EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Predictable multi-processor system on chip design for multimedia applications

Citation for published version (APA):
Shabbir, A. (2011). *Predictable multi-processor system on chip design for multimedia applications*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Electrical Engineering]. Technische Universiteit Eindhoven.
https://doi.org/10.6100/IR716801

DOI:
10.6100/IR716801

Document status and date:
Published: 01/01/2011

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 22. Aug. 2022

Predictable Multi-processor System on Chip Design for Multimedia Applications

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 10 november 2011 om 16.00 uur

door

Ahsan Shabbir

geboren te Sialkot, Pakistan

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. H. Corporaal

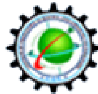
Copromotoren:
dr.ir. B. Mesman
en
dr.ir. A. Kumar

Predictable Multi-processor System on Chip Design for Multimedia Applications
/ by Ahsan Shabbir. - Eindhoven : Eindhoven University of Technology, 2011.
A catalogue record is available from the Eindhoven University of Technology Library
ISBN 978-90-386-2770-0
NUR 959
Trefw.: multi-programmeren / elektronica ; ontwerpen / multi-processoren /
ingebedde systemen.
Subject headings: dataflow graphs / electronic design automation /
multi-processing systems / embedded systems.

Predictable Multi-processor System on Chip Design for Multimedia Applications

Committee:

prof.dr. H. Corporaal (promotor, TU Eindhoven)
dr.ir. B. Mesman (co-promotor, TU Eindhoven)
dr.ir. A. Kumar (co-promotor, National University of Singapore)
prof.dr.ir. R.H.J.M. Otten (TU Eindhoven)
prof.dr. P. Marwedel (TU Dortmund)
dr. K. Bertels (TU Delft)
prof.dr.ir. R.J. Bril (TU Eindhoven)



The work in this thesis is supported by Eindhoven University of Technology Eindhoven and NESCOM (National Engineering and Scientific Commission Islamabad Pakistan).

iPhone and iPad are registered trademarks of Apple Inc.
Xbox 720 is registered trademark of Microsoft Corporation.
Philips TV is registered trademark of Philips Consumer Lifestyle.

© Ahsan Shabbir 2011. All rights are reserved. Reproduction in whole or in part is prohibited without the written consent of the copyright owner.

Printing: Printservice Eindhoven University of Technology

Abstract

Predictable Multi-processor System on Chip Design for Multimedia Applications

The design of multimedia systems has become increasingly complex due to consumer requirements. Consumers demand the functionalities offered by a huge desktop computer from these systems. Many of these systems are mobile so power consumption and size of these devices should be small. These systems are increasingly becoming multi-processor based for the reasons of power and performance. Applications execute on these systems in different combinations also known as use-cases. Applications may have different performance requirement in each use-case. The multi-processor based platform should have predictable behaviour so that we can guarantee its performance. Furthermore, the platform should be shared between different applications so that it can be used efficiently.

In this thesis, techniques have been developed to design and manage these multi-processor based systems efficiently. One of the contributions of this thesis is a communication assist. The communication assist presented in this thesis not only decouples the communication from computation but also provides timing guarantees. Based on this communication assist, an MPSoC platform generation technique is presented that can synthesize a platform capable of meeting the throughput constraints of multiple applications within a given set of use-cases. The tool can generate the implementations for FPGAs with the help of commercially available synthesis tools.

Further in the thesis, a fast and scalable simulation methodology is introduced that can simulate the execution of multiple applications on an MPSoC platform. It is based on parallel execution of SDF (Synchronous Dataflow) models of applications. The simulation methodology uses Parallel Discrete Event Simulation (PDES) primitives and it is termed as Smart Conservative PDES. Most PDES approaches fall under one of two categories – conservative and optimistic. In this thesis, a smart conservative approach is proposed, that is intelligent to figure out when the sequential program execution can be set aside for improved effi-

ciency. We have developed a mechanism which on every simulation step checks whether continuing the simulation with incomplete information can result in a causality error. By default, conservative PDES is used and as soon as it is found that causality errors can be avoided with non-sequential execution, the simulation proceeds and does not follow sequential execution. This mechanism is called as *smart conservative*. The methodology generates a parallel simulator which is synthesizable on FPGAs. The user can also select the scheduling policy which is to be implemented on each processor of the platform. The generated platform can execute the applications and their performance can be predicted. For a presented use-case consisting of two applications, the technique is 15% faster than Conservative PDES. It is also shown that the speedup increases with increase in number of applications.

The resources provided by the MPSoC platforms are shared between the applications. A run-time manager is needed that can distribute the resources of the platform in such a way that all the applications get their desired resources and no application can monopolize the resources. This thesis presents such a run-time resource management technique that can share the MPSoC platform between multiple applications. Two versions of distributed resource managers are presented which are scalable with respect to the number of applications and processors. The resource managers can be distinguished on the basis of their budget enforcement protocols. The first type named as Credit-Based RM is useful for applications which require very strict timing constraints. The credit-based RM is a type of budget-based scheduler where budgets are assigned to tasks in a large replenishment interval. Each processor executes the tasks according to assigned budgets and these budgets are reloaded at the end of each replenishment interval. The second type of RM is called Rate-based RM. In rate-based RM, the rate of executions of tasks is kept at a predetermined value. Rate-based RM is useful for applications which allow their performance to be more than a minimum constraint. Streaming encoders can employ rate-based RM. These encoders can encode at higher rates whenever there is an abundance of compute resources in the platform.

Using the contributions mentioned in this thesis, a designer can design and implement predictable multi-processor based systems capable of satisfying throughput constraints of multiple applications in given set of use-cases, and employ resource management strategies to deal with dynamism in the applications.

Acknowledgments

The work in this thesis has been conducted with help and guidance of a number of people. I would like to express my sincere gratitude to all those people who helped me during my PhD.

First of all, I would like to thank Prof. Henk Corporaal for providing me opportunity to conduct research in Electronic systems group. Throughout the duration of PhD, he guided me and provided ideas to improve my work. Despite being very busy, he always had time to discuss the work. His critical thinking helped me improve the work. He always provided feedback on my work on time.

I am also very grateful to Bart Mesman for his guidance in my work. We also did an internship in ASML. The internship helped me improve my technical skills and work in industrial environment. I would especially like to thank Akash Kumar for his constant support throughout the PhD duration. He not only guided me but he is also a very good friend.

Further, I like to thank Sander Stuijk for his time he spent in technical discussions with me. He was also my officemate throughout PhD duration. He motivated me to improve the quality of my work and always pushed me to go an extra yard in learning and creativity.

I would like to thank my family and friends for their support throughput the period. I would especially like to thank my parents, my brother and sister without whom I would not be able to achieve this result. I dedicate this thesis to my father who despite his ill health, allowed me to travel abroad and continue my PhD study. My special thanks to my wife Sara, children Aleena and Noor Fatima who had to suffer due to my late sittings in the office.

During past few years my stay at Electronic systems group has been very pleasant. I would like to thank my group members especially our group leader Prof. Ralph Otten for their support. I really enjoyed the friendly discussions with

my officemate Yang Yang and other members at the coffee break. I would like to thank the secretaries of our group, Rian and Marja for arranging my accommodation and they have been very helpful. Sander also helped me in translating and filling dutch forms.

I also want to thank my Pakistani friends living in the Netherlands. I would specially like to thank my friends in the Pakistani mosque. I also want to thank the people of Netherlands in general; Netherland is a beautiful country with very nice people and I enjoyed my stay here. In the end, I would like to thank my God who gave me strength and health to finish my work. This would not have been possible without the help and willingness of God.

Ahsan Shabbir
Eindhoven, November 2011

Contents

Abstract i

Acknowledgments iii

1 Trends and Challenges in Multimedia Systems 1
- 1.1 Trends in High Performance Media Processing . . . 2
- 1.2 Trends in Processor and Platform Architectures . . . 5
  - 1.2.1 Globally Asynchronous Locally Synchronous . . . 5
  - 1.2.2 The Emergence of Multi-processors . . . 6
  - 1.2.3 Heterogeneous vs Homogeneous Architectures . . . 8
  - 1.2.4 Homogeneous Architectures . . . 9
  - 1.2.5 Heterogeneous Architectures . . . 11
- 1.3 Predictable MPSoC Design . . . 13
- 1.4 Importance of Application Model and Specification . . . 15
- 1.5 Introduction to SDF Graphs . . . 18
  - 1.5.1 Modelling Auto-concurrency . . . 19
  - 1.5.2 Modelling Buffer Sizes . . . 20
- 1.6 Predictable MPSoC Template . . . 21
- 1.7 Key Contributions of the Thesis . . . 22
  - 1.7.1 Communication Assist . . . 23
  - 1.7.2 Design Algorithm . . . 24
  - 1.7.3 Distributed Resource Management . . . 24
  - 1.7.4 Distributed Simulation on FPGA . . . 25
- 1.8 Design Flow . . . 26
- 1.9 Thesis Overview . . . 28

2 Communication Assist Architecture **29**
2.1 Existing CA Architectures . . . 30
2.2 Novel Communication Assist Architecture . . . 32
2.3 CA Architecture . . . 33
2.3.1 Circular Buffer Management . . . 34
2.3.2 Programmability and Operation of CA . . . 35
2.4 Conservative SDF model of CA . . . 37
2.5 Hardware Implementation . . . 38
2.6 Experiments and Case Study . . . 39
2.6.1 Analytical Models of Applications and Architecture . . . 39
2.6.2 Improvement in Memory Usage . . . 42
2.6.3 Run-time Configuration of CA . . . 43
2.6.4 Reduction in Communication Latency . . . 43
2.7 Related Work . . . 44
2.8 Conclusions . . . 46

3 Predictable Multi-processor Design Approach **47**
3.1 Motivating Example . . . 48
3.2 Problem Statement and Model Definition . . . 50
3.3 Design Algorithm . . . 51
3.4 Task Scheduling and Throughput Measurement . . . 55
3.5 Experiments and comparison with other techniques . . . 58
3.6 Related Work . . . 64
3.7 Conclusions . . . 66

4 Multi-processor Platform Synthesis **67**
4.1 Architecture Template . . . 69
4.1.1 Processing Element . . . 70
4.1.2 Memories . . . 70
4.1.3 Communication Assist . . . 70
4.2 Design Flow . . . 70
4.2.1 H/W Generation . . . 72
4.2.2 S/W Generation . . . 74
4.3 Tool Implementation . . . 77
4.4 Experiments and Results . . . 78
4.5 Related Work . . . 80
4.6 Conclusions . . . 81

5 Distributed Resource Management **83**
5.1 Application and Architecture Modelling . . . 85
5.2 Motivating Example . . . 87
5.3 Proposed Resource Managers . . . 89
5.3.1 Credit-Based Resource Manager . . . 90
5.3.2 Rate-Based Resource Manager . . . 90

5.4 Comparison Between the Resource Managers 92
5.4.1 Admission of a New Application 93
5.4.2 Application stopped by the User 94
5.4.3 Variation in Actor Execution Times 96
5.4.4 Variation in Application Throughput Constraint 97
5.4.5 Buffer Requirement 97
5.4.6 Processor Utilization of Resource Managers 99
5.4.7 Scalability of RMs 100
5.5 Related Work 100
5.6 Conclusions 102

6 Distributed Simulation on FPGA 103
6.1 Simulation Platform Generation 104
6.2 PDES For Multiple Applications 108
6.2.1 Deadlocks 108
6.2.2 Smart Conservative PDES 110
6.2.3 Motivating Example 110
6.2.4 Dynamic Actor Arbitration 111
6.3 FPGA implementation, Experiments and Results 112
6.3.1 DSE Case Study 113
6.3.2 Scalability 116
6.4 Simulation of Dynamic Scheduling Policies 118
6.5 Related Work 120
6.6 Conclusions 121

7 Conclusions and Future Work 123
7.1 Conclusions 123
7.2 Future Work 125

Bibliography 127

Glossary 141

Curriculum Vitae 145

List of Publications 147

CHAPTER 1

Trends and Challenges in Multimedia Systems

Science has invented many things. The triumphs of science are too many to be counted. Some of the latest triumphs of science, like computers, are really wonderful. Computers have contributed to most aspects of our society since their emergence over half a century ago. However, the majority of computers in our daily lives are not the general personal computers we use in our offices and schools etc. Instead, they are found in the embedded systems constructed to do a particular job. They are in washing machines, auto mobiles, mobile phones, multimedia systems, and navigation systems, just to name a few. Multimedia systems in particular are becoming increasingly more popular and satisfy the information and entertainment needs of their users. The functionality offered by these embedded multimedia systems increases and this makes their design a very challenging job. These embedded multimedia systems consist of many hardware and software components which need to be verified. Most of the design effort is dedicated towards verification of these systems. A particular challenge with embedded systems design is to meet the timing constraints of applications mapped onto these platforms. In addition, the power consumption of these systems must be low as most of these embedded systems are mobile. This thesis proposes solutions to some of these design challenges.

The contributions of this thesis include a predictable communication assist, a technique that generates MPSoC platforms capable of satisfying the throughput constraints of multiple applications, a simulation framework and distributed run-time resource managers. These contributions combine to provide a complete design flow which consists of analysis, simulation, and synthesis for implementa-

tion on FPGAs.

In the next section, we look at major trends and challenges in the application domain of multimedia systems. Section 1.2 presents the trends in architectures of these multimedia systems. Section 1.3 emphasizes the need of predictability in the design of multimedia systems. Section 1.4 advocates the importance of application models in the design process and section 1.5 introduces Synchronous Dataflow model which is the application model used in this thesis. Section 1.6 presents the proposed architecture template for multimedia systems. Section 1.7 states the key contributions of the thesis and section 1.8 presents a predictable design flow which can ease the multimedia system design process. Finally, section 1.9 gives a brief overview of the thesis.

1.1 Trends in High Performance Media Processing

Modern multimedia systems, such as smart phones and PDAs offer an increasing amount of functionality to their end-users by simultaneously executing a number of real-time/non-real-time stream processing applications. Most of these applications deal with the "content" of the users. The content is a combination of forms like text, audio, video, and pictures. This combination is termed as multimedia. Figure 1.1 shows some examples of modern multimedia systems. The number of features in a multimedia system is increasing. For example, a mobile phone that was traditionally meant to only support voice calls now provides video conferencing features, streaming of television programs using 3G networks, GPS, video camera, personal agenda, wireless connectivity (WiFi) etc. The number of applications executing on these multimedia systems doubles roughly every two years [ITR07]. The processing power needed by these applications is huge.

The Apple iPad is an example of an embedded multimedia system supporting a large number of applications. The Apple iPad is originally a tablet computer. It can be used as a gaming console but it can also be used to watch movies, listen to music, or browse the Internet. The Apple iPhone is another example of a multimedia platform. Its touch screen can be used to watch movies, and its built-in GPS receiver can be used for navigation. It also has an mp3 player to play songs, a camera to take pictures, and above all it has communication circuitry to make phone calls.

Some people refer to it as the convergence of information, communication and entertainment [BMSM96]. Devices which were meant for only one of them now support all three of them. Further, there is a proliferation of standards in media processing. Take video as an example, H.264, MPEG-4, MPEG-2, H.263, VC-1, and AVS are some of the video standards being used in the industry. Similarly infrared, GPS, WiFi, and Bluetooth are standards for connectivity. A mobile phone has to support multiple bands like GSM 850, GSM 900, GSM 1800 and GSM 1900. The embedded systems have to support all these standards. Having each standard as a separate hardware module increases the power and cost budgets

(a) iPad

(b) iPhone

(c) Philips TV

(d) Xbox 720

Figure 1.1: Some examples of Embedded multimedia systems

of the device.

The concurrent execution of applications onto multi-processor platforms adds another dimension to the challenges in designing multimedia systems. Many of the applications mapped onto multimedia systems have to execute concurrently with other applications in different combinations. We define each such combination of simultaneously active applications as a *use-case*. It is also known as mode in literature [SKC00]. The designer has to make sure that each use-case has satisfactory performance and the problem can be termed as *Design for use-case*.

There are four main complex use-case challenges that have to be overcome. The first of these is designing for sufficient bandwidth. The system must have enough memory, bus/network, and processing bandwidth to handle the amount of information coming in and going out of the system without experiencing any system hangs. The next challenge is latency. Users expect applications to open instantly and to move between applications with no delay. Designing for the smallest possible latency in design requires efficient hardware resource utilization as well as highly optimized software.

The third challenge is achieving seamless transitions between applications: In other words, having multiple applications in the same handset all sharing resources without interfering or interrupting each other. The final challenge is designing for all-day battery life. Designing within the confines of today's available battery power while providing the performance needed for today's top applications, is a challenge. The power budget for mobile phones is a mere 1 W [vB09]. Even for other plugged multimedia systems, power consumption has become a global concern with growing awareness among people to reduce the energy consumption. To find the optimal balance between power consumption and performance, a multimedia system has to be designed with a holistic power management approach, one that looks at the entire system and not just on a component-by-component basis.

In addition to the problems related to the ability to design current and future systems, these systems must also be designed with low cost and low time to market (TTM). This requirement is largely a response to an ever decreasing *product lifetime*, where the consumer replaces old products much more frequently as compared to any other discipline. Mobile phone manufacturers, for example, release two major product lines per year as compared to one just a few years ago [Hen03]. Furthermore, as the product lifetime decreases, the units sold must still generate enough profit to cover the rising cost of manufacturing and design.

The requirements put forward by the multimedia applications have an influence on the architectures of these multimedia systems. In the next section, we review the trends in the architectures of these multimedia systems.

1.2 Trends in Processor and Platform Architectures

The immense performance requirements under strict power constraints imposed by applications have led to new trends in computer architecture. Moore's law [Moo98] has been a great motivation for designers against this havoc in diversity of applications. Following subsections briefly describe this trend.

1.2.1 Globally Asynchronous Locally Synchronous

Moore's law predicted the exponential increase in transistor density as early as 1965. The ongoing reduction in transistor size is enabling the designers to have more functional units and storage on the chip, but increasing resistive delay is slowing communication within the chip as shown in Figure 1.2. The figure shows the increase in global wiring signal propagation delay with decreasing feature size. Smaller transistor feature size also results in higher clock speed, enabling faster functional units. Technology has also allowed other capabilities of electronics circuit like memory capacity to improve almost at exponential rate. However, the relative increase in wire delay means that the distance travelled by the signals in one clock cycle has decreased, resulting in evolution of Globally Asynchronous Locally Synchronous (GALS) circuits.

GALS circuits combine the benefits of synchronous and asynchronous systems. The whole design is divided into blocks with each block having its own clock. Connections between these synchronous blocks are asynchronous. Further research in GALS led to multi-processor systems.

