessor has three input streams and one output stream per task. The first input stream contains control commands, such as the number of samples to be filtered with a particular coefficient set. The second input stream contains the coefficient data. These are kept alive in the stream over task executions. The task only shifts to new coefficients when a next set of coefficients is available. The third stream of the FIR/IIR coprocessor contains the signal data. The task info field in the shell contains information on FIR/IIR, the number of filter stages (maximum 256), real/complex signal data, symmetric/asymmetric, and the coefficient set (maximum 32).

The required memory bandwidth is determined by the maximum number of streams that can operate in parallel. For the high-bandwidth radio streams, we assume a block granularity of eight samples of two bytes per sample, transmitted at 48 kHz. The input and output streams FIR/IIR carrying signal data are high-bandwidth streams, as well as the input and output streams of the CORDIC. The SRC reads from and writes to its local stream of upconverted samples, accounting for two high-bandwidth streams. In a worst-case scenario, all 16 streams handled by the I/O coprocessor are high-bandwidth streams. Taking into account the maximum number of tasks on each coprocessor, the FIR/IIR, SRC and CORDIC each access eight streams and the I/O coprocessor handles 16 streams, generating a total of 40 high-bandwidth streams. The resulting cumulative worst-case memory bandwidth requirement for fast streams is approximately 31 MByte/sec ($40 \text{ streams} * 2 \text{ byte/sample} * 8 \text{ sample/block} * 48\text{K blocks/sec}$).

For the remaining streams, we assume one sample per block. The I/O coprocessor handles four such streams for I²S. The SRC accesses a slow input and output stream per task, a total of eight streams if all four SRC tasks are enabled. This re-

sults in an estimated cumulative memory bandwidth of 1 MByte/sec ($12 \text{ streams} * 2 \text{ byte/sample} * 48\text{K samples/sec}$) for the low-bandwidth streams.

A 16 bits local bus and memory operating at 125 MHz can easily accommodate the estimated worst-case bandwidth of 32 MByte/sec. A coprocessor clock frequency of 250 MHz easily allows operation of four tasks in real-time for SRC, FIR, and Cordic.

14.3 Conclusion

The SoD concept of interconnected tiles builds on the Eclipse streaming concepts and extends these to the realm of programmable processors. While the video coding subsystem of Chapter 12 focused on tight integration of the media processor with the embedded memory, SoD prepares for future process technology which forces a separation into isolated voltage and frequency islands—each with their own processor and local memory.

The asynchronous connection between SoD tiles allows to operate each tile at its own clock frequency. This caters for a dedicated accelerator tile with multi-tasking coprocessors that operate at a relatively high clock frequency to handle all tasks in real-time, while keeping the clock frequency of connected DSP tiles low to minimize power consumption. The I/O coprocessor within the accelerator tile further separates the address space of the local memory in the accelerator tile from the address space of the connected tiles.

The asynchronous links between tiles—as well as the use of shared local memory inside the accelerator tile—introduce long latencies in communication. For audio post-processing, latency is generally not a concern since the applications have no low-latency feedback loops. Radio processing does have low-latency feedback loops (e.g. for scanning antenna selection) that involve CORDIC and FIR operations. Further research is required to study alternative algorithms that can cope with longer latency and benefit from accelerators in a separate clock domain. This is outside the scope of this thesis.

The accelerator subsystem shows the feasibility of deploying Eclipse for audio (post-)processing applications. The bandwidth estimations validate the concept of cooperation between an Eclipse subsystem and a programmable DSP for audio applications. At the time of publication of this thesis, the audio subsystem has not been implemented in a product. The main reason is that the customer's focus shifted towards implementation of a digital radio platform, in which the silicon area for audio post-processing is negligible in comparison to the requirements for digital radio and audio compression.

15 Summary

Consumer multimedia systems, such as digital television, DVD recorders, smart phones, and car audio systems are produced in high volumes and sold for a relatively low price in comparison with the inherent complexity of a multimedia system. Chip designs that target this market are primarily cost-driven, needing relatively large development effort to minimize product cost. Traditionally, semiconductor vendors develop a custom-tailored chip for each new product to optimize cost and performance of the target system. However, the rapid succession of new products in the market as well as the pace at which new applications are introduced and multimedia standards evolve forfeits this approach. The answer lies in the design of flexible platforms that target a whole range of products in a single architectural effort. Particular instances of such a more generic platform can then quickly be created and tuned to the product's specific requirements. Such flexibility goes together with programmability, in which a software program largely defines the function of the chip. However, the performance per square millimeter of silicon of such programmable solutions is roughly two orders of magnitude lower than for hardwired solutions. For the high-volume consumer market, software-only solutions are therefore typically not cost-effective.

The Eclipse architecture described in this thesis addresses the dilemma between required flexibility and low product cost by combining the cost-effectiveness of hardwired modules as its basic elements, called *coprocessors*, with the flexibility of programming applications in software. The coprocessors implement the common compute kernels that can be shared by a set of different but similar applications. Each particular application is then programmed on Eclipse by mapping the application's tasks to coprocessors and interconnecting tasks via buffered data streams, following the computational model of a Kahn Process Network.

To allow system designers to quickly create Eclipse coprocessors for a specific product, Eclipse separates each coprocessor from its surroundings via a clearly defined interface. The generic *coprocessor shell* connected to each coprocessor implements the response to the coprocessor's requests. Through the use of six interface primitives, the coprocessor accesses services for data access, data caching, synchronization of data access between coprocessor tasks, scheduling tasks to execute on the coprocessor, and reporting events triggered by the coprocessor to a control processor. These services offered by the shell both simplify the coprocessor design and hide the specific implementation of the memory interconnect and implementation details of other coprocessors from the coprocessor. This abstraction enables a designer to easily deploy a given coprocessor in another Eclipse in-

stance. A coprocessor itself can be designed with some flexibility to perform a number of related functions to facilitate such reuse.

Part I of this thesis details the design and use of the six interface primitives and presents four unique features of the coprocessor shell:

- distributed synchronization, in which each shell has local knowledge of the availability of data and room in incoming and outgoing data streams;
- distributed scheduling, where each shell locally decides what task the coprocessor should execute next, based on the availability of data and room in the data streams and the results of a previous execution of the same task;
- cache coherency and automatic data prefetching derived from explicit synchronization actions by the coprocessors;
- budgeting of task execution time to meet (soft) real-time deadlines.

Despite its approach of distributed synchronization and scheduling, Eclipse supports application reconfiguration—changing the parameters or topology of the application at run-time—at a single location in the application’s data processing, e.g. at the end of an MPEG-2 video frame. So-called location packets in the data streams travel with the data and are recognized and forwarded by each coprocessor. The coprocessor reports to its shell whenever it identifies a location packet in an input stream, upon which the shell stops further processing of the task and triggers a control processor to handle the reconfiguration. Using this mechanism, individual applications can be reconfigured without interrupting the processing of other applications mapped onto the Eclipse subsystem. Eclipse offers an application programming interface (API) for creating and reconfiguring applications. Through this API, control software modifies fields in the shell to specify the in- and outgoing data streams and the tasks for the coprocessor to execute.