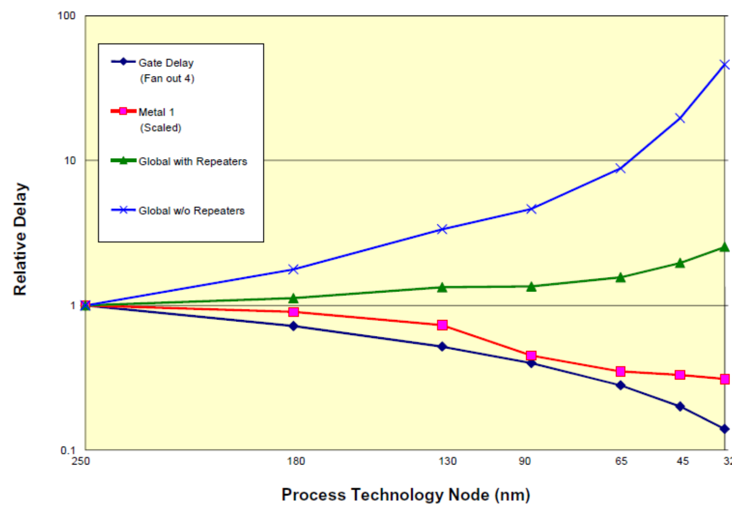


Figure 1.2: Delay for global wiring versus feature size [ITR05].

1.2.2 The Emergence of Multi-processors

The microprocessor has evolved dramatically from the simple 2300 transistors of the Intel 4004 to approximately billions of transistors, today. During this period, whenever there were slight hiccups in the progress, land mark inventions helped keep computer architecture on track. Early microprocessors processed one instruction from fetch to retirement before starting on the next instruction. Pipe-lining, which had been around at least since the 1940s in mainframe computers, was an obvious solution to that performance bottleneck. The latency to get instructions and data from off-chip memory to the on-chip processing elements was too long. This resulted into an on-chip cache. The first commercially viable microprocessor to exhibit an on-chip cache was the Motorola MC68020, in 1984. The benefits of pipe-lining are lost if conditional branches produce pipe-line stalls. Hardware branch predictors did not show up on the microprocessors until the early 1990s. In the pursuit of more parallelism, efforts continued to keep the functional units as busy as possible. The mechanism to get around this problem was known as early as 1960s, out-of-order processing. However it was restricted to high-performance scientific computation.

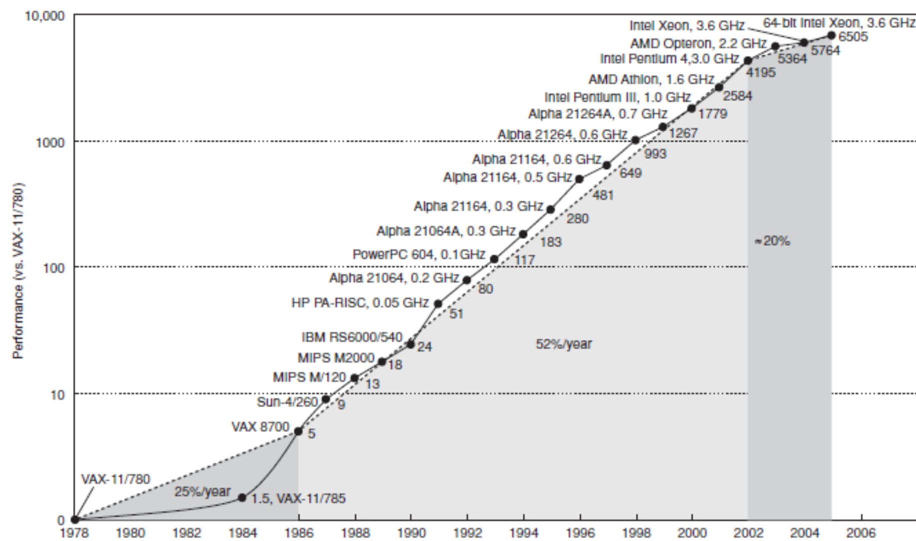


Figure 1.3: Growth in processor performance [HP06].

Further advances in computer architecture include clusters of functional units (Alpha 21264 in late 1990s), multiple levels of caches (First used in Alpha 21064, 1994) and simultaneous multi-threading. While on one hand, the hardware designers have been able to provide bigger and faster means of the processing, on the

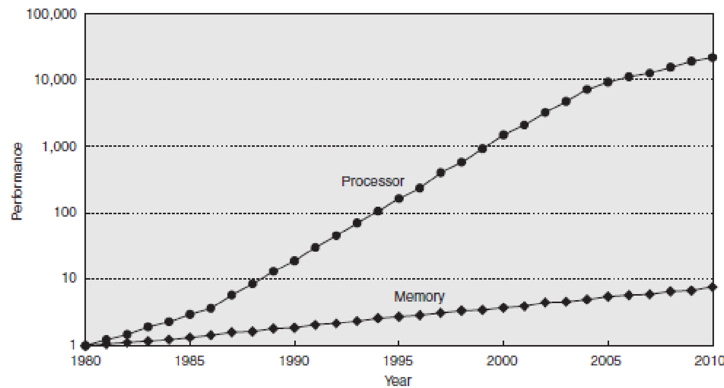


Figure 1.4: Gap in performance between memory and processors plotted over time [HP06].

other hand, the application developers have relied on improvements in technology (clock frequency) to meet the constraints of the applications. But this free lunch did not last very long. This is evident from Figure 1.3. The figure shows the performance of processors relative to VAX 11/780 as measured by the SPECint benchmarks. The figure shows that between year 1980 and 2002, the performance of processors increased at the rate of 52% per year due to the advances mentioned above. However since 2002, the average increase in performance is only 20% per year due to the triple hurdles of maximum power consumption of air-cooled chips, little instruction level parallelism left to exploit efficiently, and memory latency. Intel canceled its high-performance uniprocessor projects and joined IBM and Sun in declaring that the road to higher performance would be via multiple processors per chip rather than via faster uniprocessors. The architectures shown in Figure 1.3 are also termed as Latency-oriented [GK10] as they employ sophisticated components e.g. caches, branch prediction, out-of-order execution etc. to reduce the overall execution time of the program. Figure 1.4 shows the performance gap between processor and memory. The gap is due to the fact that the memory has to be as large as possible to meet the demands of applications and large memories cannot be faster. As explained earlier, this gap in performance has been tried to be filled with the help of multiple levels of caches and branch predictors. However, all these units consume lots of power at high frequencies and Intel's P4 processor crossed the 100 W mark. The cost of cooling the processor increased and methods like liquid cooling are employed. The three walls against single processor performance namely, ILP, memory, and power stopped the single processor innovations. Chip manufacturers are therefore shifting towards designing multi-processor chips operating at a lower frequency. The ITRS [ITR10] has predicted this trend as shown in Figure 1.5. The figure shows super linear increase in number of processing elements/System-on-Chip (SoC) in coming years.

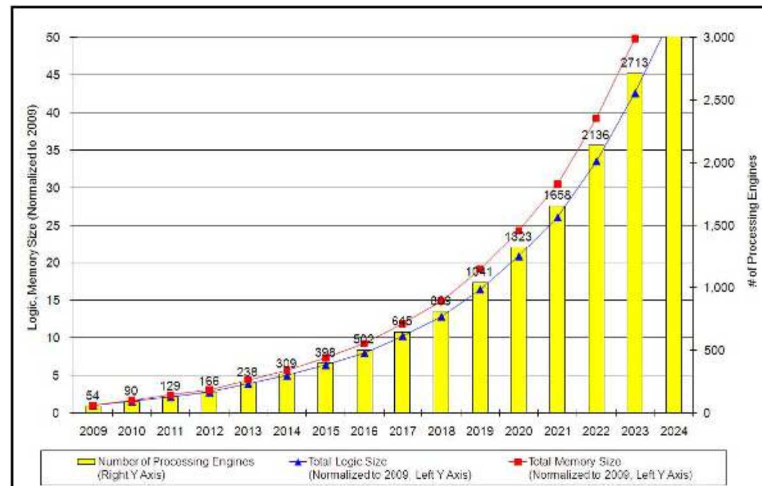


Figure 1.5: SoC consumer Portable design complexity trends [ITR10].

IBM introduced this feature in 2000, with two processors on its G4 chip. Intel reports that under-clocking a single core by 20 percent saves half the power while scarifying just 13 percent of the performance. This implies that if work is divided between two processors running at 80 percent clock rate, we may get 74 percent better performance for the same power.

In contrast to latency-oriented architectures, throughput-oriented architectures achieve even higher levels of performance by using many simple, and hence small, processing cores. The individual processing units of a throughput-oriented chip typically execute instructions in the order they appear in the program rather than trying to dynamically reorder instructions for out-of-order execution. They also generally avoid speculative execution and branch prediction.

In the next section, we further see the type of programs which can benefit from latency-oriented and throughput-oriented architectures.

1.2.3 Heterogeneous vs Homogeneous Architectures

Amdahl's law [Amd67] is used to find maximum expected improvement to an overall system when only a part of the system is improved. The speedup is defined as the original execution time divided by the enhanced execution time. According to Amdahl's law, assume that a fraction f of a program's execution time is parallelizable then the fraction $(1 - f)$ is not parallelizable and hence sequential. The speed up is defined as

$$S = \frac{1}{(1 - f) + \frac{f}{n}} \tag{1.1}$$

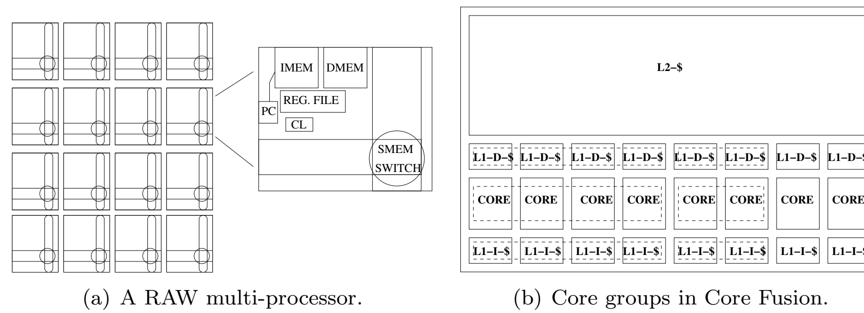
Assume that a program is 50% parallelizable then the maximum achievable speedup is a factor 2 no matter how much speedup we can achieve on the parallelizable part. According to Amdahl's law, the serial part of a program is always limiting factor in the speedup equation. The software model in Amdahl's law is simple and assumes either completely sequential code or completely parallel code. Amdahl's law has been extended for multi-processors by [HM08]. The authors conclude that heterogeneous multi-cores perform better than homogeneous multi-cores for lower degrees of parallelism, and for higher levels of parallelism homogeneous multi-cores are better than heterogeneous multi-cores. All the cores in a homogeneous multi-core are similar while in heterogeneous multi-cores, some cores use additional chip resources so that they can achieve more performance as compared to other cores on the chip. The cores utilizing more resources are good for sequential parts of the program while the parallelizable code executes on the smaller/slower cores. Dynamic multi-core chips [IKKM07, HWO98, SBV98] are designed to get best of both worlds. In the sequential mode, the cores in the chip can combine and become a bigger core so that it can achieve better performance. Similarly in the parallel mode, the bigger core can split into smaller cores and the multi-core can still give better performance. Note that the downside of the model presented by [HM08] is that it does not account for cache capacity, interconnect and synchronization overhead.

Amdahl's law has been augmented with the notion of critical sections by [EE10]. The authors present a simple analytical (probabilistic) model that reveals that the impact of critical sections can be split up in a completely sequential and a completely parallel part. The authors argue that the parallel performance is not only limited by the sequential part as suggested by Amdahl's law but it is also limited by critical sections. The paper shows that the performance benefits of heterogeneous multi-core processors may not be as high as suggested by [Amd67, HM08], and may even be worse than homogeneous multi-processors for workloads with many and large critical sections and high contention probabilities. The paper concludes that the execution of critical sections through a large core may yield substantial speedups. This emphasizes the importance of critical sections and synchronization between the cores. This also shows that using a heterogeneous system with several large cores on a chip can offer better speedup than a homogeneous system.

Currently, there are a number of modern microprocessors using different means to strive for parallelism and in-turn high performance. Some examples are presented.

1.2.4 Homogeneous Architectures

A homogeneous multi-processor consists of identical cores. This type of multi-processor is useful for applications which consist of very similar processing kernels. The MIT RAW [TKM+02] is a tile based homogeneous architecture where processing elements are arranged in a mesh, as shown in Figure 1.6(a). These tiles



(a) A RAW multi-processor. (b) Core groups in Core Fusion.

Figure 1.6: Homogeneous multi-processors.

contain simple RISC based processors, memory, and/or reconfigurable logic. The reconfigurable logic can be used for special instructions. The processing elements are connected to each other through a statically configured switched network.

Core Fusion [IKKM07] is a chip multi-processor consisting of 8 out-of-order dual issue cores, as shown in Figure 1.6(b). Being a reconfigurable architecture, it's processing elements and memory can be dynamically reconfigured. The processing cores work independently to run parallel code or up to four cores can combine to construct a large core having 8 issues for the sequential region. The operating system is responsible for fusing or splitting the cores. The dynamic fusion and splitting of the cores is dependent on the program code. During the sequential regions of the code, the cores are fused together to have one large core which benefits from the ILP. On the other hand if the application can be divided into threads then the cores are split and small cores run these threads to benefit from thread level parallelism.

Platform 2012 [IC10] (P2012) is an area and power efficient many-core computing fabric. The fabric consists of a control processor (ARM Cortex-A9) connected with multiple clusters of processing elements. The clusters are implemented with independent clock and power domains to enable efficient management of resources. Clusters are connected via a high performance fully asynchronous NoC, which provides scalable bandwidth and robust communication across different power and clock domains. Each cluster features up to 16 tightly-coupled processors sharing multi-banked level-1 instruction and data memories and a multi-channel advanced Direct Memory Access (DMA) engine and specialized hardware for synchronization and scheduling acceleration.

GPUs [Nvi11] are the leading exemplars of aggressively throughput-oriented processors. These architectures are massively parallel and operate on vectors of data. They are commonly known as Single Instruction Multiple Threads (SIMT) as all the threads in a warp execute the same instruction (Figure 1.7). They are built around an array of multi-processors, referred to as streaming multiprocessors (SMs). Figure 1.7 diagrams a representative Fermi-generation GPU

like the GF100 from Nvidia. Each multi-processor supports on the order of a thousand co-resident threads and is equipped with a large register file, giving each thread its own dedicated set of registers. The programs which have very high degree of parallelism benefit from execution on GPU like architectures.

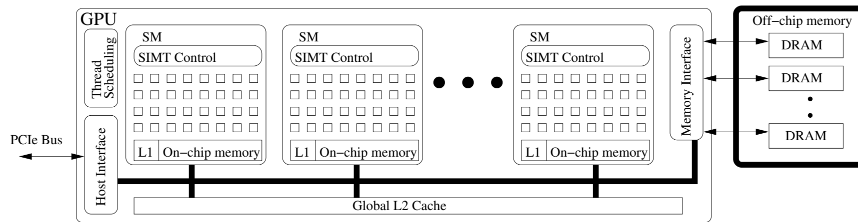


Figure 1.7: NVIDIA GPU consisting of an array of multi-threaded multi-processors.

Homogeneous processors are preferred for reasons like, ease of programming and simple operating systems. Same types of processing units are repeated in homogeneous architectures so the user has to learn single processor architecture for programming purposes of the multi-processor. Further, same binary file can execute on other cores resulting in lower instruction memory requirements. The homogeneous processors generally have lower performance per area/power figures when compared with heterogeneous multi-processors.

1.2.5 Heterogeneous Architectures

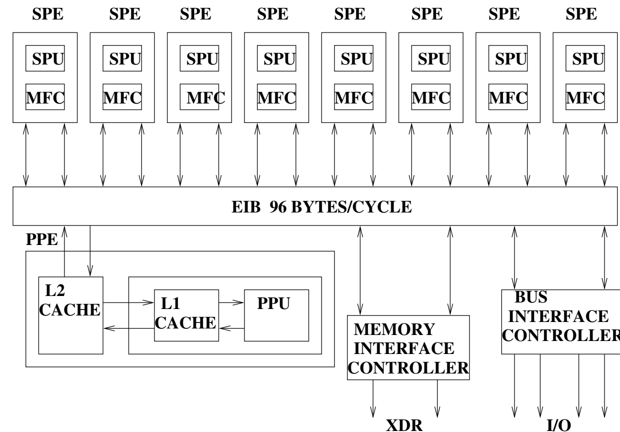


Figure 1.8: Cell System Architecture.

The heterogeneous architectures consist of different types of computational units. They are preferred over homogeneous multi-processors due to the fact that

they can combine different type of computational units to meet the different types of processing requirements in lesser area/power. Cell Broad-band Engine (Cell-BE) architecture [GHF+06] is based on heterogeneous chip multi-processor. It supports scalar and single-instruction, multiple data (SIMD) execution units to provide high-performance multi-threaded execution environment for all type of applications. Cell has one central processor, PPE (Power Processing element) to take care of control related operations and 8 SPEs (Synergistic processing engines) for more compute intensive operations.

OMAP5 [Ins11] is a multi-processor platform from Texas Instruments. It includes two ARM Cortex-A15 and two ARM Cortex-M4 processors, a DSP core, PowerVR graphics processing core and a number of audio video codecs. It also includes a dedicated power management unit. It can support 1080p video and can simultaneously support 4 video channels. It also has a set of development tools and can run a range of operating systems like Symbian, Windows mobile, Android and Linux.

We observe that these high performance platforms like CELL and GPUs use non-pre-emptive operating systems. The reason is simple; the overhead related with context switch in a pre-emptive system is so high that the benefit of pre-emption is almost lost due to large number of these processing elements. So we envision that future multi-processor platforms consisting of hundreds of processing elements may have non-pre-emptive operating systems. The brief survey of state of the art processors shows a definite trend of chip manufacturers towards designing multi-processor system on chip (MPSoC). Following is a summary that drives this trend.

- The power dissipation levels at high frequencies are difficult to handle. The solution is to have large number of relatively smaller cores which operate at a lower clock frequency to consume less power but divide the work between them. In this way, a platform can meet the processing demands of applications at lower power budget. Tile-based platforms provide additional benefit of defining a standard interface so that various tiles can be glued together.

- Multimedia systems have to support a large number of applications requiring different kinds of processing requirements. Single processor based solutions cannot handle these applications and their combinations. Heterogeneous architectures having different types of processing elements are ideal to handle diverse needs of multimedia applications. The homogeneous multi-processors have enormous benefit of being easily programmable and more user-friendly[1] (e.g. GPUs).

- The TTM of these products is decreasing very fast so tile-based architectures provide a good solution in the form of usability. The designer can build a platform by re-using tiles from previous designs so that the strict TTM deadlines can be met.

[1] the user has to learn only one ISA

- Non-pre-emptive multi-processors are preferred over pre-emptive multiprocessors for reasons of cost and efficiency (CELL, GPUs).
- Memories for these MPSoC are being organized in a hierarchy. Software controlled local memories and FIFOs are used for faster access (e.g. CELL, GPUs, RAW). DMA engines are being used to overlap communication with computation (CELL, P2012).
- Throughput-oriented architectures are gaining more market share due to reasons of high performance at low power. Many SoC chips are housing a GPU along with general purpose processors.

In the next section, we discuss the challenges in designing an MPSoC.