Part II presents a SystemC simulation model of the Eclipse infrastructure (shells, bus, memory, control interface), developed to facilitate early development of coprocessors and the accompanying control software. To suit the highly parameterized design of Eclipse, we introduced new constructs for distributed parsing and setting of parameters in the initialization phase of the simulation model. The resulting model equals Eclipse’s flexibility, allowing it to be instantiated for different types and numbers of coprocessors, with different bus and memory widths and clock speeds, coprocessor shells with or without caches, etc. In addition, the model can operate in two modes: a non-timed mode for functional verification, and a slower timed mode for cycle-based verification of coprocessor hardware. The constructs built on top of SystemC to support parameterized designs and to support both functional verification and cycle-based simulation modes are generally applicable.

The Eclipse template centers around the infrastructure that offers generic services to coprocessors through the shells and to system integrators through the control interface. This infrastructure has been validated through simulation experiments with a multi-stream MPEG-2 decoder. The experiments show the effectiveness of

the relatively small data caches in the shell, and the correct operation of budget-based scheduling and dynamic application reconfiguration.

Eclipse targets soft-real-time applications, in particular those requiring memory-to-memory operation with relatively large buffers at the input and output of the application with respect to the buffering of the data streams between internal application tasks. This property is reflected in the design of Eclipse, which deploys a centralized shared memory for the flexible allocation of the internal stream buffers. The simulations show that the data caches and the deeply pipelined shell design clearly improve average case performance. The coprocessors must be designed to handle a structural overload due to the often unpredictable, data-dependent behavior of multimedia applications through a (graceful) degradation of output quality. MPEG video decoders typically implement such a mechanism by skipping video frames.

Part III details case studies in which Eclipse subsystems were designed to achieve video compression, image processing, and audio processing. The breadth of the applications covered by these cases demonstrates the wide applicability of the Eclipse template. The case studies are evaluated using straightforward estimations of silicon area, bandwidth requirements, and power consumption. This procedure reflects the industrial practice, where architects must decide between candidate architectures long before detailed requirements, let alone executable specifications of the applications, become available.

In its description of how to apply the six interface primitives, Part I proposed a design trajectory for deriving coprocessors from executable specifications of each application that the subsystem must support. The video compression subsystem follows this trajectory to define coprocessors that can be applied for both MPEG-2 video encoding and decoding functions. The image processing subsystem demonstrates the flexibility offered by the shell to the coprocessors through the six interface primitives by detailing different options for creating maximally independent coprocessors.

The image processing subsystem presents a former Philips Semiconductors business case. Here, Eclipse is benchmarked against an older, dedicated implementation realizing the same functionality. Compared to the video compression subsystem, the coprocessors for the image processing are very simple and roughly a factor 15 smaller in silicon area. As a result, the coprocessors and the shells are almost equal in silicon area (even with all four shells consuming a mere 0.13 mm² in 90 nm technology). Despite this area overhead of the shells, the flexibility offered by the Eclipse solution for handling multiple streams and the flexibility to quickly extend the feature set by adding new coprocessors caused the customer to value the Eclipse approach as technically superior.

In the last chapter of Part III, the audio subsystem serves to show how the processing in an Eclipse setup can be intermixed with audio processing in software, executed on multiple DSP processors. Eclipse in fact paved the way for the pre-

sented multiprocessor architecture with distributed synchronization and scheduling in software as realized in a later Philips/NXP car-radio product.

A In-place update synchronization

This appendix gives the synchronization scheme for in-place updates, as discussed in Section 4.1.3. The example serves to show the flexibility of the synchronization scheme implemented by the Eclipse shells. Assume three tasks *A*, *B*, and *C*, where *A* is the producer, *C* is the consumer, and task *B* performs in-place updates on the data produced by *A* (Figure 12). All three tasks operate on the same stream buffer.

Figure 87 gives the synchronization scheme as programmed into the shells of each task. The figure depicts `put_space` messages communicated between the tasks. For clarity, we distinct here between signaling available room for writing (`put_room`) and available data for reading (`put_data`). Task *B* sends synchronization messages to itself to keep the administration of the buffer content up-to-date for both input and output ports. Note that these synchronization messages make a complete round on the synchronization token ring, as the shell does not provide a shortcut mechanism for synchronization between stream entries in the same shell. Clearly, this delay in synchronization messaging does not affect functional correctness.

Figure 87. *synchronization message for in-place updates.*

All three tasks operate on the same buffer by programming the same buffer base address in the shell for all ports. Since all tasks play the full `GetSpace/PutSpace` protocol on their input and output ports, the shell can maintain the access points and guarantee cache coherency, even with the separate read and write cache implementation. However, `GetSpace` commands on both input and output ports of task *B* without issuing a `PutSpace` on the input port in between both `GetSpace` commands will lead to immediate deadlock. Moreover, the shell does not guarantee cache coherency when task *B* re-reads overwritten data without separating the read and write stages with `GetSpace/PutSpace` synchronization actions.

Bibliography

- [1] Alpey, J. et al., "Star-IP Centric Platforms for SoC, ARM PrimeXsys Platform Architecture and Methodologies", *Winning the SoC Revolution*, G. Martin & H. Chang (Ed.), Kluwer Academic Publ., pp. 187-227, 2003.
- [2] ARM, *PrimeXsys™ Platforms, Extendible Platform Architecture*, <http://www.arm.com/products/solutions/PrimeXsysPlatforms.html>.
- [3] ARM, *AMBA Home Page, the de facto Standard for On-Chip Bus*, <http://www.arm.com/products/solutions/AMBAHomePage.html>, 2006
- [4] Arteris, *A Comparison of Network-on-Chip and Buses*, white paper, http://www.edgs.co.il/userfiles/file/noc_whitepaper.pdf, 2005.
- [5] Balarin, F. et al., *Hardware-Software Codesign of Embedded Systems—The POLIS Approach*, Kluwer Academic Publ., 1997, Dordrecht, The Netherlands.
- [6] Balarin, F. et al., "Metropolis: An Integrated Electronic System Design Environment", *IEEE Computer*, vol.36, no. 4, pp. 45-52, April 2003.
- [7] Van Balen, M. & M.J. Rutten, *C to YAPI, Guidelines and examples*, Nat.Lab. Technical Note NL-TN 2001/014, Philips Research internal report, Feb. 2001, Eindhoven, The Netherlands.
- [8] Basten, T. & J. Hoogerbrugge, "Efficient Execution of Process Networks", *Communicating Process Architectures*, IOS Press, 2001.
- [9] Berekovic, M.; G. Meyer, Y. Guo & P. Pirsch, "A Multimedia RISC Core for Efficient Bitstream Parsing and VLD", *Proc. of the SPIE: Multimedia Hardware Architectures*, vol. 3311, pp. 131-141, Jan. 1998.
- [10] Van den Berg, R. & H.S. Bhullar, "Next generation philips digital car radios, based on a sea-of-dsp concept", *IEEE ISPC GSPx*, 2004.
- [11] Bhullar, H.S.; R. van den Berg, J. Josten & F. Zegers, "Serving digital radio and audio processing requirements with sea-of-dsps for automotive applications the philips way", *IEEE ISPC GSPx*, 2004.
- [12] Bilas, A.; J. Fritts & J. Pal Singh, "Real-Time Parallel MPEG-2 Decoding in Software", *11th Int. Parallel Processing Symp. (IPPS)*, pp. 197-203, April 1997, Geneva, Switzerland.
- [13] Bove Jr., V.M. & J.A. Watlington, "Cheops: A reconfigurable Data-flow System for Video Processing", *IEEE trans. On Circuits and Systems for Video Technology*, vol. 5, no. 2, pp. 140-149, April 1995.
- [14] Bril, R.J. et al., "Multimedia QoS in Consumer Terminals", *IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 332-344, Sept. 2001, Antwerp, Belgium.
- [15] Bruls, W.H.A.; E.W. Salomons, A. van der Werf, R.K. Gunnewiek & L. Camiciotti, "A low-cost audio/video single-chip MPEG2 encoder for consumer video storage applications", *IEEE Int. Conf. on Consumer Electronics (ICCE)*, pp. 314-315, June 2000, Los Angeles, CA, USA.
- [16] Brunel, J.Y. et al., "COSY: a methodology for system design based on reusable hardware & software IP's", J.Y. Roger (Ed.), *Technologies for the Information Society*, IOS Press, pp. 709-716, 1998.
- [17] Brunel, J.Y. et al., "COSY Communication IP's", *Proc. 37th Design Automation Conf.*, pp. 406-409, June 5-9, 2000, Los Angeles, CA, USA.
- [18] Brunel, J.Y.; E.A. de Kock, W.M. Kruijtzter, H.J.H.N. Kenter & W.J.M. Smits, "Communication Refinement in Video Systems On Chip", *Proc. 7th Int. Workshop on Hardware/Software Codesign (CODES)*, pp. 142-146, May 3-5, 1999, Rome, Italy.

- [19] Buck, J.T.; S. Ha, E.A. Lee & D.G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", *Int. Journal of Computer Simulation: Simulation Software Development*, vol. 4, pp. 155-182, April 1994.
- [20] *Cadence NC-SC Simulator*, <http://www.cadence.com>.
- [21] *Cadence Virtual Component Co-Design (VCC) Environment*, <http://www.cadence.com>.
- [22] Carriero, N. & D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed", *ACM Computing Surveys*, vol. 21, no. 3, pp. 323-357, Sept. 1989.
- [23] Chen, T-F. & Jean-Loup Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors", *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609-623, May 1995.
- [24] Coffland, J.E. & A.D. Pimentel, "A Software Framework for Efficient System-level Performance Evaluation of Embedded Systems", *Proc. 18th ACM Symp. on Applied Computing*, March 2003, Melbourne, FL, USA.
- [25] Cohn, J.M., "Technology Challenges for SoC Design, and IBM Perspective", *Winning the SoC Revolution*, G. Martin & H. Chang (Ed.), Kluwer Academic Publ., pp. 255-296, 2003.
- [26] Corporaal, H., *Microprocessor Architectures; from VLIW to TTA*. John Wiley, 1998.
- [27] Culler, D.E.; J.P. Singh & A. Gupta, *Parallel Computer Architecture, A Hardware/Software Approach*, Morgan Kaufmann Publ., San Francisco, CA, 1990.
- [28] Cumming, P., "The TI OMAP Platform Approach to SoC", *Winning the SoC Revolution*, G. Martin & H. Chang (Ed.), Kluwer Academic Publ., pp. 97-118, 2003.
- [29] *Cynthesizer Closes the ESL-to-Silicon Gap*, Forte Design Systems, <http://www.fortedes.com/products/cynthesizer.asp>, 2007.
- [30] Dawid, H. & H. Meyr, *CORDIC Algorithms and Architectures* http://www.eecs.berkeley.edu/~newton/Classes/EE290sp99/lectures/ee290aSp996_1/cordic_chap24.pdf.
- [31] Dömer, R., *System-level Modeling and Design with the SpecC Language*, PhD thesis, Dept. of Computer Science, University of Dortmund, 2000, Dortmund, Germany.
- [32] Dutta, S.; R. Jensen & A. Rieckmann, "Viper: A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems", *IEEE Design and Test of Computers*, pp. 21-31, Sept-Oct. 2001.
- [33] Dwivedi, B.K.; J. Hoogerbrugge, P. Stravers & M. Balakrishnan, "Exploring Design Space of Parallel Realizations: MPEG-2 Decoder Case Study", *Proc. 9th Int. Symp. on Hardware/Software Codesign (CODES)*, pp. 92-97, April 25-27, 2001, Copenhagen, Denmark.
- [34] Van Eijndhoven, J.T.J. et al., *ECLIPSE CPU-Coprocessor Architectures*, Nat.Lab. Technical Note NL-TN 405/99, Philips Research internal report, Feb. 2000, Eindhoven, The Netherlands.
- [35] Van Eijndhoven, J.T.J., *ECLIPSE: Shell to processor services, Data-caching, dataflow synchronization, multi-tasking*, Philips Research Technical Note PR-TN 2003/00613, Philips Research internal report, Aug. 2003, Eindhoven, The Netherlands.
- [36] Van Eijndhoven J.T.J. et al., "TriMedia CPU64 Architecture", *Int. Conf. on Computer Design (ICCD)*, pp. 586-592, Oct. 10-13 1999, Austin, Texas, USA.
- [37] Van Eijndhoven, J.T.J.; E.J.D. Pol & M.J. Rutten, *Data processing system for high-definition digital television, has selector to select locations for storing elements of data stream in cache memory, according to stream identification*, Granted patents WO-2004079488 A2, EP-1604286 A2, Feb. 19, 2003.
- [38] Van Eijndhoven, J.T.J.; E.J.D. Pol & M.J. Rutten, *Data processing system for MPEG decoding has processors reading and writing for random access of memory section locations defined by administration unit*, Granted patents WO-2003052587 A2, AU-2002351136 A1, EP-1459178 A2, JP-2005513611 W, CN-1605065 A, Dec. 14, 2001.
- [39] Van Eijndhoven, J.T.J.; E.J.D. Pol & M.J. Rutten, *Processing MPEG data by using inquiry and commit operations to control memory buffer FIFO behavior for data object stream transport between processors*, Granted patents WO-2003052589 A2, AU-2002366408 A1, EP-1459181 A2, US-20050015372 A1, CN-1602469 A, JP-2005528671 W, Dec. 14, 2001.