1.3 Predictable MPSoC Design

As described in the earlier sections, the performance requirements of modern applications have stimulated the transition from hardware consumer platforms based on a single processor to platforms that feature a multitude of processors, both homogeneous and heterogeneous. This transition however, has significantly complicated the design process. Operating systems now have to be distributed. Transactions on different processors now have to be synchronized in order to respect data dependencies and prevent congestion. Furthermore, there is the additional task of balancing the computational load over the various processors, and matching (sub-) task characteristics with processor capabilities. Data duplication yields the problem of keeping data consistent and coherent among memories and caches. For consumer-oriented platforms that execute various applications (e.g. modern televisions, set-top boxes, mobile phones) the largest impact is really on the verification effort, now already taking about 60% of the design effort. The main complicating factor for verification is the vast and increasing number of *use-cases*: for n applications, there are up-to 2^n separate use-cases. The verification of use-cases largely regards the timing behaviour, and is performed by extensive simulations. It is no coincidence that the largest design effort is concentrated in the very last design step (verification): even though there seems no technical or even moral justification, common design practice evolved to a culture where most of the 'misery' is shifted to the next design step (thrown over the wall) until the very last step (hey! We're out of walls). We strongly argue that from a technical viewpoint, earlier design steps are much better suited for analyzing timing behaviour, because (among other reasons) the higher abstraction level comprises less detail, and the timing behaviour (in terms of clock cycles) does not depend on the low-level design details typically added in later design steps.

Another result from shifting design problems to the very last is that no measures are taken in the early design steps to *constrain* the search space in any sensible way. Constraining the search space (e.g. use of a computational model

or design style) can be very helpful, since some designs allow better timing analysis than others. Obviously we are more interested in the former. Current design practice often does not take the goal of timing analysis into account, and indeed we observe that the resulting timing behaviour is essentially *unpredictable*, causing the need for extensive simulations. In this section, we will identify various sources of unpredictability, and provide alternative solutions in an early design phase that, integrated in a methodology comprising both hardware and design tools, enable the *analysis* of the timing behaviour, thereby eliminating the need for extensive simulations.

An MPSoC can be termed as *predictable* if it is possible to provide guarantees on its timing behaviour. The application domains such as automotive, avionics, mechatronics, and multimedia processing have strict constraints with respect to power consumption and size. Additionally, there are high requirements in terms of predictability. Not only are the correctness of the computations, the availability, and safety of the whole embedded systems of major concern, but also the timeliness of the results. Missing deadlines of events may cause a catastrophic or at least a highly undesirable system failure. If the MPSoC system under investigation has components which have non-deterministic behaviour then it is difficult to give guarantees on its performance. The reasons of unpredictability in an MPSoC are:

- *Processor Architecture:* The components that produce unpredictability in processors are caches, pipe-lines, out-of-order execution, branch predictors, dynamic memory allocation, Memory arbitration, DMA, and multitasking [HLTW03]. These components are designed to improve the average case performance of the processor. The result of these components is variation in execution time of the tasks executing on the processor. E.g. caches are used to bring more frequently used data/instruction in a faster memory (cache) so that their access time can be improved. If the data/instruction is not present in the cache then it is fetched from the main memory. The penalty to bring data from the main memory is normally in hundreds of cycles and hence the fact that whether the next instruction will hit or miss the cache creates unpredictable timing behaviour.

  Pipe-lining is used to fetch new instructions while the previously fetched instructions are being executed. Pipe-line stalls are sources of unpredictability. Branch predictors keep a history of the previously taken branches and provide the address of the next branch target if it was previously taken. If there is a misprediction then the pipe-line has to be flushed and the whole work is performed again. This way unpredictability is introduced due to branch predictors. Similarly, if the resource sharing strategy is implemented using probabilistic techniques then it is not possible to provide guarantees on the performance.

- *Operating System/Resource Manager:* Operating system/Resource Manager

(In case of MPSoC, the operating system is distributed and it is often called Resource Manager) is another source of unpredictability in MPSoC systems. If the resource arbitration strategy is based on probabilistic arbitration methods then predictability cannot be guaranteed. Interrupts are another source of unpredictability. From multi-tasking point of view to implement pre-emption, interrupts are used. All of these interrupts occur at non-deterministic times and sometimes even the number of interrupts cannot be bounded. Pre-emption based operating systems make the predictability analysis very difficult.

- *Inter-processor Communication:* One major source of decreased timing predictability is the close interaction between computation and communication in processors of MPSoC. In particular, the response time of a process now depends on the message delay across network, that is, the time between sending and receiving a message. This interference between different communicating tasks is caused by the network performance under varying traffic conditions.

As a consequence, there are two orthogonal but related ways to improve the timing predictability of embedded systems. First method to improve predictability is to remove the parts of computer architecture that are sources of unpredictability. The second method is to model these components so that the analysis can be more accurate.

The design of a predictable system requires that the timing behaviour of the application and its mapping to the platform can be analyzed. This can be done by modelling the application and mapping decisions in a Model-of-Computation (MoC) that allows timing analysis. A model of computation is used to describe the behaviour of the application. The MoC should be able to express the parallelism in the application. Additionally, the MoC should be able to model the synchronization and communication between the tasks. Furthermore, the MoC must capture the timing behaviour of the tasks and allow analysis of the timing behaviour of the application. This makes it possible to verify whether the timing constraints imposed on the application are satisfied. Finally, the MoC should allow a natural description of the application in the model. Synchronous Dataflow graphs [LM87] possess most of the properties mentioned above. We also use this MoC in our thesis. For a detailed comparison among state-of-art MoCs, please refer to [Stu07].

1.4 Importance of Application Model and Specification

Most of multimedia systems deal with the processing of audio and video streams. This processing is done by applications that perform functions like object recognition, object detection, image, and audio enhancement on the streams. Typically,

these streams are compressed before they are transmitted from the place where they are recorded (sender) to the place where they are played-back (receiver). Applications that compress and decompress audio and video streams are therefore among the most dominant streaming multimedia applications [Wol05].

The compression of an audio or video stream is performed by an encoder. This encoder tries to pack the relevant data in the stream into as few bits as possible. The amount of bits that need to be transmitted per second between the sender and receiver is called the *bit-rate*. To reduce the bit-rate, audio and video encoders usually use a lossy encoding scheme. In such an encoding scheme, the encoder removes those details from the stream that have the smallest impact on the perceived quality of the stream by the user. For example, human eye is insensitive to high frequencies so these frequencies can be filtered out without much effect on the quality. Typically, encoders allow a trade-off between the perceived quality and the bitrates of a stream. To further reduce the bitrates, lossy encoding schemes are followed by loss less compression algorithms like Huffman coding [Huf52]. JPEG encoder [dK02] is used to encode still pictures. We use JPEG encoder as a running example in this chapter to illustrate application modeling. Figure 1.9 shows block diagram of a JPEG encoder. The first function gets macro-blocks from the stream and performs colour conversion to extract R, G and B components. The converted macro-blocks are sent to Discrete Cosine Transform (DCT) block where the stream is converted into frequency domain and high frequency components are filtered out. The stream is then fed to Variable Length Coder block (VLC) and the resulting JPEG stream is sent to the receiver or rendering device.

In order to ensure that this high performance can be met by the platform, the designer has to be able to model the application requirements. In the absence of a good model, it is very difficult to know in advance whether the application performance can be met at all times, and extensive simulation and testing is needed. Even now, companies report a large effort being spent on verifying the timing requirements of the applications.

To achieve high performance, maximum achievable parallelism must be extracted from the application. Parallelism can be exploited at different levels i.e. instruction level, data level and task level as described earlier in the chapter. For example, super-scalar and VLIW processors exploit instruction level parallelism. An MPSoC platform consisting of VLIW/super-scalar processors can exploit both task level and instruction level parallelism. An application model should be such that the maximum available parallelism is visible to the design tools. When multimedia applications are mapped onto multi-processor platforms, all three levels of parallelisms can be exploited. The individual processors exploit instruction level and data level parallelism and they exhibit task level parallelism by assigning tasks to the processors.

Parallelizing an application to make it suitable for execution on a multiprocessor platform is an active research area. Some notable works include the SUIF [HAA$^+$96] and OPENMP [CDK$^+$01]. The user has to provide all kinds of

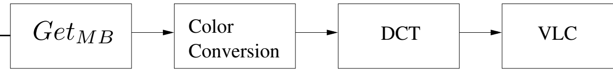


Figure 1.9: Block Diagram of JPEG encoder

pragmas to get any performance from these tools so a lot of work needs to be done in this area. The parallelization of applications is out of scope of this thesis and we assume that the applications have already been divided into tasks.

When multiple applications are to be executed onto the platform, the combinations of the executing applications determine their combined resource requirement. With multiple applications executing on multi-processors, the potential number of use-cases increases rapidly, and so does the cost of verification. We model a use-case by a Boolean vector, as follows:

Definition 1 (USE-CASE)*: Given a set of n applications $A_0, A_1, \ldots A_{n-1}$, a use-case U is defined as a vector of n elements $(x_0, x_1, \ldots x_{n-1})$ where $x_i \in \{0, 1\} \forall\ i = 0, 1, \ldots n-1$, such that $x_i = 1$ implies application A_i is active.*

To summarize, following are our requirements from an application model that allows mapping and analysis on a multi-processor platform:

- *Evaluate computational requirements:* The computation requirements of applications are to be known precisely so that the platform can be dimensioned appropriately. The size of compute resources affects the cost of the platform. The model of application should reflect its compute requirements accurately.

- *Memory requirements:* The cost of memories is still very high despite cheap transistors available on the die. The application model should specify the memory requirement of the applications. The throughput of streaming applications depends on the buffering between the communicating actors. The application model should be capable to capture the buffer requirements of the applications such that the analysis tools can provide the memory throughput trade-off points. The designer can choose the buffering that meets the constraints of the applications.

- *Communication requirements:* The dataflow model should be able to model the communication delay between the actors mapped onto different processing elements. Other communications components, like communication assists, routers, and switches should also be modelled.

- *Scheduling and Performance Analysis:* When multiple applications share the platform, a schedule is required that specifies the assignment, order,

and execution of actors onto processors. The application model should be able to model the scheduling decisions so that the analysis tools can predict the performance of applications when mapped onto the MPSoC platform.

- *Synthesize the System:* Once the performance of system is considered satisfactory, the system has to be synthesized such that the properties analyzed are still valid.

Dataflow models of computation fit well with the above requirements. They provide a model for describing signal processing systems where infinite streams of data are incrementally transformed by processes executing in sequence or parallel. The next section provides an introduction to Synchronous dataflow model.

1.5 Introduction to SDF Graphs

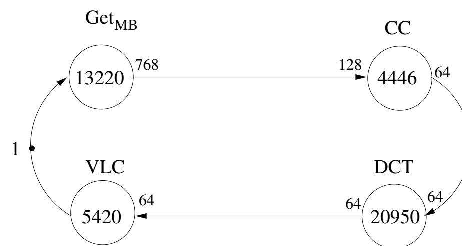


Figure 1.10: Example of an SDF Graph (JPEG Encoder)

Synchronous Dataflow Graphs (SDFGs) are often used for modelling modern DSP applications [SB00] and for designing concurrent multimedia applications implemented on multi-processor systems-on-chip. Both pipe-lined streaming and cyclic dependencies between tasks can be easily modelled in SDFGs. Tasks are modelled by the vertices of an SDFG, which are called *actors*. The communication between actors is represented by *edges* through which they are connected to other actors. Edges represent *channels* for communication in a real system.

The time that the actor takes to execute on a processor is indicated by the number inside the actor. It should be noted that the time an actor takes to execute may vary with the processor. For sake of simplicity, we shall omit the detail as to which processor it is mapped on and just define the time (or clock cycles) needed on a typical RISC processor [PD80], unless otherwise mentioned. This is also sometimes referred to as *Timed SDF* in literature [Stu07]. Further, when we refer to the time needed to execute a particular actor, we refer to the worst-case execution time (WCET). The average execution time may be lower than the WCET.

Figure 1.10 shows an example of an SDF graph. There are four actors in this graph. As in a typical dataflow graph, a directed edge represents the dependency between actors. Actors need some input data (or control information) before they can start, and usually also produce some output data; such information is referred to as *tokens*. The number of tokens produced or consumed in one execution of an actor is called *rate*. In the example, Get_{MB} has an input rate of 1 and output rate of 768 pixels (1 macro-block). Further, its execution time is 13220 clock cycles. Actor execution is also called *firing*. An actor is called *ready* when it has sufficient input tokens on all its input edges and sufficient buffer space on all its output channels; an actor can only fire when it is ready. This property directly translates into predictable application model. When a processor starts to execute a ready actor, it will successfully complete the execution as input data and the space for the output data is available, so the actor will surely finish its execution. Compare it with the case if we drop any one of the condition e.g. if we assume that we start the execution of an actor as soon as input tokens are available then the processor may block when it goes to store the output data as the output buffer is not empty. This will result in processor stalling and delaying the execution of other actors scheduled onto the same processor and hence resulting in unpredictable application behaviour. This is also one of the reasons we choose SDFGs as our model of computation in this thesis.

The edges may also contain *initial tokens*, indicated by bullets on the edges, as seen on the edge from actor VLC to Get_{MB} in Figure 1.10. In the above example, only Get_{MB} can start execution from the initial state, since the required number of tokens is present on its only incoming edge. Once Get_{MB} has finished execution, it will produce 768 tokens on the edge to colour conversion CC. CC can proceed, as it has enough tokens, and upon completion produce 64 tokens on the edge to DCT. The actor DCT then produces 64 tokens on its edge to VLC. The actor VLC is the last actor to be executed during an iteration of the graph.

A number of properties of an application can be analyzed from its SDF model. We can calculate the maximum achievable performance of an application. We can identify whether the application or a particular schedule will result in a deadlock. We can also analyze other performance properties, e.g. latency of an application, buffer requirements. Below we give some properties of SDF graphs that allow modelling of hardware constraints that are relevant to this thesis.

1.5.1 Modelling Auto-concurrency

SDF models can show the achievable task level parallelism in an application. Concurrency is a property of systems in which several computations are executing simultaneously. The example in Figure 1.10 shows this property. According to the model, since CC requires only 128 tokens on the edge from Get_{MB} to fire, as soon as Get_{MB} has finished executing and produced 768 tokens, six executions of CC can start simultaneously. However, this is only possible if CC is mapped and allowed to execute on multiple processors simultaneously. In a typical system,

CC will be mapped on a single processor. Once the processor starts executing, it will not be available to start the second execution of CC until it has at least finished the first execution of CC. If there are other actors mapped on it, the second execution of CC may even be delayed further.

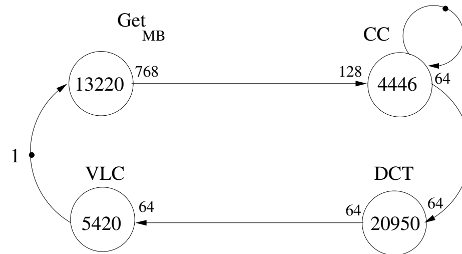


Figure 1.11: SDF Graph after modeling auto-concurrency of 1 for the actor CC

Fortunately, there is a way to model this particular resource conflict in SDF. Figure 1.11 shows the same example, now updated with the constraint that only one execution of CC can be active at any one point in time. In this figure, a *self-edge* has been added to the actor CC with one initial token. (In a self-edge, the source and destination actor is the same.) This initial token is consumed in the first firing of CC and produced after CC has finished the first execution. Interestingly enough, by varying the number of initial tokens on this self-edge, we can regulate the number of simultaneous executions of a particular actor. This property is called **auto-concurrency**. In Figure 1.11, the auto-concurrency of CC is 1.

Definition 2 (Auto-concurrency)*: The* **auto-concurrency** *of an actor is defined as the maximum number of simultaneous executions of that actor.*

1.5.2 Modelling Buffer Sizes

SDF graphs can model the buffer space between the actors. Buffer-sizes may be modelled as a back-edge with initial tokens. In such cases, the number of tokens on that edge indicates the buffer-size available. When an actor writes data on a channel, the available size reduces; when the receiving actor consumes this data, the available buffer increases, modelled by an increase in the number of tokens.

Figure 1.12 shows such an example, where the buffer size of the channel from CC to DCT is shown as 64. Before CC can be executed, it has to check if enough buffer space is available. This is modelled by requiring tokens from the back-edge to be consumed. Since it produces 64 tokens per firing, 64 tokens

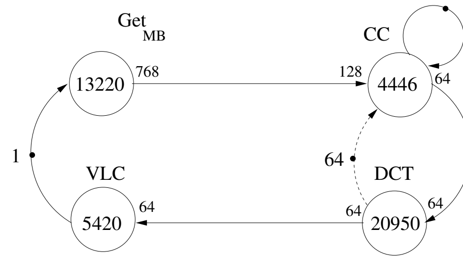


Figure 1.12: SDF Graph after modelling buffer-size of 64 on the edge from actor DCT **to** CC

from the back-edge are consumed, indicating reservation of 64 buffer space on the output edge. On the consumption side, when DCT is executed, it frees 64 buffer spaces, indicated by a release of 64 tokens on the back-edge. In the model, the output buffer space is claimed at the start of execution, and the input token space is released only at the end of firing. This ensures atomic execution of the actor.

In the next section, the hardware architectural template used in this thesis is presented to explain how the predictability of the system has been improved.

1.6 Predictable MPSoC Template

To address the unpredictability issues in MPSoC platforms, an MPSoC template is presented in this thesis. The multi-processor template used in this thesis is shown in Figure 1.13. This template is based on the tile-based multi-processor platform described in [CSG99]. It consists of multiple tiles connected with each other by an interconnection network. Each tile consists of a processing element, local memory and a communication assist (CA) [SSK+10]. The processing elements used in the template are either simple RISC processors or application specific accelerators. The processing element accesses the memory through the CA. The advantage for using a CA is two-fold. Firstly it relieves the processor to push and pop data from the network. Secondly it implements the SDF semantics, i.e. checking input tokens and output space before the execution of an actor. In Chapter 2, we explain the architecture of CA in more detail.

The inter-connect network between the tiles in the platform template should offer unidirectional point-to-point connections between pairs of NIs. In a predictable platform, these connections must provide guaranteed bandwidth and a tightly bounded propagation delay per connection. They must provide a guaranteed throughput. The connections must also preserve the ordering of the communicated data. A number of Network-on-Chip (NoC) architectures can provide all these properties. The NoC consists of a set of routers which are connected to each other in an arbitrary topology. Each tile is connected through its NI with a router (R) in the NoC. The connections between routers and between routers

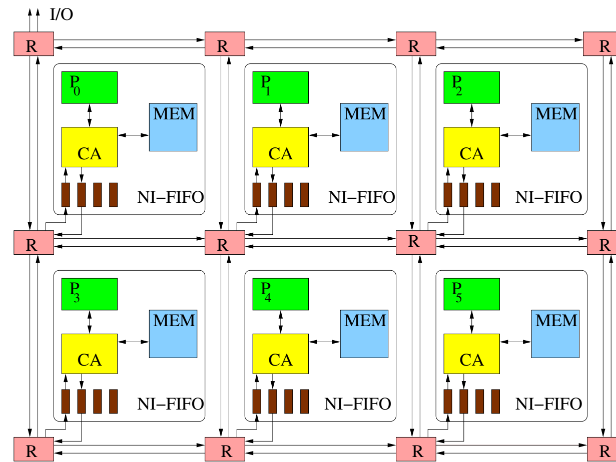


Figure 1.13: Multi-processor template with a communication assist (CA) a network interface (NI) and a processor (P)

and NIs are called links. Examples of NoCs providing the required properties are Æthereal [RDG+04] and Nostrum [MNTJ04].