- [40] Van Eijndhoven, J.T.J.; E.J.D. Pol, M.J. Rutten & O.P. Gangwal, *Data processing system for MPEG decoding issues commit or inquiry commands to release assigned space*, Granted patents WO-2003052588 A2, AU-2002366404 A1, EP-1459180 A2, US-20050015637 A1, JP-2005521124 W, CN-1605065 A, Dec. 14, 2001.
- [41] Van Eijndhoven, J.T.J.; E.J.D. Pol, M.J. Rutten, P. van der Wolf & O.P. Gangwal, *Data processing system for MPEG decoding has adaptable first task-level interface and communication means hiding implementation aspects of network to processor*, Granted patents WO-2003052586 A2, AU-2002343180 A1, EP-1459177 A2, KR-2004065258 A, US-20050021807 A1, JP-2005513610 W, CN-1605064 A, Dec. 14, 2001.
- [42] Van Eijndhoven, J.T.J.; J. Hoogerbrugge, Jayram M.N, P. Stravers, & A. Terechko, "Cache-coherent heterogeneous multiprocessing as basis for streaming applications", *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, P. van der Stok (Ed.), Springer, pp. 61-80, 2005, The Netherlands.
- [43] Van Eijndhoven, J.T.J.; M.J. Rutten & E.J.D. Pol, *Data processing system for high-definition digital television, performs prefetch operation to replace data at location in cache memory identified to be dismissible without penalty, with another data that fits in that location*, Granted patents WO-2004079489 A2, EP-1604285 A2, Feb 19, 2003.
- [44] Essink, G.; G. Hekstra & P. van der Wolf, *Multi-Processor Performance Data Collection and Visualization*, Nat.Lab. Technical Note NL-TN 2001/089, Philips Research internal report, Dec. 2001, Eindhoven, The Netherlands.
- [45] Fairbank, D. et al., "The VSI Alliance: journey from vision to production", *Electronic Design*, vol. 46, no. 1, pp. 86-92, Jan. 12, 1998, <http://www.vsi.org>.
- [46] *Flexible Platform-Based Design with the CoWare N2C Design System*, CoWare Inc., White paper, Oct. 2000, <http://www.coware.com>.
- [47] Florescu, O.; M. de Hoon, J.P.M Voeten & H. Corporaal, "Probabilistic Modeling and Evaluation of Soft Real-Time Embedded Systems", *Proc. Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS '06)*, pp. 206-215, 2006.
- [48] Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
- [49] Fowler M. & K. Scott, *UML Distilled, A Brief Guide to the Standard Object Modeling Language*, 2nd Edition, Addison-Wesley, 2000.
- [50] Fu, J.; J. Patel & B. Janssens, "Stride directed prefetching in scalar processors", *Proc. 25th Int. Symp. on Microarchitecture*, pp. 102-110, Dec. 1992, Portland, Oregon, USA.
- [51] Gamma, E.; R. Helm, R. Johnson & J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, Oct. 1997.
- [52] Gangwal, O.P.; A. Nieuwland & P. Lippens, "A Scalable and Flexible Data Synchronization Scheme for Embedded HW-SW Shared-Memory Systems", *Int. Symp. on System Synthesis (ISSS)*, pp. 1-6, Oct. 2001, Montréal, Canada.
- [53] Gangwar, A. & J.T.J. van Eijndhoven, *ECLIPSE Performance Analyzer, Multi-Processor Multi-Tasking Performance Measurement and Visualization*, Nat.Lab. Technical Note NL-TN 2001/9, Philips Research internal report, Jan. 2001, Eindhoven, The Netherlands.
- [54] Garg, M., *Eclipse Communication Architecture, Coprocessor Bus Speed and Power Studies*, Nat.Lab. Technical Note NL-TN 2000/330, Philips Research internal report, Aug. 2000, Eindhoven, The Netherlands.
- [55] Gelsinger, P.P., "Microprocessors for the New Millennium—Challenges, Opportunities and New Frontiers", *Int. Solid-State Circuits Conf. (ISSCC)*, Intel keynote, Feb. 4-8, 2001, San Francisco, USA.
- [56] Gigascale Silicon Research Center, *Metropolis: Design Environment for Heterogeneous Systems*, <http://www.gigascale.org/metropolis>.
- [57] Goel, M., *Process Networks in Ptolemy II*, MSc Report, Technical Report UCB/ERL M98/69, University of California, Dec. 16, 1998, Berkeley, CA, USA.

- [58] Goossens, K.; J. Dielissen & A. Radulescu, "Aethereal Network on Chip: Concepts, Architectures, and Implementations", *IEEE Design & Test of Computers*, vol. 22, no. 5, pp. 414-421, Sept.-Oct. 2005.
- [59] Gries, M., "Methods for Evaluating and Covering the Design Space during Early Design Development", *Integration, the VLSI Journal*, vol. 38, no. 2, pp. 131-183, 2004.
- [60] Grötter, T. et al., *System Design with SystemC*, Kluwer Academic Publ., 2002.
- [61] Gupta, R.K. & S.Y. Liao, "Using a programming language for digital system design", *IEEE Design and Test of Computers*, vol. 14, no. 2, pp. 72-80, April-June 1997.
- [62] De Haan, G. & E.B. Bellers, "De-interlacing – An overview", *Proc. IEEE*, vol. 86, no. 9, pp. 1839-1857, Sept. 1998.
- [63] De Haan, G. & R. Lodder, "De-interlacing of video data using motion vectors and edge information", *Digest of the ICCE'02*, pp. 70-71, June 2002.
- [64] Halambi, A.; P. Grun, V. Ganesh & A. Khare, "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability", *Proc. Design, Automation and Test in Europe (DATE)*, pp. 485-490, March 9-12, 1999, Munich, Germany.
- [65] Haverinen, A.; M. Leclercq, N. Weyrich & D. Wingard, *White paper for SystemC based SoC Communication Modeling for the OCP Protocol*, Version 1.0, Oct. 14, 2002. <http://www.ocpip.org>.
- [66] Hekstra, G.J.; G.D. La Hei, P. Bingley & F.W. Sijstermans, "TriMedia CPU64 Design Space Exploration", *Int. Conf. on Computer Design (ICCD)*, pp. 599-606, Oct. 10-13 1999, Austin, Texas, USA.
- [67] Hennessy, J.L. & D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publ., San Mateo, CA, 1990.
- [68] Hoare, C.A.R., "Communicating Sequential Processes", *Communications of the ACM*, vol. 21, no. 8, pp. 666-677, Aug. 1978.
- [69] Hocevar, D.; S. Sriram & C-Y. Hung, "A Performance Simulation Approach for MPEG Audio/Video Decoder Architectures", *IEEE Int. Symp. on Circuits and Systems (ISCAS)*, vol. 6, pp. 203-206, June 1998, Monterey, CA, USA.
- [70] Hofmeister, C.R., *Dynamic Reconfiguration of Distributed Applications*, PhD thesis, Dept. of Computer Science, University of Maryland, 1993, College Park, USA.
- [71] Holliday, M.A. & C.S. Ellis, "Accuracy of memory reference traces of parallel computations in trace-driven simulation", *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, no. 1, pp. 97-109, Jan. 1992.
- [72] *Intel MXP5800/MXP5400 Digital Media Processor Architecture Overview*, Intel, Sept. 2003. <http://www.intel.com/go/imageprocessing>.
- [73] *International Technology Roadmap for Semiconductors*, ITRS public home page, <http://public.itrs.net/>.
- [74] ISO/IEC Int. Standard 13818-2, *Information Technology—Generic Coding of Moving Pictures and Associated Audio Information: Video*, MPEG-2 standard, Dec. 21, 2000, <http://www.chiariglione.org/mpeg/>.
- [75] ISO/IEC Int. Standard 14496-2, *Information Technology—Generic Coding of Audio-Visual Objects: Visual*, MPEG-4 standard, 2000.
- [76] Jaspers, E.G.T., *Architecture Design of Video Processing Systems on a Chip*, PhD thesis, Eindhoven University of Technology, April 2003, Eindhoven, The Netherlands.
- [77] Jaspers, E.G.T. & P.H.N. de With, "Architecture of Embedded Video Processing in a Multimedia Chip-set", *Proc. of IEEE Int. Conf. on Image Processing (ICIP 99)*, vol. 2, pp. 787-791, Oct. 1999, Kobe, Japan.
- [78] Jeffreys, R.; A. Anderson & C. Hendrickson, *Extreme Programming Installed*, Addison-Wesley, 2001.

- [79] Jouppi, N.P., "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", *Proc. 17th Int. Symp. Computer Architecture*, pp. 364-373, May 1990.
- [80] Kahn, G., "The Semantics of a Simple Language for Parallel Programming", *Proc. of Information Processing '74*, August 5-10, Stockholm, Sweden, North-Holland Publ. Co., pp. 471-475, 1974, Stockholm, Sweden.
- [81] Kahn, G. & D.B. MacQueen, "Coroutines and Networks of Parallel Programming", *Proc. of Information Processing '77*, North-Holland Publ. Co., pp. 993-998, 1977, Toronto, Canada.
- [82] Keppel, D. *Tools and Techniques for Building Fast Portable Thread Packages*, Technical Report UWCSE 93-05-06, University of Washington, 1993.
<ftp://ftp.cs.washington.edu/tr/1993/05/UW-CSE-93-05-06.PS.Z>.
- [83] Keutzer, K.; S. Malik, A.R. Newton, J.M. Rabaey & A. Sangiovanni-Vincentelli, "System-Level Design: Orthogonalization of Concerns and Platform-Based Design", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1523-1543, Dec. 2000.
- [84] Kienhuis, A.C.J., *Design Space Exploration of Stream-based Dataflow Architectures, Methods and Tools*. PhD thesis, Delft Technical University, 1999, Delft, The Netherlands.
- [85] De Kock, E.A. et al., "YAPI: Application Modeling for Signal Processing Systems", *Proc. 37th Design Automation Conf. (DAC)*, pp. 402-405, June 2000, Los Angeles, CA, USA.
- [86] Krishnan, R.; O.P. Gangwal, J.T.J. van Eijndhoven & A. Kumar, "Design of a 2D DCT/IDCT application specific VLIW processor supporting scaled and sub-sampled blocks", *VLSI Design Conf.*, pp. 177-182, Jan. 4-8, 2003, New Delhi, India.
- [87] Lavagno, L.; A. Sangiovanni-Vincentelli & E. Santovich, "Models of Computation for Embedded System Design", *NATO ASI Proc. on System Synthesis*, Sept. 1998, Il Ciocco, Italy.
- [88] Lee, E.A. et al., *Overview of the Ptolemy Project*, Technical Memorandum UCB/ERL MO1/11, University of California, EECS Dept., March 2001, Berkeley, CA, USA.
- [89] Lee, E.A. & S. Neuendorffer, *MoML – A Modeling Markup Language in XML – Version 0.4*, Technical Memorandum ERL/UCB M 00/12, University of California, March 14, 2000, Berkeley, CA, USA.
- [90] Lee, W. & C. Basoglu, "MPEG-2 Decoder Implementation on MAP-CA Mediaprocessor using the C Language", *Proc. of the SPIE: Media Processors 2000*, vol. 3970, Jan. 2000.
- [91] Leijten, J., *Real-Time Constrained Reconfigurable Communication between Embedded Processors*, PhD Thesis, Eindhoven University of Technology, Nov. 1998, Eindhoven, The Netherlands.
- [92] Lieverse, P.; P. van der Wolf & E. Deprettere, "A Trace Transformation Technique for Communication Refinement", *Proc. 9th Int. Symp. on Hardware/Software Codesign (CODES)*, pp. 134-139, April 25-27, 2001, Copenhagen, Denmark.
- [93] Lieverse, P.; P. van der Wolf, E. Deprettere & K.A. Visser, "A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems", *Journal of VLSI Signal Processing for Signal, Image, and Video Technology*, vol. 29, no. 3, pp. 197-206, Nov. 2001.
- [94] Lin, B. et al., "Synthesis of concurrent system interface modules with automatic protocol conversion generation", *Computer Aided Software/Hardware Engineering*, IEEE Press, 1994.
- [95] *List of EDA tools*, Christian-Doppler Pilot Laboratory for Design Methodology of Signal Processing Algorithms, July 2002, <http://www.nt.tuwien.ac.at/cdlab/EDATools.htm>.
- [96] Lodder, R., *Automated application configuration in the Eclipse architecture*, Internal Prommpt Project Document Nr. 255, Philips Research internal report, Nov. 2000, Eindhoven, The Netherlands.
- [97] Loeffler, C.; A. Ligtenberg & G. Moschytz, "Practical Fast 1-D DCT Algorithms with 11 Multiplications", *Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 988-991, May 1989, Glasgow, England.