Task execution on the processors of this MPSoC template is non-pre-emptive. We observe that for high-performance multimedia systems (like CELL processing engine and graphics processors), non-pre-emptive systems are preferred over pre-emptive ones for a number of reasons [JSM91]. Implementation of non-pre-emptive systems does not require interrupts so it increases the predictability of the system. In many practical systems, properties of device hardware and software either make the pre-emption impossible or prohibitively expensive due to extra hardware or (potential) execution time needed. Further, non-pre-emptive scheduling algorithms are easier to implement than their pre-emptive counterparts and have dramatically lower overhead at run-time [JSM91]. Further, even in multi-processor systems with pre-emptive processors, some processors/co-processors is usually non-pre-emptive; for such processors non-pre-emptive analysis is needed. In this thesis, we have used non-pre-emptive scheduling algorithm for cost and predictability reasons.

In the next section, key contributions have been presented which enhance the predictability of the MPSoC template.

1.7 Key Contributions of the Thesis

In this section, we summarize the trends in applications and architectures presented in the previous sections and present the solutions given in this thesis. The development of modern multimedia systems is driven by the growing demand for

high-end value-added functionality. High-bandwidth digital communication, gaming, augmented reality, high-quality image and video playback and encoding are just a few examples of applications that are often decisive for the market success of a silicon platform. These applications are extremely demanding from a computational viewpoint: multi-GOPS performance requirements are not uncommon. Fortunately, they usually exhibit high levels of fine- and coarse-grain parallelism (data- and task-level parallelism, respectively). In this context, computing engines have traditionally been implemented as hard-wired functional units, but this scenario is changing under the pressure for increased flexibility.

Today, we see a trend towards multi-processor fabrics, with a throughput-oriented memory hierarchy featuring software-controlled local memories, FIFOs, and specialized DMA engines. The design of these many-core fabrics is a complex problem as the fabric has to meet the throughput constraints of multiple applications. The multi-processor should have predictable behaviour so that the applications meet their constraints in all use-cases. The system should also be able to handle dynamic situations like admission of new applications, variation in application constraints etc. The designer faces the challenge to design such systems at low cost and short time. The solutions to some of these challenges are provided in this thesis. Following is a summary of these contributions.

1.7.1 Communication Assist

The processors in an embedded system communicate with each other so that the application can be divided in tasks and the processors can cooperate to handle this large number of applications. The communication fabric should have a predictable behaviour so that application performance can be guaranteed. Guarantees on the performance can be provided by decoupling communication and computation. A number of CAs/DMAs [MBB+05, SIAM+04] have been proposed in the literature but they suffer from the high memory requirement and lack of analysis support. The CA is an advanced distributed DMA controller. Distributed means in this context that the CAs at both ends of the connection are working together to execute a block transfer, using a communication protocol on top of a network protocol [MBC07]. These communication assist architectures require separate space for data to be communicated. The communication assist presented in this thesis has predictable timing behaviour and requires less memory as compared to the communication assists mentioned in the literature [MBB+05, SIAM+04]. The communication assist uses the data memory as communication memory such that the overall memory requirement is lower than the reference architectures. Further, it allows out-of-order access, re-reading, and skipping within the data/communication memory.

We present detailed architecture of the communication assist. It also performs memory management functions so that the programmer is free from the overhead. The communication is performed using circular buffers so that memory can be used efficiently. We also present its model so that analysis tools can be used to

predict the performance of the overall system. Note that the network should also provide guarantees for the overall system to be predictable.

1.7.2 Design Algorithm

The number of possible use-cases executing on a multimedia platform is exponential in the number of applications. A design strategy should be able to generate an MPSoC platform capable of guaranteeing the performance of applications in all these use-cases with minimum possible hardware resources. In this thesis, we present an MPSoC design strategy that generates MPSoC platforms capable of guaranteeing the performance of applications in all use-cases. The processing elements in our platform communicate with the help of CAs. There are a number of platform generation strategies [SKC00, OH02, SBGC07] for multiple applications but only [OH02] can perform optimization across use-cases. The technique assumes that applications are modelled as acyclic task graphs. This model-of-computation (MoC) is not very suitable for modelling streaming, pipelined applications. The synchronous dataflow MoC [LM87] allows modelling of pipe-lining. A design approach that uses this MoC can potentially save resources as compared to a design approach based on acyclic task graphs. Our design approach uses SDFGs and can meet the throughput constraints with fewer resources. Additionally, our design approach also tries to minimize the communication memory requirement of the design.

The main contribution of this work is the fact that we perform hardware optimization across the use-cases. For example, a platform has to execute 3 applications named A, B and C. Assume that these applications are active in three use-cases AB, AC and BC, and each application requires the same resources. Now, if dedicated resources are allocated to the applications then the total resources needed for these applications will be the sum of the resources needed by each application, individually. In this example, the total required resources will be 3 units. On the other hand, if we know the use-cases which will be executing on the platform then the resources can be shared across the use-cases. In this case, the required resources will be two units only.

The platforms proposed by our algorithm are synthesized on Xilinx FPGAs using our tool CA-MPSoC [CM09]. The tool also generates the static order schedules and software needed to execute the applications onto MPSoC platform. One of the drawbacks of our design flow is that it does not support task migration. This means that the mapping of actors onto the processors remains the same across all use-cases. This may result in a slightly less efficient use of resources by our design flow but the platforms synthesized are simple and easy to program.

1.7.3 Distributed Resource Management

Resource management on MPSoCs is equivalent to operating systems in general purpose processors. The jobs of a resource manager (RM) include receiving re-

quests from the user for application execution, starting a new application (admission control), stopping an executing application, and changing the performance constraints of the application according to user's request. Resource management for multimedia systems is different from general purpose computers as the application domain is generally well known. So the resource management functions can be tailored according to the requirements of the domain. For example, if the MPSoC does not contain any caches then the cache coherency protocols are not needed for the resource manager. Further, most decisions can be taken during design time to make the run-time intervention cost as low as possible. This also makes the resource manger simpler to implement.

Normally centralized resource managers [BPBL06, ABC$^+$09, KMT$^+$08] are employed that monitor the performance of applications and take appropriate corrective measures. These centralized resource managers are not very scalable with increasing number of applications and processing elements. In this thesis, the resource managers use off-line information about the applications and use it at run-time to control the applications. The time consuming application specific computations are done at design time for each application and independent from other applications. The off-line computation includes partitioning of the application into tasks, modelling them into a model of computation, determination of worst-case execution times, determination of maximal throughput, etc. All this analysis is time consuming and has to be carried out at design time. At run-time this information is used to estimate the resource requirements of each application and the resource manager decides either to accept the application or reject it so that the user can re-try the application at a lower level of Quality-of-Service (QoS). An admission controller is part of resource manager and provides an interface to the applications to meet their service demands.

Along with the admission controller, the RMs presented in this thesis can be distinguished on the basis of their budget enforcement protocols. The first type named as *Credit-Based* RM is useful for applications which require very strict timing constraints. The credit-based RM is a type of budget-based scheduler where budgets are assigned to tasks in a large replenishment interval. The tasks are executed according to its credits and reloaded after the finishing of replenishment interval. The second type of RM is called *Rate-Based RM*. The tasks in a rate-based RM maintain a ratio which is specified by the central admission controller. In rate-based RM, the tasks are never disabled so it is possible that they may achieve more than desired throughput.

1.7.4 Distributed Simulation on FPGA

MPSoC platforms for real-time applications are designed for worst-case task execution time estimates. Kumar has shown that the average-case performance is often two fold better than worst-case estimated performance (with full virtualization) [Kum09]. Therefore, knowing the average-case performance is also important for the system designer.

Software simulation is often employed to estimate the difference between worst-case and average-case behaviour. Simulation of multi-processors in software becomes slow as the numbers of processing elements are increased. The simulation can be accelerated with the help of hardware support. In this work, we have developed a parallel simulation framework (MAMPSIM) where we generate parallel simulators on FPGA platforms. The user can specify the topology of the platform and the applications which he/she wants to simulate and our strategy generates a simulator which can be synthesized with the help of commercial FPGA synthesis tools. After synthesis, the simulator can be executed and performance results can be analyzed by the user so that he/she can generate the suitable MPSoC platform. Again, the simulator can simulate multiple use-cases.

The simulation framework can simulate scheduling policies like First Come First Serve (FCFS) and Round Robin with Skipping (RRWS) on FPGAs. These scheduling polices are difficult to analyze so simulation is an option to observe their behaviour. In our approach, we use Parallel Discrete Event Simulation [Fuj89] (PDES) for simulating multiple applications – each consisting of parallel tasks – executing on multiple processors. Most PDES approaches fall under one of the two categories – *conservative* and *optimistic*. We propose and use a *smart conservative* approach that is intelligent to figure out when the sequential program execution can be set aside for improved efficiency. We have developed a mechanism which on every simulation step checks whether continuing with the simulation on incomplete information can result in a causality error. If sequential execution is imperative then we apply conservative PDES, otherwise we turn to *smart conservative* PDES. Contrary to the optimistic PDES where the simulation is continued on incomplete information, the smart conservative verifies that the next simulation step will not create any causality error before taking that step.

The contributions of this thesis have been put together in the form a complete design flow. The next section provides details about the design flow.

1.8 Design Flow

Figure 1.14 shows the design flow that is presented in this thesis. Specifications of applications are provided by the designer in the form of SDFGs. The task level source codes of the actors and the use-case information in also provided to our tool flow. The flow has two parts. One part generates platforms using analyzable components and scheduling techniques. The other part of the flow is dedicated to run-time scheduling techniques and simulation is used to find the difference between worst-case and average-case behaviour of the synthesized platform. The part of our design flow which deals with analyzable components, finds the minimal platform that can meet the throughput constraints of all applications in all use-cases. The mapping of the actors onto processor is also an output of our design algorithm as shown in Figure 1.14. The proposed platform is then fed to our CA-based platform generation tool which generates the platform and the executable

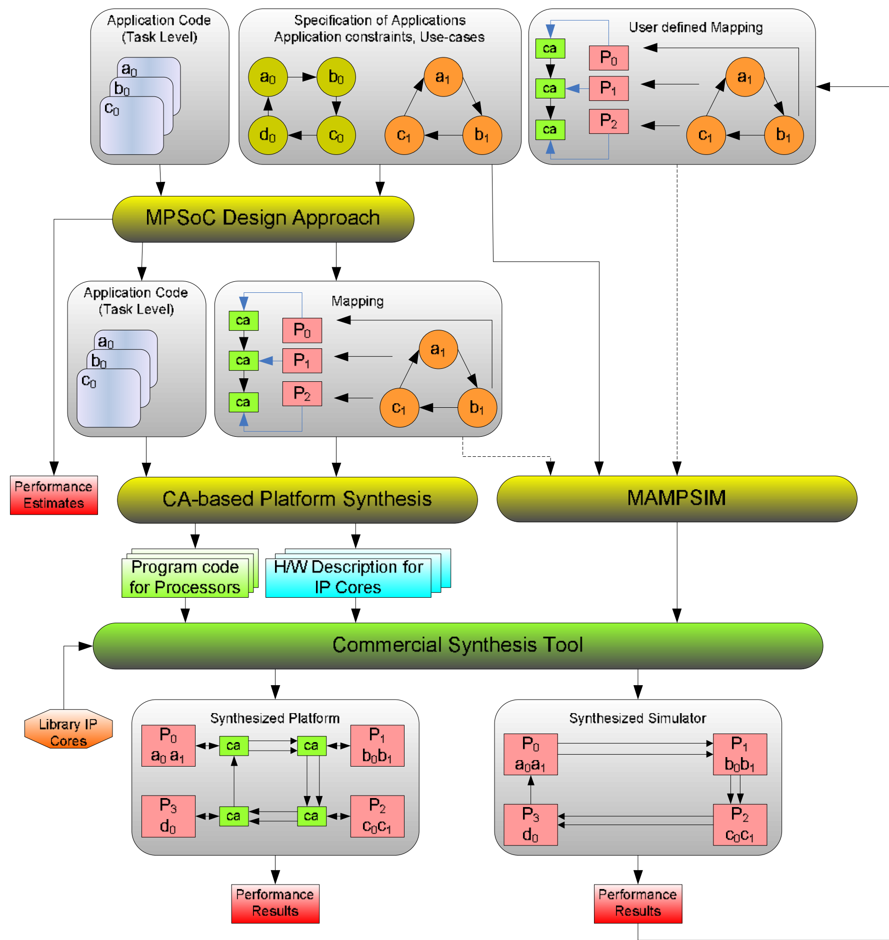


Figure 1.14: Multi-processor system design flow

code for each processor according to the mapping of actors onto processors. The generated platform can be synthesized using Xilinx tools and can be executed on Xilinx development board.

The second part of our design flow simulates the execution of applications on to the synthesized platform and provides average-case performance results. It can also be used for scheduling algorithms which are difficult to analyze. For this part of flow, the platform and the mapping of applications onto processors has to be provided by the user. Our MAMPSIM tool [CM09] generates the simulation platform of the design and performance results can be generated from the hardware model executing on the Xilinx board. The user can iterate over the flow to get the satisfactory results.

1.9 Thesis Overview

The thesis is organized as follows. Chapter 2 presents the architecture and model of the communication assist. The communication assist provides guarantees on the latency of transfer of data between the processors. We also present its model so that it can be used in an SDFG to predict the performance of applications when mapped onto our CA-based platforms. Chapter 3 presents our platform generation methodology. The algorithm tries to generate a minimal platform that can satisfy the performance constraints of multiple applications under the given use-cases. Chapter 4 presents our platform synthesis tool flow. The flow synthesizes the hardware and software of the platform. Chapter 5 presents two distributed resource managers for MPSoC platforms. The resource managers have been designed to address the scalability issues of budget enforcement techniques. Chapter 6 explains the second part of our tool flow which relies on simulation to give performance results of platforms under dynamic scheduling techniques. Simulation also gives the average-case performance of the applications and can also be used for Design Space Exploration purposes. Finally, Chapter 7 concludes this thesis and presents future directions in this field.

CHAPTER 2

Communication Assist Architecture

A predictable multi-processor platform should provide a conservatively-estimated lower bound on the minimum achievable throughput and upper bound on end-to-end latency for each application executing on it. This requires that the computation and communication between the processors is predictable. Predictable memory access latencies are also a requirement for predictable MPSoC platforms. If the memory is shared between the tasks, the arbiters should provide guaranteed resource budgets so that the applications can meet their constraints. Predictability in communication means that the communication infrastructure provides guarantees.

To demonstrate the lack of predictability, consider a simplified (without support for virtualization, etc.) multi-processor platform executing the program fragments as depicted in Figure 2.1. Suppose we are interested in the timing behavior of application 'appA' running on processor P_0. Assume that 'appA' runs in 100 clock cycles without interruptions or processor stalls. What can happen in the worst case, is that each time 'appA' performs an action on the bus (receive or send), it is occupied by 'appB' running on processor P_2. In the simple platform of Figure 2.1 this would result in a processor stall (on P_0) until P_2 releases the bus. Now suppose that 'appB' (running on P_2) is moving big chunks of data taking 1000 clock cycles for each 'send' transaction. That means that each 'receive' action in 'appA' yields in a 1000 clock cycle stall, resulting in a worst-case execution time (for 'appA') of 100+3×1000=3100 clock cycles! The execution time of 'appA' can therefore vary between 100 and 3100 clock cycles depending upon the amount of traffic congestion. The variation in execution times makes it very

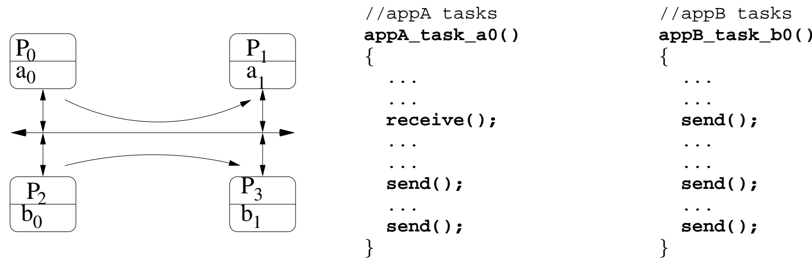


Figure 2.1: Multi-processor platform and source code snippets for applications A and B.

difficult to guarantee the performance.

The guarantees on the performance can be provided by decoupling computation and communication. Decoupling means that the communication of data has no effect on computation. The processor needs to perform computation while a specialized piece of hardware is responsible to perform the communication. As described in Chapter 1, DMA controllers and CAs are examples of such specialized hardware modules. There are a number of CA architectures [MBB+05, SIAM+04], but they suffer from high memory requirements and lack of analyzability. In this chapter, architecture of a CA is presented. It is shown that a proper programming model combined with hardware support can guarantee performance of multiple applications executing on an MPSoC. The CA ensures predictability by implementing the semantics of SDFGs, namely checking the available output and input buffer space before the actual firing of the actor. The CA also implements the memory management functions for the processor. The SDF model of CA is presented so that the bounds on the performance can be calculated using the standard SDF analysis techniques. APIs are developed so that the CA can be used easily.

This chapter is organized as follows. Section 2.1 presents existing CA architectures and their problems. Section 2.2 introduces our CA. Section 2.3 describes architectural details while Section 2.4 presents its SDF model. Section 2.5 discusses the hardware implementation of the CA. Section 2.6 presents results of experiments done to evaluate our CA. Section 2.7 reviews the related work for CA architectures. Section 2.8 concludes this chapter and gives directions for future work.

2.1 Existing CA Architectures

A predictable MPSoC platform should have a predictable temporal behaviour so that we can reason about the performance of applications when they are mapped onto the platform at design time. This results in requirements for the predictability of memory access latencies. If the memory is shared between tasks, the arbiters should provide guaranteed resource budgets so that the applications can

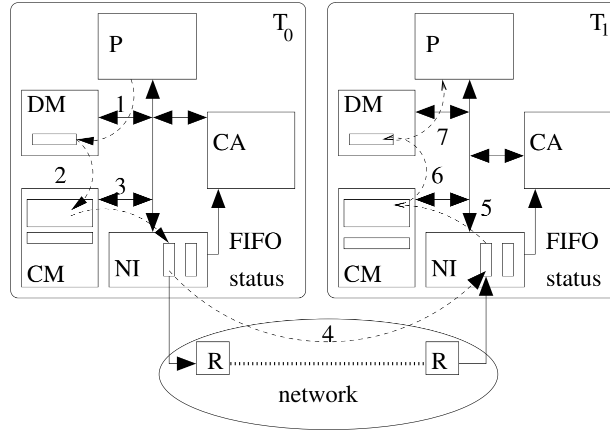


Figure 2.2: CA-based platform from [MBB+05]

meet their constraints. An instance of the CA-based platform from [MBB+05] consisting of two tiles, is shown in Figure 2.2. Each tile consists of a processor (P), a data memory (DM), a communication memory (CM), a CA, and a network interface (NI). The processor P in tile T_0 is executing a producer task and the processor on tile T_1 is running a consumer task. The producer task sends data to the consumer task. There is a dedicated network connection between these tasks starting at a FIFO in the NI of tile T_0 and ending at a FIFO in the NI of tile T_1. The producing task performs the processing on the data in its data memory (step 1 in Figure 2.2). The processor then copies this data into a logical FIFO in its communication memory (step 2). The CA transfers this data to the NI FIFO (step 3) and then it is transported over the dedicated point-to-point connection to the corresponding NI FIFO at the receiving tile (step 4). As soon as the data arrives in the NI FIFO of the tile T_1, it is copied by the CA into the logical FIFO of this tile (step 5). Processor P in tile T_1 reads this data into its data memory once it detects that the data is available (step 6). Next, it processes this data (step 7). One apparent disadvantage of this CA is the duplication of data in the communication and data memory. This data duplication results in a high memory requirement. Furthermore, the processors have to transfer the data between communication memory and data memory which costs precious processor time. This in turn results in a large latency when communicating data between tiles. To rectify these problems we propose a novel CA. Our design uses one memory region for both computation and communication data. This results in up-to 50% decrease in memory requirement. The communication latency is also decreased at the same time.