- [98] Martin, G. & B. Salefski, "Methodology and technology for design of communications and multimedia products via system-level IP integration", *Proc. Design, Automation and Test in Europe (DATE)*, pp. 11-18, Feb. 23-26. 1998, Paris, France.
- [99] Mercer, C.W.; S. Savage & H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications", *Int. Conf. on Multimedia Computing and Systems (ICMCS)*, pp. 90-99, May 14-19, 1994, Boston, MA, USA.
- [100] Mihal, A. et al., "Developing Architectural Platforms: A Disciplined Approach", *IEEE Design and Test of Computers*, vol. 19, no. 6, pp. 6-16, Nov./Dec. 2002.
- [101] Muller, H.L., *Simulating computer architectures*, PhD thesis, Dept. of Computer Science, University of Amsterdam, Feb. 1993, Amsterdam, The Netherlands.
- [102] Nieuwland, A. et al., "C-HEAP: a Heterogeneous Multi-processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems", *Design Automation for Embedded Systems*, Kluwer Academic Publ., vol. 7, no. 3, pp 233-270, Oct. 2002.
- [103] Oka, M. & M. Suzuoki, "Designing and Programming the Emotion Engine", *IEEE Micro*, vol. 19, no. 6, pp. 20-28, Nov.-Dec. 1999.
- [104] De Oliveira, J.A. & H. van Antwerpen, "The Philips Nexpedia Digital Video Platform", *Winning the SoC Revolution*, G. Martin & H. Chang (Ed.), Kluwer Academic Publ., pp. 67-96, 2003.
- [105] *OMAP™ Platform: Overview*, Texas Instruments, <http://www.ti.com/sc/omap>.
- [106] *OpenMAX – The Standard for Media Library Portability*, Khronos Group, <http://www.khronos.org/openmax/>, 2006.
- [107] Otero Pérez, C.M.O.P.; M.J. Rutten, E.F.M. Steffens, J.T.J. van Eijndhoven & P. Stravers, "Resource reservations in shared-memory multiprocessor SoCs", *Dynamic and robust streaming in and between consumer-electronic devices*, Kluwer Academic Publ., P. van der Stok (Ed.), 2005.
- [108] Pimentel, A.D., *A Computer Architecture Workbench*, PhD thesis, Dept. of Computer Science, University of Amsterdam, Dec. 1998, Amsterdam, The Netherlands.
- [109] Pimentel, A.D.; C. Erbas & S. Polstra, "A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels", *IEEE Trans. on Computers*, vol. 55, no. 2, pp. 99-112, Feb. 2006.
- [110] Pimentel, A.D.; L.O. Hertzberger, P. Lieverse, P. van der Wolf & E.F. Deprettere, "Exploring Embedded-Systems Architectures with Artemis", *IEEE Computer*, vol. 34, no. 11, pp. 57-63, Nov. 2001.
- [111] *Platform Architect*, CoWare, <http://www.coware.com/products/platformarchitect.php>, 2006.
- [112] Pollack, F.J., "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies", *Proc. 32nd Int. Symp. on Microarchitecture (Micro 32)*, pp. 2, 1999, Haifa, Israel.
- [113] Pontius, T. & P. Klaproth, *CoReUse 4.2, Device Transaction Level (DTL) Protocol Specification*, RTG/CMD/2001/0299, Philips Semiconductors internal report, Feb. 14, 2005.
- [114] Rao, K.R. & P. Yip, "Discrete cosine transform; algorithms, advantages, applications", Academic Press, 1990.
- [115] Rathnam, S. & G. Slavenburg, "Processing the New World of Interactive Media—the Trimedia VLIW CPU Architecture", *IEEE Signal Processing Magazine*, vol 15, no. 2, pp. 108-117, March 1998.
- [116] Van Rompaey, K.; D. Verkest, I. Bolsens & H. de Man, "CoWare – A design environment for heterogeneous hardware/software systems", *Design Automation for Embedded Systems*, vol. 1, no. 4, pp. 357-386, Oct. 1996.
- [117] Rowson, J.A. & A. Sangiovanni-Vincentelli, "Interface-Based Design", *Proc. 34th Conf. on Design Automation (DAC)*, pp. 178-183, 1997, Anaheim, CA, USA.

- [118] Rutten, M.J., *Eclipse simulation technology*, Nat.Lab. Technical Note NL-TN 2002/246, Philips Research internal report, June 18, 2002, Eindhoven, The Netherlands.
- [119] Rutten, M.J., *On-Chip Communication Architectures, an Overview*, Nat.Lab. Technical Note NL-TN 278/99, Philips Research internal report, Oct. 15, 1999, Eindhoven, The Netherlands.
- [120] Rutten, M.J., *TKM coprocessor shell specification, version 5.0*, Philips Research Technical Note PR-TN 2003/00503, Philips Research internal report, July 2003, Eindhoven, The Netherlands.
- [121] Rutten, M.J. et al., "Eclipse: A Heterogeneous Multiprocessor Architecture for Flexible Media Processing", *IEEE Design and Test of Computers: Embedded Systems*, pp. 39-50, July/Aug. 2002.
- [122] Rutten, M.J.; J.T.J. van Eijndhoven & E.J.D. Pol, *A data processing system for media processing applications, includes multiple multi-tasking processors each with its own task scheduling means*, Granted patents WO-2003052597 A2, AU-2002353280 A1, EP-1459179 A2, US-20050081200 A1, CN-1602467 A, Dec. 14, 2001.
- [123] Rutten, M.J.; J.T.J. van Eijndhoven & E.J.D. Pol, "Design of Multi-Tasking Coprocessor Control for Eclipse", *10th Int. Symp. on Hardware/Software Codesign (CODES)*, pp. 139-144, May 2002, Estes Park, CO, USA.
- [124] Rutten, M.J.; J.T.J. van Eijndhoven & E.J.D. Pol, "Eclipse: heterogeneous multiprocessor architecture for flexible media processing", *Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (PDIVM)*, April 15, 2002, Fort Lauderdale, FL, USA.
- [125] Rutten, M.J.; J.T.J. van Eijndhoven & E.J.D. Pol, "Robust media processing in a flexible and cost-effective network of multi-tasking coprocessors", *14th Euromicro Conf. on Real-Time Systems*, June 2002, Vienna, Austria.
- [126] Rutten, M.J.; J.T.J. van Eijndhoven & E.J.D. Pol, *Data processing system passes unique identifier from one processing element to another processing element, for task execution*, Granted patents WO-2004077206 A2, EP-1609038 A2, Feb. 17, 2003.
- [127] Rutten, M.J.; J.T.J. van Eijndhoven & E.J.D. Pol, "Caching Techniques for Multi-Processor Streaming Architectures", *Workshop on Media and Signal Processors for Embedded Systems and SoCs (MASES)*, Sept. 22, 2004, Washington D.C., USA.
- [128] Rutten, M.J.; O.P. Gangwal, J.T.J. van Eijndhoven, E.G.T. Jaspers & E.J.D. Pol, "Application Design Trajectory towards Reusable Coprocessors: MPEG Case Study", *Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, Sept. 6-7, 2004, Stockholm, Sweden.
- [129] Rutten, M.J.; E.J.D. Pol, J.T.J. van Eijndhoven, K. Walters & G. Essink, "Dynamic reconfiguration of streaming graphs on a heterogeneous multiprocessor architecture", *IS&T/SPIE Electronic Imaging: Embedded Processors for Multimedia and Communications II*, vol. 5683, Jan. 17-18, 2005, San Jose, CA, USA.
- [130] Rutten, R.J., "The Dutch Open Telescope", *Small Telescopes in the Era of Large Telescopes*, T. Oswalt (Ed.), Kluwer, 2003, Dordrecht, <http://dot.astro.uu.nl>.
- [131] Schaumont, P.; S. Vernalde, L. Rijnders, M. Engels & I. Bolsens, "A Programming Environment for the Design of Complex High Speed ASICs", *Proc. 35th Conf. on Design Automation (DAC)*, pp. 315-320, June 15-19, 1998, San Francisco, CA, USA.
- [132] Schiffelers, R. et al., "Epics7b - a learn and mean concept", *IEEE ISPC GSPx*, 2003.
- [133] Sima, M.; S.D. Cotofana, J.T.J. van Eijndhoven, S. Vassiliadis & K.A. Vissers, "IEEE-compliant IDCT on FPGA-Augmented TriMedia", *Journal of VLSI Signal Processing*, vol. 39, no. 3, pp. 195-212, March 2005.
- [134] Stefanov, T.; C. Zissulescu, A. Turjan, B. Kienhuis & E. Deprettere, "System Design using Kahn Process Networks: the Compaan/Laura Approach", *Proc. Design Automation and Test in Europe (DATE)*, vol. 1, pp. 340-345, Feb. 16-20, 2004.
- [135] Stravers, P. & J. Hoogerbrugge, "Homogeneous Multiprocessing and the Future of Silicon Design Paradigms", *Proc. Int. Symp. on VLSI Technology, Systems, and Applications (VLSI-TSA)*, pp. 184-187, April 2001, Hsinchu, Taiwan.

- [136] Strik, M.T.J.; A.H. Timmer, J.L. van Meerbergen & G.J. van Rootselaar, "Heterogeneous Multiprocessor for the Management of Real-Time Video and Graphics Streams", *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pp. 1722-1731, Nov. 2000.
- [137] *SystemC User's Guide*, Open SystemC Initiative (OSCI), version 2.0, 2001.
- [138] Thompson, M. & A.D. Pimentel, "A High-Level Programming Paradigm for SystemC", *Proc. 4th Int. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, Springer LNCS Tutorial series, vol. 3133, pp. 530-539, July, 2004, Samos, Greece.
- [139] *TMS320 DSP Algorithm Standard, Rules and Guidelines*, white paper no. SPRU352E, Texas Instruments, Oct. 2002, <http://focus.ti.com/lit/ug/spru352e/spru352e.pdf>.
- [140] Van der Tol, E.B. & E.G.T. Jaspers, "Mapping of MPEG-4 decoding on a flexible architecture platform", *Media Processors 2002*, vol. 4674, Jan. 2002, San Jose, CA, USA.
- [141] Van der Tol, E.B.; E.G.T. Jaspers & R.H. Gelderblom, "Mapping of H.264 decoding on a multiprocessor architecture", *Media Processors 2003*, 2003, USA.
- [142] Vahid, F., "The Softening of Hardware", *IEEE Computer*, vol. 36, no. 4, pp. 27-34, April 2003.
- [143] Vanmeerbeeck, G.; P. Schaumont, S. Vernalde, M. Engels & I. Bolsens, "Hardware/Software Partitioning of Embedded System in OCAPI-xl", *9th Int. Symp. on Hardware/Software Codesign (CODES)*, pp. 30-35, April 2001, Copenhagen, Denmark.
- [144] Vassiliadis, S.; S. Wong, G. N. Gaydadjiev, K.L.M. Bertels, G.K. Kuzmanov & E. Moscu Panainte, "The Molen Polymorphic Processor", *IEEE Trans. on Computers*, vol. 53, no. 11, pp. 1363-1375, Nov. 2004.
- [145] Veendrick, H., *Deep-submicron CMOS ICs: from basics to ASICs*, 2nd Edition, Kluwer Academic Publ., 2000, Dordrecht, The Netherlands.
- [146] Vercauteren, S.; B. Lin & H. de Man, "Constructing Application-Specific Heterogeneous Embedded Architectures for Custom HW/SW Applications", *33rd Design Automation Conf. (DAC)*, pp. 521-526, June 1996, Las Vegas, Nevada, USA.
- [147] Webb, D.L.; A.L. Wendelborn & J. Vayssière, "A Study of Computational Reconfiguration in a Process Network", *7th Workshop on Integrated Data Environments Australia (IDEA'7)*, pp. 51-55, Feb. 2000, Victor Harbor, Australia.
- [148] Walters, K.; J.T.J. van Eijndhoven, G. Essink & M.J. Martijn Rutten, *An API for Run-Time Programming of Eclipse*, Philips Research Technical Note PR-TN 2003/00563, Philips Research internal report, Aug. 2003, Eindhoven, The Netherlands.
- [149] Van der Wolf, P.; P. Lieverse, M. Goel, D. La Hei & K.A. Vissers, "An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology", *7th Int. Workshop on Hardware/Software Codesign (CODES)*, pp. 33-37, May 1999, Rome, Italy.
- [150] Wu., Z. & W. Wolf, "Study of Cache Systems in Video Signal Processors", *IEEE Workshop on Signal Processing Systems (SIPS)*, pp. 23-32, Oct. 8-10, 1998, Cambridge, MA, USA.
- [151] *Xtensa Configurable Processors*, Tensilica, http://www.tensilica.com/products/xtensa_overview.htm.
- [152] Yau, D.K.Y. & S.S. Lam, "Adaptive Rate-Controlled Scheduling for Multimedia Applications", *ACM Multimedia 96*, pp. 129-140, Nov. 1996.
- [153] Živković, V.D. & P. Lieverse, "An Overview of Methodologies and Tools in the Field of System-level Design", *Proc. 2nd Int. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, Springer LNCS Tutorial series, no. 2268, pp. 74-88, 2002.
- [154] Zucker, D.F.; R.B. Lee & M.J. Flynn, "Hardware and Software Cache Prefetching Techniques for MPEG Benchmarks", *IEEE Trans. Circuits and Systems for Video Technology*, vol. 10, no. 5, Aug. 2000.

Samenvatting

Dit proefschrift presenteert “Eclipse”: een architectuurtemplate (sjabloon) voor de realisatie van heterogene multiprocessoren die uiteenlopende multimedia gegevensstromen kunnen verwerken.

Multimedia-apparatuur, zoals DVD-spelers, digitale televisies en moderne autoradio's, bevatten geavanceerde, specialistische chips die een groot deel van de functionaliteit verzorgen. Dergelijke chips worden in grote aantallen toegepast; minimalisering van de productiekosten is daarom van groot belang voor de chipfabrikant. Hoe kleiner het siliciumoppervlak per chip, hoe lager de productiekosten. De processoren die de digitale signalen bewerken vormen een belangrijk onderdeel van elke chip. In dit proefschrift onderscheiden we twee soorten processoren: programmeerbare processoren, die softwareprogramma's uitvoeren die ook na de fabricage van de chip nog aangepast kunnen worden, en daarnaast de functiespecifieke processoren, *coprocessoren*, waarbij de functie al wordt vastgelegd tijdens het ontwerp van de chip. Doordat de functie al van tevoren wordt gedefinieerd zijn coprocessoren aanzienlijk kleiner dan programmeerbare processoren terwijl hun rekenkracht aanzienlijk groter is.

Multimedia-applicaties volgen elkaar in snel tempo op. MP3-spelers moeten nu ook Windows WMA aan kunnen, mobiele telefoons zullen in de nabije toekomst hoge-resolutie video moeten kunnen weergeven. De complexiteit van multimedia chips is te groot om voor elke nieuwe toepassing een specifieke bijbehorende chip te ontwikkelen. Chipfabrikanten pakken dit probleem aan met de introductie van een *platform*: een generieke chiparchitectuur waarmee op relatief eenvoudige wijze passende chips voor verschillende producten met vergelijkbare functionaliteit vervaardigd kunnen worden.