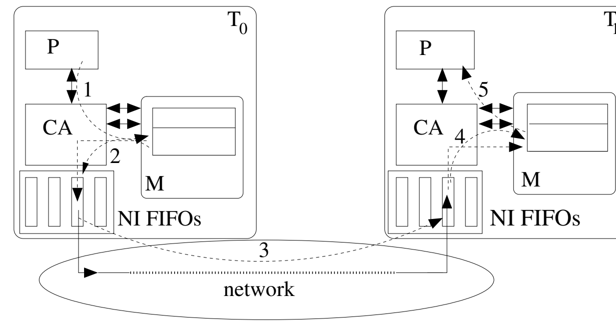


Figure 2.3: Proposed CA-based platform

2.2 Novel Communication Assist Architecture

Figure 2.3 shows an instance of our CA-based platform. We use the C-HEAP [NKG+02] protocol of synchronization, as it consists of simple and easy to program primitives. The producer task running on processor P in tile T_0 asks the CA for *write space* as shown in Figure 2.3. The CA returns the virtual address of the output buffer in the memory. The producer task processes the data and then releases the space indicating the CA to pump this data into the network (step 1 in Figure 2.3). The CA in tile T_0 copies this data in its NI FIFO (step 2). The data is transported through the network (step 3). The CA at the consumer receives the data at its input buffer. Furthermore, it sends the pointer of this buffer to the consuming task (step 4). This task processes the data and releases the space so that the CA can use this space for future data receptions (step 5). Note that our CA does not use a separate communication memory. Therefore, the copying steps 2 and 6 from Figure 2.2 are not required in our CA, resulting in a lower communication latency and memory requirement.

Our communication assist sits between processor and its data memory. The data memory is virtually divided into two parts. One part is used as data memories and the other part is used to implement the circular buffers. These circular buffers are used to store the data that is to be sent/received to the neighbouring tiles. Figure 2.3 shows the global view of our CA and following are its basic functions:

1. It accepts data transfer requests from the attached processor and splits them into local and remote memory requests.

2. Local memory requests are simply bypassed to the data memory.

3. Remote memory requests are handled through a round robin arbiter. Every two cycles, a 32-bit word is transferred from the buffer in the memory to a NI FIFO channel or vice versa.

4. The communication buffers are implemented in the memory as circular buffers. The number of NI FIFO channels can be greater than or equal to the number of buffers in the data memory. Our CA is programmable, so the same buffer in the memory can be used as input and output depending on the NI FIFO to which it is connected.

Our CA acts as an interface that provides a link between the NoC and the sub systems (processor and memory). It also acts as a memory management unit that helps the processor keep track of its data. As a result, it decouples communication from computation and relieves the processor from data transfer functions.

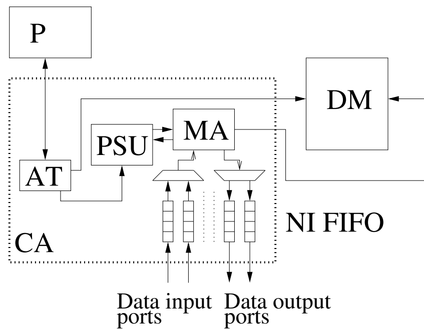


Figure 2.4: CA architecture.

Content	Offset
Base address of the buffer	0x00
Size of the buffer	0x02
NI FIFO ID, direction	0x04
Write Start, W_S	0x06
Write End, W_E	0x08
Read Start, R_S	0x0A
Read End, R_E	0x0C

Figure 2.5: Context registers of a data buffer.

2.3 CA Architecture

Figure 2.4 shows the block diagram of our CA. The CA lies in between the processor and a dual port data memory (DM). One port of the data memory is directly connected to the CA whereas the other port is connected to the data bus of the processor through an Address Translation Unit "AT" inside the CA. The CA is connected to the network through input/output ports. Each data port has a FIFO buffer (NI FIFO) that connects the Memory Arbiter (MA) to the network. The NI FIFOs can be driven by two clocks: 1) the network clock and 2) subsystem clock. Separate clock domains allow the integration of subsystems with different clock frequencies. Following are the main components of our CA.

The **Address Translation Unit (AT)** is connected to the processor of a subsystem. The AT monitors the address bus of the processor and distinguishes between the local memory accesses and buffer memory accesses. It passes the local memory accesses to the DM and translates the virtual address of a buffer into physical memory address. The detail about virtual memory addressing is presented in the next subsection.

The **Pointer Store Unit (PSU)** contains a set of registers (called buffer context) describing the status of each buffer. A buffer context consists of 7 registers as shown in Figure 2.5. The PSU selects one of the buffer contexts as indicated by the MA, sends the selected context to the MA and updates the registers for management of the circular buffers. Possible configurations of the PSU include the *size* of the buffer, the *base address* of the buffer in physical memory, and the *id* of the connected NI FIFO. The direction of the buffer specifies whether the buffer is an input, output, or an internal buffer. An input buffer receives data from a CA in the neighbouring tile while data from an output buffer is transferred to a CA in neighbouring tile. If the source and destination actors are in the same tile then an internal buffer is used.

The **Memory Arbiter (MA)** receives an active context from the PSU and executes it. The MA executes the data transfer by generating a memory address, memory control signal and NI FIFO control signals according to the received context. The MA switches context every two clock cycles and checks the next buffers' context.

Every context belongs to a buffer such that the MA transfers one word between the NI FIFO and the buffer and then moves on to the next buffer. The transfers are performed in the same number of clock cycles every time so that the CA has a predictable timing behaviour.

The buffers, which are managed by the CA, have been implemented as circular buffers. Circular buffers present contiguous address space to its user so that the programmer does not need to perform complex pointer operations while sending or retrieving data from these buffers. The CA performs these operations on the pointers and they are stored in the "PSU" module. The next subsection describes circular buffer management.

2.3.1 Circular Buffer Management

Our circular buffers are different from normal circular buffers in the sense that normal circular buffers have access restrictions like single assignment, in-order reading/writing etc. Our CA does not impose such restrictions. This is also the main reason why we need less memory than other CAs. To realize this, we use two sets of read and write pointers for each buffer. These 16-bit pointers are (Read Start (R_S), Read End (R_E), Write Start (W_S), and Write End (W_E)). We distinguish two pointer update schemes. The first scheme is employed when a buffer is configured as an output buffer. The write pointers (W_S, W_E) are updated in response to processor command *claimwritespace* and the CA increments the read pointers (R_S, R_E) on every transfer from the buffer to an NI FIFO. The second scheme is used when a buffer is configured as an input buffer. The read pointers are updated by the processor command *claimreadspace* and the CA increments its write pointers on each transfer from an NI FIFO to the buffer.

The buffer space claimed is only released with explicit release commands (discussed in the next subsection) whereas normal circular buffers do not have explicit

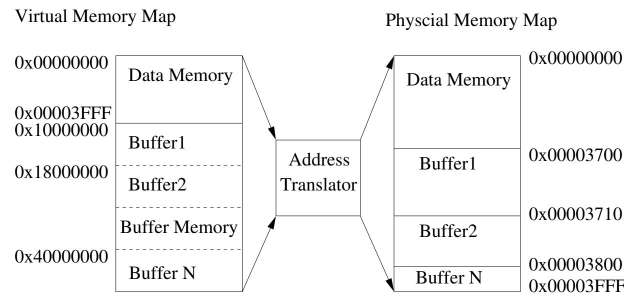


Figure 2.6: Address translation

commands for releasing the claimed space. Normal circular buffers use read and write signals to keep track of the accesses to the buffer. This results in restrictions on the access patterns inside the buffers (e.g. If a location is read then it cannot be read again). The use of explicit release commands allows our CA to support random access inside the claimed space. This also enables the use of the same memory region as computation and communication memory.

Figure 2.6 shows the address translation mechanism. During configuration of the buffers, the processor assigns each buffer an *id* and a physical address inside the memory. The CA appends the buffer id with its W_S/R_S pointer, depending on direction of the buffer, e.g. output/input, and sends it to the processor as a virtual address. The virtual addressing is employed because we want the same memory space to be used as data and communication memory. During the address translation process, the CA checks the buffer id of the address. A buffer id value of zero means normal data memory access and does not require any address translation. A non-zero buffer id means that the access is for the buffer memory and the CA sends the corresponding physical address to the memory. Note that the virtual buffer is assumed to be twice as big as the physical buffer. Consequently the translation from virtual to physical address requires only an "and" operation with an inverse of the buffer size. However, this simple translation restricts the size of the buffer to be a power of 2.

2.3.2 Programmability and Operation of CA

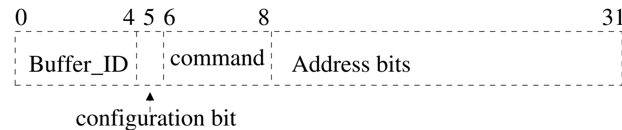
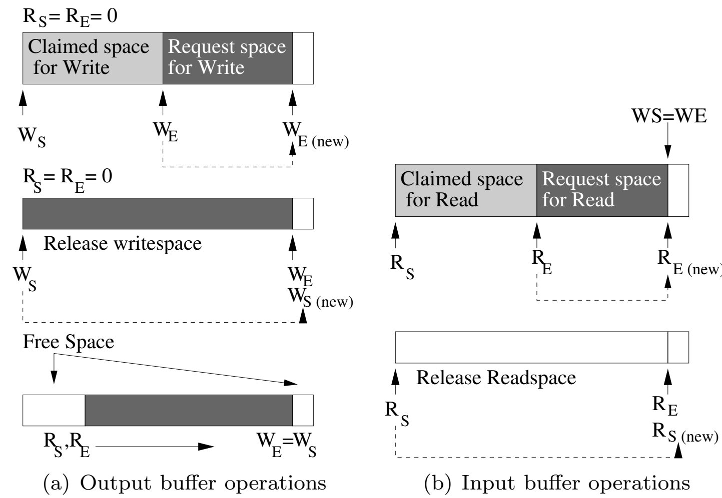


Figure 2.7: Address decoding for configuration, commands and accesses

In modern multi-processor systems such as multimedia platforms, applications may be started and stopped at run time. Consequently, the communication infrastructure should support reconfiguration during run time. The processor issues commands to the CA so that it can configure a buffer. These commands are decoded by command bits (6-8) and the configuration bit inside the CA, as shown in Figure 2.7.

A 3 clock cycle command *init(id, base, size, dir, ni_fifo_id)* sets the size (*size*), direction (*dir;input/output*) and the base address of the buffer in the physical memory (*base*). The user also specifies the NI FIFO identification number (*ni_fifo_id*), with which the buffer will be connected. The buffer is selected by its identification number (*id*).



(a) Output buffer operations (b) Input buffer operations

Figure 2.8: Pointer updates with commands. (a) Claim write space command moves the Write end pointer to its new location. Release write space moves the Write start pointer to Write end pointer. The space between write and read pointers is available for further claim commands. (b) shows similar operations for input buffer

The command *claimwritespace(id, nbytes)* reserves (*nbytes*) number of bytes in the configured buffer to store data to be sent to the buffer in the receiving tile. The CA checks whether the required amount of space is available using Equation 2.1 and sends the pointer to the processor (and updates $W_{E_{new}} \leftarrow W_E + nbytes$). If the processor wants to claim additional write space it can do so and the CA updates the write pointers and sends the new pointer to the processor as shown in Figure 2.8(a). The processor can then use the space as normal data memory. Once finished with the processing of data, the processor releases the space. During the release operation the value of W_E is copied in W_S. The CA starts to transfer the

data from the buffer into NI FIFO and the R_S, R_E pointers move towards pointers W_S and W_E. The command *claimwritespace(id,nbytes)* has been implemented to take two cycles as the pointer arithmetic (Equation 2.1) elongates the critical path.

$$S - (W_E - R_S) \geq nbytes \tag{2.1}$$

Similarly the command *claimreadspace(id,nbytes)* is used to read data from the selected input buffer. The command is blocking and verifies that *nbytes* are available before returning the pointer to the start of this buffer. Once the processor has read these bytes, it releases the read space. The release operation also copies the R_E into R_S pointer as shown in Figure 2.8(b).

$$(W_S - R_E) \geq nbytes \tag{2.2}$$

Our CA restricts the addresses for the buffers to be word aligned only. If the address would not be word aligned then two cycles are required to transfer the complete word from a word wide memory. This not only slows down the transfer but also complicates byte alignment logic which would consume precious hardware area. Therefore we restrict the base address of our buffers to be word aligned. It also makes it possible to provide tight guarantees on the timing behaviour of our CA.

2.4 Conservative SDF model of CA

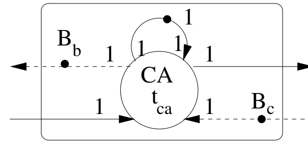


Figure 2.9: SDF model of CA [SSK+10].

The application SDF model can be refined to include the mapping decisions, buffer sizes and the timing impact of architectural components. This results into an SDF graph of the application and the architecture with a predictable behaviour. Our CA can be modeled as an actor with a self edge (see Figure 2.9). The self edge is given one initial token to model the behaviour that the next execution of the actor cannot start before the previous execution has finished. The dotted edges in Figure 2.9 model the buffer size of the CA. The CA polls the NI FIFO channels in round robin fashion. Every channel requires two cycles. During the first cycle the CA checks whether there is a word to be transferred from output buffer to the channel or from channel to the input buffer. The second cycle is required for the transfer. As the number of channels/CA increases, the

Table 2.1: FPGA Resources for a four channel CA.

	Proposed design	used resources of xc2vp30
No. of Slices	824	5%
No. of Slice flip flops	452	1%
No. of 4 input LUTs	1648	6%

response time of the CA gets larger. The CA takes t_{ca} number of clock cycles to transfer one word (Equation 2.3):

$$t_{ca} = 2 \times No_of_Channels \tag{2.3}$$

where $No_of_Channels$ is the number of NI FIFO channels which the CA has to manage.

In the combined model of the application and architecture, the model of CA is attached with the actor from one side whereas the other side is connected with the NI FIFOs. The depth of NI FIFOs is B_c words as shown in Figure 2.9. The rate at this edge is one as after each execution; the CA transfers one word from buffer to the NI FIFO or vice versa. Note that the direction of this edge will be reversed in case of an input buffer. Similarly B_b models the buffer space claimed by the processor for reading or writing. The rate at this edge is also one as one word space is released with each execution of CA. In the next section, we present SDF models of JPEG and Sobel along with our CA to illustrate the use of our predictable CA model.

2.5 Hardware Implementation

We implemented our CA on an XUP Virtex II Pro Development Board with an xc2vp30 FPGA. Xilinx EDK 8.2i and ISE 8.2i were used for synthesis and implementation. All tools run on a dual core 2.0 GHz with 1GB of RAM. Table 2.1 shows the resources claimed by CA. The CA takes only 5% of the resources of this medium sized FPGA. The maximum frequency of operation of CA is 108 MHz.

Implementation in Silicon

The MSAP presented in [SIAM+04] is very similar to our CA. It uses a control network for the hand-shake between the processors, before the actual data transfer. Our CA does not require a control network as it uses “back-pressure” as a flow control mechanism. This makes our CA more area efficient when compared to [SIAM+04]. Our CA compares favorably to classical DMA controllers. Table 2.2 shows the gate count (NAND2 equivalent) comparison of our CA with other architectures. The CA is synthesized for a clock frequency of 200 MHz. The design is implemented using Synopsis Design Compiler and 0.18μm Standard Cell

library (Standard Chartered). The results show that our CA is 44% smaller than a commercial DMA [ARM]. The hardware results for the CA by [MBB+05] are not available in the literature. Note that our CA does not require complex functionality like "scatter and gather"; this makes our CA light weight when compared with the architectures shown in Table 2.2. All of the designs have 8 channels.

Table 2.2: Gate count comparison with other DMAs.

Property	our CA	MSAP [SIAM+04]	PrimeCell [ARM]
queue config. (word)	32bit*8	32bit*8	32bit*4
gate count	36.3k	68k	82k

2.6 Experiments and Case Study

The SDF model of the CA needs validation before it can be used. A number of experiments are performed to validate the presented model. This section is further divided into four subsections. Subsection 2.6.1 presents SDF models for two multimedia applications mapped onto CA and non-CA based platforms, and compares the application period computed through analytical models with the measured period achieved from their FPGA implementations. An analytical tool SDF3 [SGB06a] is used to find the period of these applications. The subsection 2.6.2 shows the memory savings of the CA-based platform as compared to a non-CA based platform. Run time configuration of the CA is discussed in subsection 2.6.3. The final subsection 2.6.4 describes the effect of the CA on communication latency.

2.6.1 Analytical Models of Applications and Architecture

We implemented two real life applications (JPEG encoder and Sobel) to evaluate our CA. A CA-based platform, consisting of 4 Microblaze processors and 4 CAs, is compared with a non-CA based platform having 4 Microblaze processors only. FSL buses are used for interconnection in both platforms. The JPEG Encoder application is split into four actors as shown in top part of Figure 2.10. Each actor is mapped on one processor of the platform. The four actors are Macro-block Sampling (get_{MB}), Colour Conversion (CC), Discrete Cosine Transform (DCT), and Variable Length Coding (VLC). The first actor get_{MB} parses the input BMP file (RGB format) and sends macro-blocks to the CC actor. Each macro-block is 16×16 pixels big and 3 such macro-blocks are sent to the CC (one each for R, G and B pixels). The CC actor converts these macro-blocks into 4 luminance Y, and two Cr, Cb chrominance 8×8 smaller macro-blocks. These macro-blocks are fed to the DCT actor which is most compute intensive. The

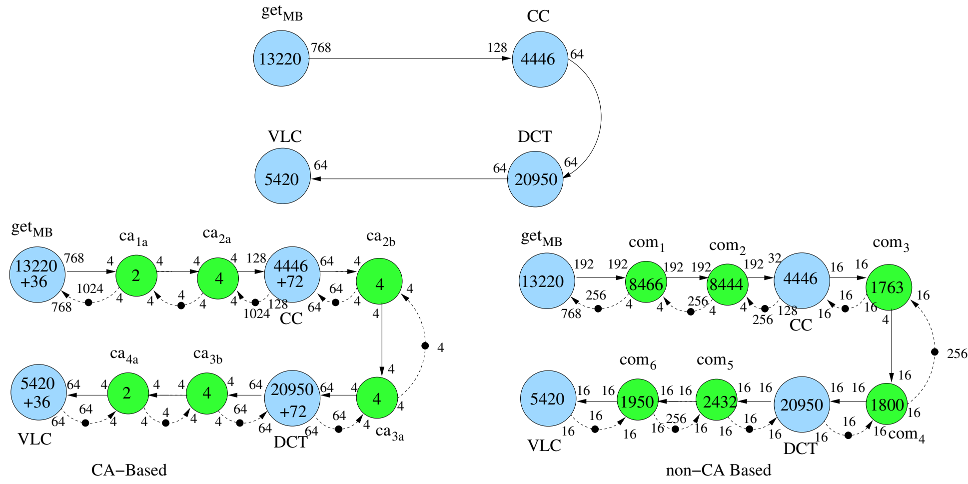


Figure 2.10: SDF graphs for CA and non-CA based platforms for JPEG.