De Eclipse-architectuur in dit proefschrift definieert een dergelijke architectuurtemplate voor het ontwerp van subsystemen die specifieke functies moeten uitvoeren. Zo'n Eclipse-subsysteem op een chip bestaat uit meerdere kleine functiespecifieke coprocessoren die op een flexibele manier kunnen worden gekoppeld om samen één enkele functie uit te voeren. Elke coprocessor verzorgt zo een deel van de functionaliteit. Vaak levert een coprocessor functionaliteit die toepasbaar is in verschillende applicaties, zodat hij kan worden toegepast in verschillende subsystemen. Anderzijds kan één enkele coprocessor verschillende applicaties uitvoeren binnen één enkel subsysteem. Een volledige applicatie bestaat uit een aaneenschakeling van coprocessoren die door software wordt geconfigureerd. De coprocessoren communiceren met elkaar via gebufferde gegevensstromen. Deze manier van werken implementeert het wiskundige model voor procesnetwerken van G. Kahn (1974). Eclipse bezit de flexibiliteit om met gege-

ven coprocessoren verschillende, mits gerelateerde, functies uit te voeren en biedt bovendien lage productiekosten en grote rekenkracht door de toepassing van functiespecifieke coprocessoren.

Deel I van dit proefschrift behandelt de generieke architectuur van Eclipse. Het gaat hier vooral om de *coprocessor-schil*, een hardwaremodule die alle communicatie van en naar de coprocessor afhandelt. De Eclipse coprocessor-schil is zodanig ontworpen dat hij kan worden toegepast voor elke coprocessor. Het ontwerpen van een nieuwe coprocessor wordt eenvoudiger doordat zoveel mogelijk algemene functionaliteit naar de schil is verschoven. Om het siliciumoppervlak van de schil te beperken en flexibiliteit in de toepassing te bereiken werd het ontwerp geparametriseerd. Voor elke coprocessor-schil kunnen verschillende parameters ingesteld worden, bijvoorbeeld het al dan niet aanwezig zijn van een cache-geheugen.

De coprocessor praat met zijn schil via zes interface-primitieven: twee voor lezen en schrijven van en naar buffers in de centrale geheugenmodule in het subsysteem, twee voor het synchroniseren van de lees-en schrijfacties met andere coprocessoren, één om de schil te vragen wat de volgende taak is voor de coprocessor, en één om via de schil te rapporteren aan de control-processor, bijvoorbeeld problemen of vooraf-gedefinieerde gebeurtenissen zoals de start van een nieuw videobeeld in de binnenkomende gegevensstroom. Het gebruik en de implementatie van deze primitieven wordt in detail beschreven in hoofdstukken 3 t/m 6.

Deel II gaat nader in op het Eclipse-simulatiemodel. Dit SystemC-model dient enerzijds om de architectuur te valideren, anderzijds om de programmering van nieuwe Eclipse-applicaties te verifiëren voordat de chip beschikbaar komt. Het ontwerp van het simulatiemodel wordt beschreven in hoofdstuk 9, met nadruk op de manier waarop het simulatiemodel gebruik maakt van parameters zoals het type en aantal coprocessoren in een specifiek Eclipse-subsysteem. Hoofdstuk 10 beschrijft de simulaties die gedaan zijn om de Eclipse-concepten van gedistribueerde synchronisatie en gedistribueerde taakverstrekking te valideren. Deze experimenten werden uitgevoerd op een Eclipse-ontwerp voor MPEG-2 video decoding. Het onvoorspelbare, data-afhankelijke gedrag van deze applicatie maakt haar bij uitstek geschikt om de Eclipse-architectuur te toetsen.

Deel III beschrijft een drietal Eclipse-subsystemen, respectievelijk voor video-compressie in digitale televisie, beeldbewerking in DVD-spelers en geluidsbewerking in geavanceerde autoradio's. Tezamen bestrijken deze drie subsystemen een breed applicatiegebied en demonstreren ze de ruime toepasbaarheid van Eclipse. De eisen voor videocompressie en beeldbewerking zijn extreem en lopen ver uiteen: videocompressie van hoge-definitie digitale televisiesignalen vereist enorme rekenkracht terwijl beeldbewerking voor DVD-spelers relatief eenvoudig is en weinig additioneel silicium toestaat. In beide gevallen brengt Eclipse, ten opzichte van gangbare functiespecifieke implementaties, extra flexibiliteit in het programmeren van verschillende applicaties en in het eenvoudig toevoegen van nieuwe coprocessoren. De derde toepassing, geluidsbewerking in autoradio's,

wordt behandeld in het laatste hoofdstuk en laat zien hoe Eclipse kan worden ingepast in een architectuur voor applicaties op meerdere programmeerbare processoren. Hun onderlinge communicatie via interface-primitieven werd geïnspireerd door Eclipse. De architectuur illustreert ook hoe een Eclipse-subsysteem naadloos kan samenwerken met signaalbewerking in software. De combinatie van functie-specifieke coprocessoren en programmeerbare processoren geeft de ontwerper de mogelijkheid een optimale afweging te maken tussen efficiëntie en flexibiliteit in de ontwikkeling van een nieuw platform.

Curriculum Vitae

Martijn Rutten was born on October 19, 1973 in Rumpt, The Netherlands. In 1998, he obtained his MSc. in Electrical Engineering at the Eindhoven University of Technology on modeling a 3D graphics pipeline for the 64-bit Tri-Media processor. This work, carried out at Philips Research Laboratories Eindhoven, introduced him to media processing and system architecture.

Afterwards, Martijn joined Philips Research and took part in the definition of Eclipse—culminating in a 3-month visit to Bangalore, India to transfer Eclipse technology to Philips Semiconductors.

From 2002 onwards, Martijn’s research activities moved to execution architecture of media-processing systems. Hereto, Martijn spent four months at the Philips’ Digital Systems Laboratory (PDSL) to develop an overview of the execution architecture of DVD recorder systems. From September 2004 onwards, Martijn worked as system architect in the SoC Design Technology Department (CTO/SoC-DT) of Philips/NXP Semiconductors. Here he was involved in the design of multiprocessor SoC subsystems, in particular with the Sea-of-DSP architecture and software streaming infrastructure. At this time, Martijn took up development of TimeDoctor—an open source project on performance visualization—implemented as a plug-in for the Java Eclipse environment. While the widely popular Java Eclipse environment has no relation to the Eclipse architecture in this thesis, clearly, “Eclipse”-based solutions are a recurring theme in Martijn’s career. Presently, Martijn is taking on a next challenge as co-founder of Vector Fabrics, a technology start-up company in Eindhoven.

Whenever Martijn is not at work, he is most likely to be found in his kayak exploring white-water rivers in remote corners of the world.