DCT actor sends these 6 macro-blocks one by one to the VLC actor, where each of these macro-blocks is variable length encoded.

Sobel filter is extensively used in image processing, particularly within edge detection algorithms. Technically it is a discrete differentiation operator and computes the approximation of the gradient of the image. The reference implementation of Sobel is mapped on a 4-Microblaze platform. Figure 2.11 shows the SDF model of Sobel. The first actor (get_pixels) opens the input file stored in the CF card and loads it into the data memory. It then forwards 6 pixels each to the connected actors. These actors (GX,GY) find the gradient of the image in x and y direction. Finally the fourth actor (ABS) finds the absolute value of the gradients computed by the preceding actors.

The application graphs along with mapping decisions, buffer sizes and communication actors for JPEG encoder and Sobel are shown in Figure 2.10 and Figure 2.11 respectively. Worst case task execution times (WCET in clock cycles) of the actors are specified inside the circles in the graphs. Both platforms are optimized for period, which means that the period achieved by the platforms is the minimum given the resources consumed. The graphs on the side are for non-CA based platforms whereas the graphs on the left are of CA-based platforms. Self-edges are removed for more visibility in the Figures. The response time of each CA is calculated using Equation 2.3 whereas for a non-CA based platform it is measured through profiling.

The rates at the edges of graphs are specified in words. For example, 3 macro-blocks each of 16×16 pixels are transferred as 192 words from first actor get_{MB} to the communication actor com_1 as shown in Figure 2.10. The communication actor requires 8466 cycles to transfer these macroblocks to the receiving actor com_2 in

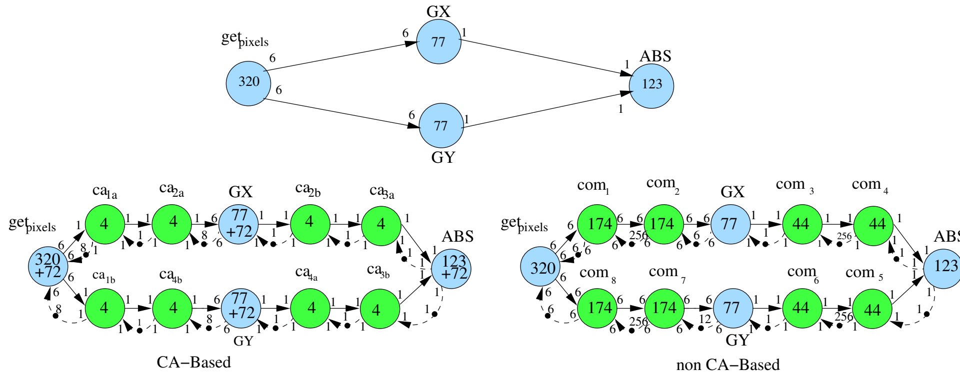


Figure 2.11: SDF graphs for CA and non-CA based platforms for Sobel

second processor. The channel buffer between the two actors is 256 words deep. Once the communication actor com_1 transfers the data it sends empty tokens to the get_{MB} actor so that it can start its next execution.

Similarly for CA-based platform, a CA has been attached with each actor to take care of all the communication[1]. The CA transfers one word on each of its execution. The actor CC has a two channel CA attached to it. One is for receiving the data and the other one is for sending it to the next actor. Due to these two channels, the transfer time for each word is 4 cycles as shown in Figure 2.10. The get_{MB} actor claims 192 words from a buffer of size 256 words. The CA frees one word space after each execution and the actor get_{MB} waits until a buffer space of 192 words is available to send the next set of macro-blocks. In CA-based platform, command (*claimreadspace* or *claimwritespace*) overhead of 36 clock cycles/channel has been added to the execution time of actors. This is a fixed overhead and it reflects the number of cycles needed by the processor to get the pointer to the buffer from the CA.

The period/iteration (in clock cycles) of these applications is calculated using SDF3 [SGB06a]. In one iteration, the JPEG encoder encodes 3 macro-blocks (R,G,B) and Sobel filters one pixel. The measured period from FPGA implementation is shown in Table 2.3. Our predicted worst-case period is very close to the measured one whereas non-CA based platform shows large deviation between worst-case and the measured (average case).

It is worth observing from the graph that the JPEG application has a very low communication to computation ratio. This means that the actor execution times are substantially larger as compared to the execution times of the communication actors. As the CA only accelerates the communication part so application speedup achieved in this case is not very high (only 8% as shown in Table 2.3). On the

[1] In the next chapter, we present a design flow which automatically adds the CA actors to anlyze the performance of CA-based platforms.

Table 2.3: Period for applications.

Appl.	CA-based (cycles)		non-CA based (cycles)		Gain
	SDF3	FPGA	SDF3	FPGA	
JPEG Encoder	126,168	122,543	140,292	132,921	8%
Sobel	359	355	668	640	45%

Table 2.4: Area overhead for both applications for Xilinx xc2vp30

Architecture	Slices	Bram	LUT	Used FPGA resources
CA-based	6,779	56	11,212	49%
non-CA based	4,075	60	6,587	29%

other hand for Sobel, the communication to computation ratio is relatively high so the speed up achieved in this application is as high as 45%. Further, the introduction of CA introduces predictability in the platform which is as important as communication speedup. The communication and computation are decoupled, and the processor is not stalled due to data transfer. Note that the CA is only effective if the communication network provides guarantees on the transfer. Many NoCs [RDG+04, MNTJ04] provide these guarantees and our CA can be used with them to generate NoC-based predictable MPSoC platforms.

2.6.2 Improvement in Memory Usage

Our CA-based platform uses half of the amount of memory when compared with that of [MBB+05]. This advantage comes from the fact that the CA of [MBB+05] first stores the data in communication memory and then shifts it to the data memory of processor whereas our CA-based platform uses the same memory for communication and data. Our CA also uses less amount of memory when compared with a non-CA based platform. Figures 2.10 and 2.11 show the SDF graphs for CA-based and non-CA based platforms. Here the communication memory is implemented as FIFO buffers between the communication actors e.g. there is a 256-word FIFO memory between actors com_1 and com_2 in Figure 2.10 whereas for CA-based platform this memory is only 1 word as shown in Figure 2.10. The FIFO memory is implemented as BRAMs so the non-CA based platform needs 4 additional BRAMs as shown in Table 2.4. The minimum application period available from non-CA based platform is achieved at the cost of this additional memory. Note that the number of BRAMs shown in the table also includes the ones needed to hold the code.

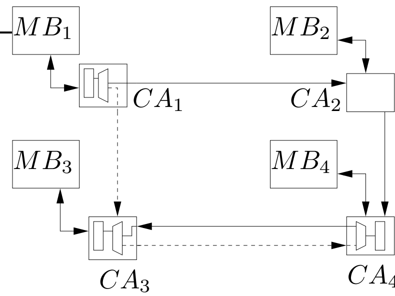


Figure 2.12: Reconfiguration of channels

2.6.3 Run-time Configuration of CA

In modern multimedia platforms, applications can be switched on and off by the user. A multi-processor platform should be able to re-configure so that other applications can also execute. In our experiment, both Sobel and JPEG encoder share the same platform. Once JPEG has finished its execution, Sobel is started. The solid lines in Figure 2.12 show the unidirectional FSL channels for JPEG encoder application and dashed lines show the additional channels for Sobel except the channel from CA_4 to CA_3 which is not needed in case of Sobel. It is evident from the figure that CA_1, CA_3 and CA_4 reconfigure once the JPEG has finished. Our configurable CA requires 2 contexts only at CA_3 and CA_4 to run both applications one after the other. If a non-configurable CA were used it would need 3 sets of contexts for three channels of CA_3 and CA_4.

2.6.4 Reduction in Communication Latency

We present an experiment to evaluate the speed improvement in data transfer for CA-based platforms. We implement two 2-processor platforms. The first one uses two CAs to communicate with each other whereas the second platform uses direct Microblaze-to-Microblaze FSL links. One processor produces the data whereas the second one receives it. We increase the number of words and measure the number of clock cycles required to transfer these words.

Figure 2.13 shows that for up to 4 words, the non-CA based platform takes fewer clock cycles as compared to the platform with CA. The CA-based platform should consume 8 clock cycles to send 4 words however, it takes 36 clock cycles. As stated earlier, the claim space commands take 36 cycles so the communication latency of 8 cycles is dominated by the command latency. As the transfer size increases beyond 5 words the CA outperforms the non-CA based platform as shown in Figure 2.13. For a transfer of 512 words, CA-based communication is 4 times faster than non-CA based communication (1024 clock cycles vs 4098 clock cycles). Therefore we conclude that our CA is faster for transfers of more than 5 words as compared to non-CA based platforms.

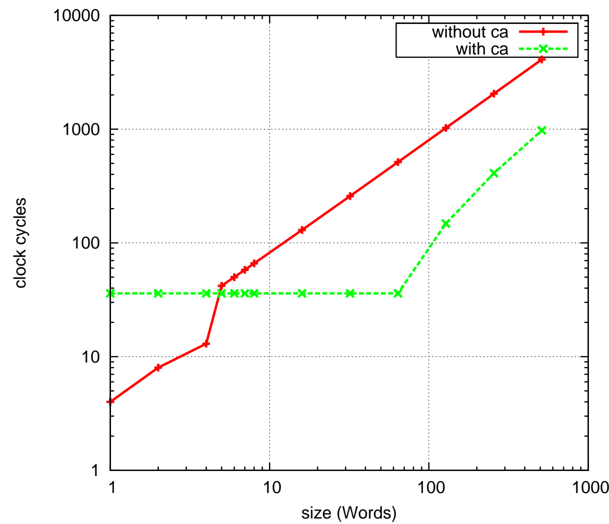


Figure 2.13: Number of clock cycles to transfer data in CA and non-CA based platforms.

2.7 Related Work

The communication controller presented in [NSD06] implements FIFO based communication between tasks. Writes to the FIFOs are always local to a processor whereas reads are always remote (from the FIFO memory of a producer). The programming model is based on Kahn Process Network (KPN) [Kah74]. Due to FIFO based communication, out-of-order, access, re-reading, and skipping is only possible after storing the data locally in the consuming task. In our CA-based platform, all the reads/writes to the memory are local to the producer/consumer. This results in saving of memory space as compared to the approach of [NSD06]. In [MBvdBvM07], the authors have presented a SystemC model of a CA. However, there are some key differences with our CA. They have used a single port data memory which is shared between the processor and the communication controller. We use dual-port data memory so that the processor does not stall when the CA is using the memory. Although our dual port memory may seem costly than their single port memory, they require an arbiter to share the port of the memory between processor and the CA. As they do not have a hardware implementation, we cannot compare the area overhead of their arbiter with our dual port memory.

In [GNL01] authors have presented a synchronization scheme for embedded shared memory systems. They propose channel controllers for synchronization of data between tasks. Out-of-order access inside the buffer is not possible as the buffer memory is implemented as FIFOs. They also have channel controllers

per channel while our implementation has one controller for all the channels. Authors in [BBjS08] describe communication between Nested Loop Programs (NLP) in multi-processor systems. The algorithm is implemented in software and can handle out-of-order access to the buffer. Both producer and consumer have their respective write and read windows for mutually exclusive access. Like our implementation, the authors have also used the C-Heap protocol. However, the algorithm is limited to single assignment codes; this means that a location in an array is assigned a value at most once per execution of the task. On the other hand, in our implementation the producer/consumer can write/read the same memory location as many times (multiplicity) as required before releasing the write/read space. We also support skipping, out-of-order, and multiplicity access patterns within the buffer. As a matter of fact our implementation is free from all these restrictions and the communication memory can be used as data memory.

With the method presented in [TKD04] a KPN is derived from NLP. In KPN communication between the tasks is arranged via FIFO buffers. When the consuming task has to read a location multiple times, the consumer stores the array in an additional buffer. Instead of FIFO buffers, we use circular buffers and also there is no need to copy values in an additional buffer. The work by [HDT07] is quite similar to [TKD04] and uses a read and write window. A window supports reading locations multiple times, reading and writing the locations out-of-order, or skipping the locations. However this work is also limited to single assignment code.

Cell BBE [Gsw06] implements communication between processing elements (SPEs) and the memory through DMA controllers called Memory Flow controller (MFC). The key difference between MFC and our CA is the fact that in MFC the synchronization between the memories has to be performed explicitly by the SPEs while in case of CA the synchronization is taken care of by the CA itself and processor is free from synchronization overhead. Table 2.5 gives a brief overview of the related work. For the same producer/consumer example, our CA implementation requires less memory as compared to message passing platform and still we can have pipe-lined task execution. The shared memory system on the other hand requires double buffering and mutual exclusion mechanism to have pipe-lined execution.

In [SIAM+04], authors have presented a distributed memory server to transfer data between tiles in an MP-SoC. The MSAP used in their server consists of a control and data network. The control network is used for end-to-end flow control while the data is transferred through the data network. We, on the other hand, use “back-pressure” as flow control mechanism and hence our implementation is very lightweight when compared to theirs.

Table 2.5: Comparison of memory usage in existing architectures.

Architecture	Data replication	Task Execution	out-of-order	code assign.
Shared Memory	No	alternate	yes	multiple
Message Passing	yes	alternate	no	single
CA	yes	pipe-lined	yes	multiple
ESPAM [NSD06]	yes	alternate	no	single
NLP[BBjS08]	no	alternate	yes	single

2.8 Conclusions

This chapter introduces a programmable CA which uses a shared data and buffer memory. This leads to lower memory requirement for the overall system and a lower communication latency. The CA has a predictable timing behaviour. This makes it possible to predict the performance of an application when mapped to a platform which uses our CA. These predictions can be used to provide timing guarantees on the application. It is observed that for applications with lower communication to computation ratio, the performance speed-up is not very significant as compared to the area overhead. On the other hand, for applications with high communication to computation ratio, the speed-up can be as high as 45%. In the future, a DSE methodology can be developed which can estimate the communication to computation ratio at each node and decide upon the placement of a CA at a particular node.

In the next chapter, we present an MPSoC design strategy that can generate MPSoC platforms capable of satisfying the throughput constraints of multiple applications in all use-cases. The platforms designed by this strategy use CAs as predictable communication components. The SDF model of our CA is used to predict the performance of applications when mapped onto the designed platforms.

CHAPTER 3

Predictable Multi-processor Design Approach

Modern multimedia systems support a large number of applications. Most of these applications execute concurrently and often require guarantees on their throughput. The multi-processor platform should satisfy the throughput constraints of these applications with minimum possible resources. The design of MPSoC platforms involves determining the minimum amount of required resources and their efficient utilization.

In the previous chapter, we have seen how to design and synthesize a predictable communication assist and model it in an SDFG. In this chapter, a multi-processor design strategy is introduced that can generate predictable platforms capable of satisfying the throughput constraints of multiple applications. The strategy incorporates the communication assist as a predictable component. As has been motivated in the earlier chapters, in embedded systems not all applications are active at the same time. For example, a mobile phone in one instant may be used to talk on the phone while surfing the web and downloading some Java application in the background, and in another instant be used to listen to MP3 music while browsing JPEG pictures stored in the phone, and at the same time allow a remote device to access the files in the phone over a Bluetooth connection. A strategy which does not optimize across the use-cases will dimension the platform with the assumption that all applications (Bluetooth, MP3 player, etc.) are active all the time. The sharing of resources with an eye on use-case information allows a smaller platform to meet the performance constraints of multiple applications. These resources include processors, an interconnect network and memories. The design strategy should share these resources efficiently, so

the first problem is to find the minimum number of processors as generally their impact on hardware area is the largest. Secondly, the communication network should be designed in such a way that it requires minimum silicon area. Thirdly, the size of the memories attached to the processors should be minimized.

The problem of mapping actors onto processors is NP-complete and the problem becomes NP-hard if the required number of processors is also unknown [GJ79]. We deal with the latter case.

There are a number of heuristic and genetic algorithm based solutions [KFH+08, CFR99, EEP03, HGR05, HM03, CCK+08, HCY+07, HG11, SBGC07, MMBM05, HW96, DJ97, SKC00, OH02, HFK+07] in the literature. Some of these approaches like Daedalus framework [EEP03] and [CCK+08, HCY+07] do not support platform generation for multiple applications. Others [HGR05, HM03, HG11] optimize only the communication flows on a given platform and [CFR99, SBGC07, MMBM05] solve the problem of mapping tasks on a given platform.

Although there has been some research on co-synthesis of multi-application systems [KFH+08, HW96, DJ97, HFK+07], only a few research results exist for multiple use-cases of multiple applications [SKC00, OH02, KFH+08]. The work by [KFH+08] assumes that the mapping of actors onto processors is provided. The approach proposed in [SKC00] does not optimize across the use-cases. The work in [OH02] does consider use-case optimizations between applications, but the used heuristic aims at reducing the computation requirements and does not consider sharing of communication resources; this does not always give the minimum platform. We, on the other hand, also look at the communication and memory requirements in our approach. This leads to potentially smaller platforms as compared to [OH02, SKC00].

The chapter is organized as follows. In Section 3.1, we highlight the problem of MPSoC design. Section 3.2 describes the architecture model. Our proposed heuristic algorithm is presented in Section 3.3. Section 3.4 explains task scheduling for application throughput measurement. In Section 3.5, the experiments to evaluate our algorithm are presented. Section 3.6 gives an overview of the work related to our method; finally, Section 3.7 concludes the chapter and discusses directions for the future work.

3.1 Motivating Example

In this section, we consider an MPEG4-decoder (simple profile) as a motivating example. This decoder supports video streams consisting of I and P frames. These frames consist of a number of macro-blocks, each requiring processing. Figure 3.1 shows the application graph of an MPEG4 decoder. The application consists of four actors namely Variable Length Decoding (VLD), Inverse Discrete Cosine Transform (IDCT), Motion Compensation (MC) and Reconstruction (RC). Task VLD processes one macro-block at a time and sends it to the IDCT and MC actors. The actor IDCT performs the inverse discrete transform of the whole

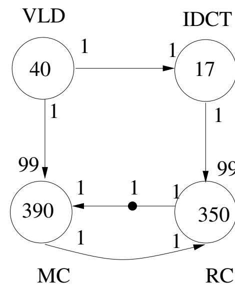


Figure 3.1: Application graph of MPEG4 decoder simple profile [TGB+06b]

macro-block and sends it to the actor RC. The actor RC works on a frame basis. A QCIF frame consists of 99 macro-blocks so the actor RC only executes when it has received a whole frame. Similarly, the actor MC executes once it has received one complete frame. The transfer granularity between MC and RC is one frame.

The figure also shows the execution times in kilo clock cycles measured on a ARM7TDMI processor. The actors RC and MC are the most computationally intensive actors among the four actors. Any technique based on load balancing will place them on different processors. Similarly, the technique described in [SKC00] considers the cost of computation elements and does not include the cost of memory or interconnect. Their approach would also place these two actors on different processors. Putting these two actors on different processors has an adverse effect on the memory requirement of the overall system. The reason for this high memory requirement is the fact that the actors MC and RC transfer complete frames between them. If these actors are placed onto different processors, they require at least one frame buffer at each processor. Additionally, at least one frame buffer equivalent of buffer space is also required between the processors (SDF model).

The techniques which map the actors onto processors purely on the basis of the computation cannot yield a minimal platform as they do not consider the memory requirements. In this work, it is argued that load balancing does not guarantee minimal platform for multi-application platforms when mapping multiple applications. Similarly, techniques which try to reduce the cost of the overall platform without taking into account the costs of memory and buffer requirement cannot yield a minimal platform either. In our algorithm, as a good starting point, an initial mapping is generated on the basis of load balancing the computation. The next steps in the algorithm try to move the actors onto the same processors so that the overall memory requirement of the platform is reduced (Note that we assume that the data memory is local to the processors). In this way, the generated platform may not have a perfectly balanced computation load but it will meet the throughput constraints of the given applications with fewer resources.

3.2 Problem Statement and Model Definition

The design of MPSoC platform for media applications requires that the applications are represented in some Model of computation. SDFGs are used as MoC in this thesis. Further, the architecture models are described. The assumptions and restrictions in this work are listed below:

- SDF graphs of the applications are available.
- The worst-case execution time (in clock cycles) of the actors on the processors is known. This can be achieved through static analysis [GSBC05].
- The platform consists of homogeneous processors connected through communication assists. The CAs are connected through point-to-point interconnect. Note that our method can be extended to heterogeneous platforms by adding a processing element selection step.
- The point-to-point interconnects are built from unidirectional FIFOs. Our work can also be extended to NoC based platforms as long as the NoC provides guarantees on the data transfer.
- Task execution is non-pre-emptive. In many practical systems, properties of the device hardware and software either make the use of pre-emption impossible or prohibitively expensive. For embedded systems in particular, non-pre-emptive scheduling algorithms are easier to implement than pre-emptive algorithms and have dramatically lower overhead at run-time [Bar06]. The down side of non-pre-emptive systems is their large response times as compared to pre-emptive systems. This means that the timing bounds of non-pre-emptive systems are often worse than pre-emptive systems.
- Each processor has its own instruction and data memory. Further in this work, we assume that the instruction memory is sufficiently large to store the code of the actors.
- We do not support task migration. Hence, the application mapping does not change across the use-cases. Task migration is used very rarely in embedded systems.

The representation of MPSoC architecture in the form of a model allows the designer to tweak the parameters and gets its performance estimates before actual implementation. The architecture model is represented by a graph (P, C) where the sets P and C denote the processors and their interconnections respectively. Note that our technique builds the architecture graph while most other techniques map an application graph onto a given architecture graph. The interconnections between the processors are based on point-to-point FIFOs. Interconnection $c_{kij} \in C$ represents the kth interconnection from processor p_i to processor p_j and has a storage capacity of Cp_{kij}.

Design Problem

A solution to the MPSoC design problem consists of a platform and mapping of the actors of all applications $\in \mathbb{G}$ onto this platform such that the throughput constraints of all applications are met. In doing so, the solution technique should try to minimize the hardware area. Minimizing the hardware area in general also results in lower energy consumption. The following constraints are to be satisfied:

- Each SDF actor has to be mapped onto a single processor. The processors are homogeneous so the actor can be mapped onto any one of the available processors.
- Each edge in the application model has to be mapped onto an internal memory of a processor or a FIFO in the interconnection between two processors. If two actors are mapped onto the same processor, then the communication edge(s) between these actors have to be mapped onto the same processor. If two actors are mapped onto different processors, then the communication edge(s) between these nodes have to be mapped onto an interconnection, containing a FIFO.
- The buffer capacity of an interconnection is the maximum of all the edges mapped to it. The interconnection network is shared between applications and it is possible that the same interconnection can be used by different applications in different use-cases. In this case, the buffer capacity of the interconnection should be the maximum of all the application edges mapped onto this FIFO.
- The throughput constraint of each application in each use-case should be satisfied.

The minimization objectives include the number of processors, number of communication links, buffer sizes of the FIFOs inside the communication links, and buffer sizes of the edges implemented in the data memories of the processors. The problem is addressed by taking one objective at a time, starting with the number of processors and then moving on to number of communication links and the buffer sizes inside the communication links.

3.3 Design Algorithm

To solve the MPSoC design problem, a heuristics-based algorithm is developed. The reason we choose a heuristic-based solution is the fact that heuristics can be tailored according to the problem and can provide good solutions in a reasonable amount of time. As an alternative, we could use a genetic algorithm or an Integer Linear Programming (ILP) based approach but in our problem the number of use-cases is exponentially related to the number of applications and these methods do not scale well with the problem size.

Algorithm 1 Determine Lower Bound on the number of processors

Procedure: lower_bound_proc(Applications)
Input: Applications $A_i \in \mathbb{G}$, actor execution times α, and repetition vectors γ
Output: Lower bound on number of processors

```
1: for all applications A_i ∈ G do
2:     total_load=0; req_proc=0
3:     for all actors a_j ∈ A_i do
4:         total_load+= γ(a_j) × α(a_j) // Processor load of each actor is calculated
5:     end for
6:     req_proc+=total_load × T(A_i)  // T(A_i) is throughput constraint of application A_i
7: end for
8: ⌈req_proc⌉ is a lower bound on number of processors
```

Our algorithm starts with an initial mapping which is based on computation load balancing. From this initial mapping, we check the processors, memory and buffer requirements of the mapping and try to reduce these quantities by moving actors between the processors. After each actor movement, the schedulability of the system is checked to see whether all application constraints are met. The algorithm is repeated until a mapping is found that satisfies the throughput constraints while trying to use minimum resources.

The algorithm starts by finding the lower bound on the number of processors as shown in Algorithm 1. The repetition vector entry of each actor $\gamma(a_j)$ is multiplied with its execution time $\alpha(a_j)$ and this is added for all actors. This gives the total number of clock cycles needed to complete iteration of the application graph. The throughput constraint $\mathbb{T}$(A) of an application A is the inverse of the average number of clock cycles in which the application must complete iteration. By multiplying these two quantities we get the minimum number of processors required to meet the throughput constraint of an application.

For example, if the total load of an application is 2 million clock cycles and the throughput constraint is such that iteration should be completed in 1 million cycles, we need at least two processors to meet this throughput constraint. This is the lower bound on the number of processors for one application. We perform this step for all the applications mapped onto the platform. When multiple applications are executing concurrently, then the required number of processors may be more than this lower bound due to interference between the applications.

The upper bound on the number of processors is set to the total number of actors of all applications e.g. each actor can have its own processor which is the maximum possible task level parallelism in all applications. Our design algorithm finds the number of processors between these bounds. We choose the *number of processors* as our first optimization objective since in most platforms the size of the processor is the most cost intensive in terms of area. We start from the lower bound as it is observed that for most designs, the optimal number of processors is near the lower bound. The actors are then sorted in decreasing order of their processing requirement as shown in line 3 of Algorithm 2. Sorting the actors in decreasing order of their processing requirement allows us to map the actors

with large processing requirement in the early stage. Generally, actors with large processing requirement influence the throughput more as compared to the actors with lower processing requirement. We pick an actor from the sorted list and map it to the processor having minimum load. This means that the actors are mapped evenly with respect to processing load, as we keep on picking them through the sorted list. This way, the initial mapping is generated using the load balancing technique (line 4 Algorithm 2).

Algorithm 2 Find the hardware needed to satisfy throughput constraints of multiple applications

Procedure: Find_MPSoC_platform(Applications, Constraints)
Input: Applications $A_i \in \mathbb{G}$ and Throughput constraints $\mathbb{T}_{A_i}$
Output: All applications satisfy their throughput constraints with the generated hardware platform

1: upper_bound=$\sum_{ij} a_{ij}$// Total number of actors in applications
2: N_{proc} = lower_bound_proc($\mathbb{G}$)// lower bound on processors as obtained from Algorithm 1
3: sort_actors_decrease(actors, execution_time)// Sort actors in decreasing order of their processing load
4: mapping=(actors, N_{proc})
5: platform_found=false
6: **while** (platform_found=false) $\wedge$ ($N_{proc} \leq$ upper_bound) **do**
7:     **for all** Applications $A_i \in \mathbb{G}$ **do**
8:         **for all** Actors $a_j \in A_i$ **do**
9:             $\mathbb{T}_{A_i}$=Evaluate_mapping_sdf^3(A_i) in all use-cases
10:             **if** (Thr(A_i) $\leq \mathbb{T}_{A_i}$) **then**
11:                 **if** REMAP($a_j \in A_i$) == $SUCCESS$ **then**
12:                     platform_found = true
13:                 **end if**
14:             **end if**
15:         **end for**
16:     **end for**
17:     $N_{proc}++$
18: **end while**
19: **if** (platform_found = true) **then**
20:     Minimize-Connections()
21:     Edge-interconnect/data memory Optimizations
22: **else**
23:     Change actor granularity
24: **end if**

When all the actors from the list are mapped, this mapping is checked and the use-cases and applications which cannot satisfy their throughput constraints are listed (Throughput measurement is discussed in the next section). We remap the actors of the applications failing their constraints to the processor with the lowest processing load (line 11, Algorithm 2). Note that the throughput of an application is measured in all use-cases in which it is active. So the resources allocated for this application are free to be used by other applications in the use-cases where this particular application is not active. The super-set hardware is generated from this individual use-case resource usage information.

Algorithm 3 REMAP ACTOR FUNCTION

Procedure: REMAP(actor)
Input: actor a $\in A_i$
Output: Generate new mapping

```
 1: p=processor_lowest_load(P)// Find processor with lowest processing load
 2: Remap a to p
 3: for all Applications A_i ∈ 𝔾 do
 4:    for all Actors a_j ∈ A_i do
 5:       Evaluate-mapping(A_i) in all use-cases
 6:       if (Thr(A_i) ≤ 𝕋_{A_i}) then
 7:          return (FAILURE)
 8:       end if
 9:    end for
10: end for
11: return (SUCCESS)
```

The "REMAP ACTOR" routine is shown in Algorithm 3. First, processor with lowest load is found and then the actor is mapped to this processor. The new mapping is evaluated again and this step is repeated until all actors of the applications failing throughput constraints are moved to processors having lower processing load. If a mapping that satisfies the throughput constraints is not found then the number of processors is increased (line 17 algorithm 2) and the algorithm continues until the mapping satisfies the constraints is found or until the upper bound on the number of processors is reached. In the latter case, no feasible platform and mapping is found.

Once a mapping that satisfies the throughput constraints of all applications is found, we turn to our second objective i.e. *minimizing the number of interconnects* (line 20 of Algorithm 2). The minimization of communication interconnect is performed by iterating over all applications and checking each edge. If source and destination actors of an edge are mapped onto the same processor then we move to the next edge otherwise we shift the destination actor to the same processor as that of the source actor. The new mapping is checked and if it satisfies the application constraints, then one interconnect is reduced. This process is repeated for the next edge. If an application fails its constraints then the previous mapping is kept and we move to the next edge. During the shifting of destination actors to the processor having the source actor, the number of processors may also decrease if the destination actor is the only actor on that processor. A similar heuristic to reduce the number of communication links has also been proposed by [IB10] quite recently.

The final step of our algorithm tries to reduce the sizes of the *FIFO buffers* in the point-to-point interconnect and the sizes of *data memories* of processors. The buffer sizes assigned to the edges of the application graphs are calculated by the technique described in [SGB06b]. The technique gives all the trade-off points between throughput of an application and the buffer sizes of the edges between its actors. We choose the buffer sizes which are just enough to meet the throughput constraints of the applications (Note that it is possible that the buffer sizes are

not sufficient as the mapping is not modelled in the SDF graphs yet; so multiple iterations may be needed to select the correct buffer sizes).

The reduction in sizes of FIFO buffers is possible when there are at least two interconnects between two processors. This simple step assigns edges having similar buffer requirement to the same interconnect. Figure 3.2 shows one example of edge to buffer assignment. Two use-cases are visible showing two channels each with their buffer requirements. The bottom part of the figure shows two options for implementation of the super set platform and shows that by choosing option 1, a reduction of 3 units in buffer size can be achieved.

As described earlier, when the source and destination actors of an edge from the application graph, are mapped to the same processor the edge is also mapped to the same processor. The mutual exclusion of applications allows us to dimension these buffers in the data memories of the processors in such a way that this memory can be used by different applications in different use-cases. This is achieved for every processor and for each use-case in the super set architecture; the data memory for these edges is configured to be the maximum of all the edge assignments in the use-cases.

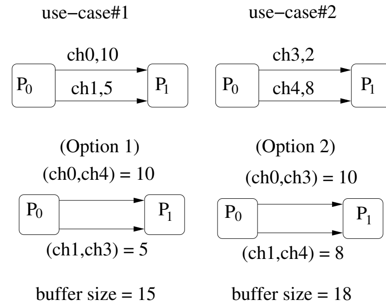


Figure 3.2: Edge-interconnect assignment

3.4 Task Scheduling and Throughput Measurement

In a system with predictable timing behaviour, a scheduling mechanism must order the execution of actors such that it is possible to provide guarantees on the maximum amount of time between the moment that an actor is ready to fire and the completion of the firing (its response time). We employ a round robin scheduler between the actors from different applications. The actors from the same application are scheduled using a static order scheduling. Static order scheduling is used as it provides tight timing guarantees on application throughput [SB00].

To measure the throughput of applications from their mappings onto processors, the SDFGs are updated with buffer sizes and communication latency. The

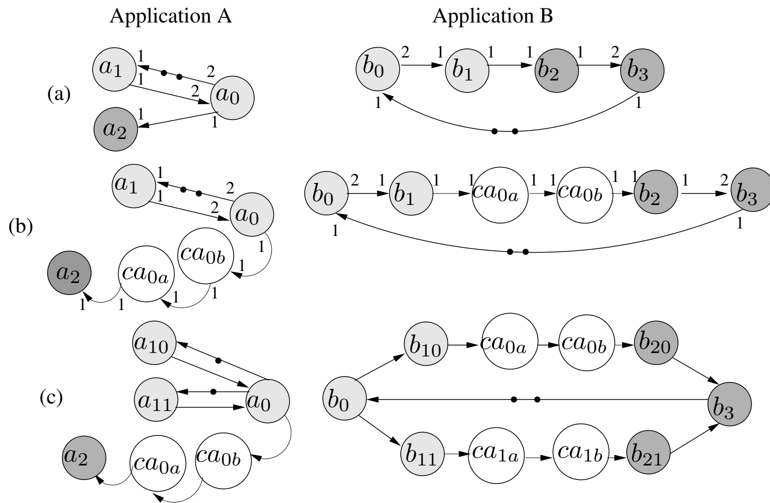


Figure 3.3: Application SDF graphs (a) are converted into communication aware graphs (b) and then to HSDFGs (c).

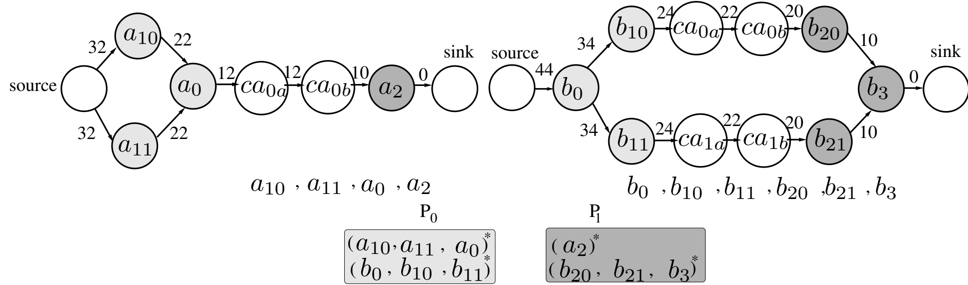


Figure 3.4: Extraction of static order schedules (bottom) from application APGs (top)

communication latency is modelled by adding actors in the application graphs between the actors mapped onto different processors.

To find the static order schedules, the application SDFGs are converted into Homogeneous Synchronous Dataflow Graphs [SB00] (HSDFGs). The HSDFGs are then converted into Acyclic Precedence Graphs [SB00] (APGs). The conversion removes all the edges containing initial tokens. Choosing APGs gives us a single iteration schedule which requires small memory space for its storage. The weights of the nodes of APGs are equal to the execution times of corresponding actors in the HSDFGs. To extract the schedule from the application APGs, weights are assigned to its edges. The weights are assigned starting from the bottom node of the graph and going to the top node using Breadth First Search (BFS). At each node, the weight of incoming edges is calculated by adding the weight of the node with the maximum weight of all the outgoing edges from the node.

Once the edges have their weights, the APG is traversed from the top node to the bottom node in BFS; again the nodes having large edge weights are visited first. In this way, the actors having large execution times are scheduled first as the objective is to minimize the period of the graph (make span). Once schedules for each application are obtained, they are enforced in the application graphs by adding edges [SB00]. The process of enforcing the schedules is performed on a per-processor basis e.g. the actors from the same application mapped onto the same processor. These edges depict the order in which the actors on the same processor have to fire. Assume that the actors $a_1 a_2 a_3 ... a_n$ are scheduled in this order in a static order schedule. To enforce this schedule, the method adds a cycle with edges $((a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n), (a_n, a_1))$ with one initial token on edge (a_n, a_1) to the HSDFG.

Figure 3.3 shows an example of SDFGs of two applications. Assume the actors in these graphs have execution time of 10 clock cycles each. The applications are to be mapped onto a two-processor platform. The mapping is shown with the help of shading in Figure 3.3. The graphs are converted into communication aware SDFGs by adding actors which model communication delay. The SDF model of our communication assist (as described in chapter 2) is added between the actors mapped onto different processors. For example, actors b_1 and b_2 are mapped onto different processors so communication actors representing "send" and "receive" functions are added between these two actors. The resulting graphs are then transformed into HSDFGs as shown in Figure 3.3c. The HSDFGs are converted into APGS as shown in Figure 3.4. Edge weights are assigned to APGs as shown in Figure 3.4. To extract schedules from these APGs, we traverse in BFS from source node in APG to the sink node and going through the nodes which have large execution times. Figure 3.4 shows the extracted schedules for processors P_0 and P_1 from the APGs. The processor level schedules are extracted from these application level schedules.

Next, the actor response times are calculated according to the context-switch between the applications. A context-switch is allowed to occur after the complete execution of an actor. This means that the waiting time has to be added in

the execution time of each actor. Algorithm 4 shows the pseudo code of our waiting time calculation technique. These waiting times are calculated per use-case and per-processor basis. The actors from an application may have to share the processor with actors from the other applications. Each time an actor is ready for execution, in the worst-case it has to wait for the actor having maximum execution time from other applications as shown in line 5 of algorithm 4. These waiting times are added to the execution times of the actors to model the sharing of the resources. The updated HSDFGs are used to find the throughput using

Algorithm 4 Find the waiting times of the actors due to processor sharing.

Input: Calculate the waiting times of the actors a_{ij} from application i in case of a shared resource
Output: Waiting times of the actors calculated
1: **for all** processors $p_m \in$ P **do**
2: **for all** Applications $A_i \in \mathbb{G}$ **do**
3: **for all** Actors $a_j \in A_i$ **do**
4: **if** Application A_i is active in the use-case **then**
5: $t_{wait}(a_{ij}) = \sum_{k=1,k\neq i}^{N}$ max α_{ki} $\forall$ j : a_{ki} is mapped on processor p_m // α_{ki} is //the execution time of kth actor from active application A_i in the use-case
6: **end if**
7: **end for**
8: **end for**
9: **end for**

SDF[3]. The context-switch in non-pre-emptive systems occurs after the completion of execution of an actor.

3.5 Experiments and comparison with other techniques

In this section, we compare our technique with similar works [SBGC07, SKC00, OH02]. The work by [SKC00, OH02] treat the problem as hardware/software co-synthesis while [SBGC07] maps applications onto a given platform.

Comparison with a mapping technique

Table 3.1: Use-cases for 3 application case study

use-case	H.263 decoder	H.263 encoder	mp3 decoder
0	-	-	1
1	-	1	-
2	1	-	-
3	1	1	-
constraints	10 frames/sec	10 frames/sec	38.46 samples/sec

There are three important differences between our technique and the technique presented in [SBGC07].

1. The technique from [SBGC07] maps the applications onto a given MPSoC while our approach *generates* an MPSoC.

2. We exploit the use-cases to save resources. The technique from [SBGC07] assigns separate resources to each application. Hence, it assumes that in the worst-case, all applications are active simultaneously.

3. The technique from [SBGC07] assumes a platform with pre-emptive scheduling whereas we target a platform that uses non-pre-emptive scheduling.

We use three applications, a H.263 decoder, a H.263 encoder, and an MP3 decoder, to make this comparison. Using these three applications, we created four different use-cases as shown in Table 3.1. Each active application in a use-case is represented with a "1" at its position. The throughput constraints for each application are also shown in the table and they are the same in all use-cases. These constraints can be converted into iteration/cycle format, which are used by our algorithm. For example, if the processors have a clock frequency of 200 MHz and H.263 decoder has to decode 10 frames/sec then the throughput constraint for the graph is 5.0×10^{-8} iterations/cycle.

Table 3.2 shows the resources consumed by the two techniques. Columns (2-4) show the number of actors of each application mapped onto the various processors (p_i). The table also shows the number of connections through the interconnect (#FIFOs) and the total buffer size used by each application (memory). The results show that our approach, when considering use-cases, requires 33% fewer processors, 93% fewer interconnect FIFOs and 1% less memory as compared to the technique from [SBGC07]. These resource savings come from the fact that we exploit the property that not all applications are active simultaneously. The last columns (labeled 'Ours (with all applications)') shows the resource usages of our strategy when we would not consider use-cases (i.e. we would assume that all applications are active simultaneously). The results show that in this situation we use more processors as compared to the technique from [SBGC07]. The reason for this high resource usage is the fact that we use non-pre-emptive scheduling which has larger response times as compared to pre-emptive scheduling techniques.

From the results it can be observed that the processor load of the platform generated ('with use-cases') by our technique is not balanced. Only one actor has been mapped onto processor P_1 while all others have been mapped onto processor P_0. The load on processor P_1 is 18% of the load on processor P_0. Our algorithm tries to keep the communicating actors on the same processing element so that the communication memory and buffer memory can be reduced along with the processing elements. The case-study shows that optimization of resources across different use-cases results in smaller platforms.

Table 3.2: Resource used by [SBGC07] and our algorithm.

Algorithm	[SBGC07]					Ours (with use-cases)				Ours (with all applications)					
	actors			edges		actors		edges		actors				edges	
Applications	p0	p1	p2	#FIFOs	memory	p0	p1	#FIFOs	memory	p0	p1	p2	p3	#FIFOs	memory
H.263 decoder	1	2	1	3	1194	4	0	0	1196	0	0	0	4	0	1196
H.263 encoder	3	1	1	4	304	5	0	0	304	5	0	0	0	0	304
mp3 decoder	5	3	6	7	19	13	1	1	18	0	1	13	0	1	18
Total	3 proc			14 FIFOs	1518 bytes	2 proc		1 FIFO	1500 bytes	4 proc				1 FIFOs	1518 bytes

Comparison with a co-synthesis technique

The co-synthesis strategy presented by [SKC00] synthesizes heterogeneous platforms for multi-application systems. Their approach does not optimize across the use-cases and generates individual embedded system platforms for each use-case. The example presented in their paper consists of three types of processing elements and four applications consisting of 10 actors each. The execution time of each actor while executing on a given processing element type is shown in Table 3.3. They tested three use-cases as shown in Table 3.4. The cost for individual platforms is shown in Table 3.4. The cost of each communication link is $20, as given in [SKC00].

Table 3.3: Cost and computation time of each actor on each PE as given in the example from [SKC00].

PE	cost	Computation time of actors on different processors									
	($)	a	b	c	d	e	f	g	h	i	j
X	100	5	10	5	35	15	30	15	15	7	10
Y	50	12	18	12	85	22	75	25	35	10	28
Z	20	18	40	18	195	80	180	85	47	30	35

Table 3.4: Comparison with [SKC00].

	[SKC00]			Ours, X-processors			Ours, Y-processors			Ours, Z-processors		
use-case	#pe	#link	cost	#pe	#link	cost	#pe	#link	cost	#pe	#link	cost
task1,task2	2	1	$170	1	0	**$100**	2	0	$100	4	4	$160
task1,task3	3	2	$240	2	2	**$240**	4	2	$240	8	5	$260
task3,task4	4	1	$210	2	1	$220	3	2	**$190**	7	4	$220

The strategy presented by [SKC00] selects the processing elements such that the cost of the platform is minimized and the throughput constraints of the applications are satisfied. We performed experiments considering only X, Y and Z type of processors. Table 3.4 shows that out of three platforms generated by our strategy, two are cheaper as compared to the platforms generated by [SKC00] and one has the same cost. The co-synthesis strategy [SKC00] starts by selecting the cheapest processing elements, and if the application constraints are not satisfied then it replaces the processing elements with costlier processing elements. Their strategy does not thoroughly search the homogeneous platforms and that is why our strategy can generate low cost homogeneous platforms as compared to the platforms generated by [SKC00]. Additionally they do not perform optimizations for communication channels while our strategy does.

Table 3.5: An example multiple use-case embedded system.

Use-cases	$\prod_1$				$\prod_2$		$\prod_3$
Applications	τ_1	τ_2	τ_4	τ_5	τ_2	τ_3	τ_3
Period	100	100	25	25	50	40	40
Deadline	100	100	25	25	50	40	40

Table 3.6: Execution time of actors when mapped onto processor $ARMP_1$ and $ARMP_2$.

	HW	$ARMP_1(100)$	$ARMP_2(900)$
me	17	518	259
diff		5.2	2.6
dct		17	8.5
q		11.4	5.7
vlc		16	8
deq		4	12
idct		18	9
mc		7.2	3.6
pd1		4.4	2.2
pd2		0.7	0.4
hd		3	1.5
demq		1.2	0.6
imdct		5.7	2.9
fb		10.2	5.1
τ_4		1.6	0.8
τ_5		1.8	0.9

Comparison with [OH02]

The co-synthesis strategy presented by Hyunok et al. [OH02] is based on three steps. It selects appropriate processing elements, maps and schedules the actors to the selected processing elements and performs schedulability analysis. Their objective is to reduce the overall cost of the system while meeting the throughput constraints of multiple applications. Like our technique, they also look at mutual exclusion conditions of applications and claim to provide smaller platforms as compared to techniques which do not consider use-cases. However, there are some differences in the assumptions of application models and generated platforms. They assume that applications are modelled as acyclic graphs whereas we assume that the applications are modelled as SDFGs. Moreover, the platforms synthesized by their approach require pre-emption while the platforms synthesized by our technique are non-pre-emptive.

The example given in the paper by Hyunok et al. consists of three use-cases and five applications. The system supports three different use-cases: video phone

($\prod_1$), video player ($\prod_2$), and MP3 player ($\prod_3$), consisting of five different applications: H.263 encoder (τ_1), H.263 decoder (τ_2), MP3 decoder (τ_3), G.723 encoder (τ_4), and G.723 decoder (τ_5). Table 3.5 shows which applications compose which use-case of operation. For instance, the video phone mode runs 4 applications $\tau_1, \tau_2, \tau_4, \tau_5$.

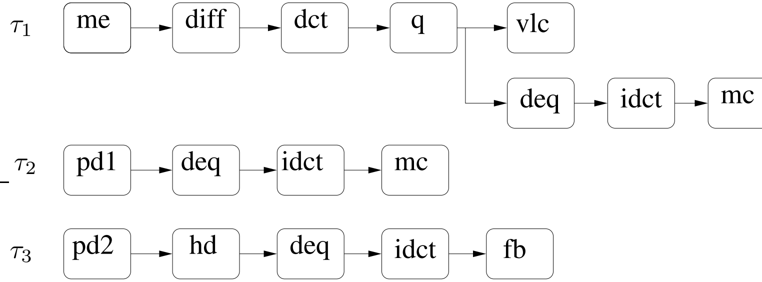


Figure 3.5: Applications specified as acyclic graphs

Each application is specified by an acyclic graph as shown in Figure 3.5. Applications τ_4 and τ_5 consist of only one actor so the graphs are not shown here. Table 3.6 shows the execution times of actors when mapped on processors of type $ARMP_1$ and $ARMP_2$. The timing information for these actors has been obtained by running them on 500MHz ARM processor ($ARMP_1$), while processor ($ARMP_2$) is twice as fast. The cost of $ARMP_1$ is 100 while $ARMP_2$ is 9 times costlier than $ARMP_1$ as shown in Table 3.6. Note that the execution time for actor *me* is very large and we assume that for both techniques its hardware module is used. We perform two experiments. In the first experiment, we assume that we can only use ARM processors of type $ARMP_1$ for generating MPSoC platforms and for the second experiment we assume that only the ARM processors of type $ARMP_2$ can be used.

The platforms generated by both techniques are shown in Figure 3.6. The figure does not show the hardware module motion estimation *me* as it has to be connected with all platforms and we have omitted the module for better readability. The use-case $\prod_1$ is most compute intensive and determines the maximum resource requirement of the complete platform. The technique by [OH02] defines *slack* as the difference between the utilization constraint and current utilization and moves the actors to the new processors so that the slack is reduced. Initially, all actors are mapped to processor P_0 and a schedulability test is performed. The applications do not meet their constraints so a new processor P_1 is added. Application τ_1 is most compute intensive so some of its actors are moved to the newly added processor. The utilization of processor P_0 by application τ_4 is very small so it stays on processor P_0 and shares the processor with the remaining actors of application τ_1.

The optimal resource requirement (rate-monotonic scheduling) for this exam-

ple is a 2-processor platform. One processor is solely required for application τ_1 and the rest of the applications are mapped onto the second processor. The technique by [OH02] cannot find this mapping as it moves the actors on the basis of their individual actor utilization and does not look at the total computation requirement of an application. The technique by [OH02] requires a 3-processor platform while our strategy requires a 2-processor platform for these applications. The cost of the platforms (excluding the cost of hardware module *me* for all platforms) is shown in Figure 3.6. Our strategy finds a cheaper solution. We assign the processors to individual applications by moving the actors of an application which are initially mapped to other processors, to the same processor. In this way, not only the communication infrastructure requirement is reduced, the overall execution time of the application is also reduced.

In the second experiment, we assume that only ARM processors of type $ARMP_2$ are available to generate the MPSoC platform. Both methods require only 2-processor platforms of type $ARMP_2$ for the given constraints. The execution times of the actor while running on ARM processor of type $ARMP_2$ are lower as compared to the case when they are executed on ARM processors of type $ARMP_1$ so only a 2-processor platform is sufficient. Note that our strategy also requires a 2-processor platform for this experiment. In these experiments, our strategy was able to find the cheapest possible platform.

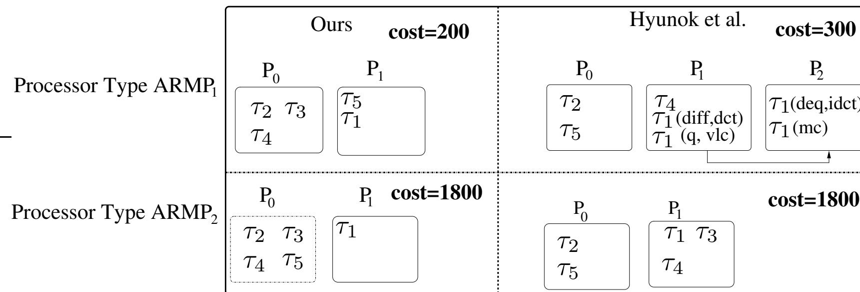


Figure 3.6: Comparison of the generated platforms, our technique vs [OH02]

3.6 Related Work

In recent years, a number of authors have considered the problem of mapping task graphs to an architecture graph. Hu et al. [HM03] use a branch and bound algorithm to traverse the task graph to architecture graph mapping search space. The authors in [SBGC07] and [MMBM05] also solve the mapping problem. Stuijk et al [SBGC07] employ a design time heuristic to assign and schedule a task graph onto a given architecture graph. The mapping of tasks to the tiles is determined

by a cost function. They assume that the MPSoC platform is provided by the user. We on the other hand generate the MPSoC platform.

Yang et al. [YWM+01, YC03] use a genetic algorithm for design time detection of the most promising task graph to architecture graph assignment and scheduling option. At run-time they use a greedy heuristic to select the right option for all active task graphs. Ma et al. [MSC07] extend this approach by introducing a run-time local search heuristic to further optimize the scheduling (not assignment) produced by the greedy heuristic. Our approach does not require any run time selection.

Works in [LK03, WAHE03, EEP03] use genetic mapping algorithms and try to optimize application execution time and energy consumption respectively. They assume that the number of processors is already given and they basically optimize for energy by changing task to processor mappings. Additionally the methods in [EEP03, WAHE03] are for single application only and do not provide guarantees on the performance. A Taboo search algorithm is adopted in [HG11, MEP07] to explore large search space for finding the placement of tasks. However the approach is communication centric and is directed towards optimizing the communication bandwidth and latency.

Chou et al. [CM08] propose a run-time mapping strategy that incorporates user behaviour information in the resource allocation process. Nollet et al. [NAE+08] describe a run-time task assignment heuristic for mapping the tasks on an MPSoC containing FPGA fabric tiles. Smit et al. [SHS05] present a mapping algorithm to map an application task graph on an MPSoC at run-time. The algorithm places each task near its communicating entity in order to save the energy consumption.

The work by Hung et al. [HCY+07] generates a NoC-based multi-processor platform and uses branch and bound algorithm to select the required number of processors from a library of processors. However, their work does not support multiple applications. SystemCoDesigner [HFK+07] present a simulation based design space exploration and system synthesis methodology. The user can select the model of computation from a range of options. The methodology cannot optimize across multiple use-cases.

Hansson et al. [HGR05] use a greedy design time heuristic to map an application graph of IP blocks onto an empty interconnection network. Further, their processors are not shared by applications. On the other hand, our technique determines the required platform resources at design time. MAMPS [KFH+08] is a design flow that can synthesize platforms for multiple applications. The mapping step in MAMPS is manual and the user provides the actor-to-processor mappings. Authors in [SCT09] optimize the multi-processor platform for power and energy. Like our work, they also assume to know a priori the mutual exclusion conditions for executing applications, but that information is presented as probabilities.

The work by [IB10] presents 3 heuristics to synthesize MPSoC platforms for applications modelled as acyclic graphs. They employ all three heuristics and choose the best result. Their "edge elimination" heuristic is similar to the "minimize communication links" step in our algorithm. Their work is for acyclic graphs while

we model applications as cyclic dataflow graphs. Further, they cannot synthesize MPSoC platforms for concurrently executing multiple applications. In [LSG+09] the authors present a synthesis technique to generate MPSoC platforms. They use multi objective optimization to dimension computation and communication resources of an MPSoC. Their work is limited to single applications and hence they cannot handle use-cases.

There has been some research on co-synthesis of multi-application systems [HW96, DJ97] however, only a few research results exist for multiple use-cases of multiple applications [SKC00, OH02]. The approach proposed in [SKC00] tries to map all applications onto processors and checks if all applications in all use-cases are schedulable. If a schedulability constraint is violated, they single out the best application (having highest throughput) and implement some of the tasks from this application onto hardware. This way the execution time of the application is reduced. However, this method does not consider resource sharing between applications. The work by [OH02] does consider resource sharing between applications but it considers the computation requirement for their heuristic which does not always give the minimum platform as shown in Section 3.1. We on the other hand, also look at the communication and memory requirement in our approach. Additionally, their technique assumes pre-emptive task scheduling while we use non-pre-emptive task scheduling.

3.7 Conclusions

In this chapter, a novel algorithm to generate MPSoC platforms that can meet the throughput constraints of multiple applications in all use-cases is presented. Experimental results show that the algorithm can guarantee throughput of multiple applications with fewer resources than existing state-of-the-art techniques [SBGC07, OH02, SKC00]. The algorithm presented here takes into account computation, communication and memory requirements in order to generate a minimal MPSoC platform for multiple use-cases. It also guarantees that the applications meet their constraints in all use-cases. We also observe that computation load balancing based techniques are not adequate for embedded system design. One of the limitations of our work is that presently it only generates platforms with point-to-point networks. In the future, it will be extended to NoC-based platforms. As mentioned, it can be easily extended to heterogeneous platforms also by introducing processing element selection step.

After describing the platform generation strategy in this chapter, the next chapter is dedicated to synthesis of the generated CA-based platforms onto FPGAs. The technique can synthesize the platform with the help of commercial FPGA synthesis tools.

CHAPTER 4

Multi-processor Platform Synthesis

Modern multimedia embedded systems have to support a large number of independent applications. In the area of portable consumer systems, such as mobile phones, the introduction of new technology solutions is increasingly driven by applications [ITR07]. Tile-based multi-processor platforms [TKM+02, KRH+03, KTN+00, FSV99, SVM01] are increasingly being used in modern embedded systems to meet tight timing and high performance requirements of these large number of applications and their *use-cases*.

In Chapter 2, a multi-processor platform has been introduced that decouples the computation and communication of applications through a hardware *communication assist* (CA). This decoupling off-loads the communication load from the processor, thereby improving the performance significantly. Further, this makes it easier to provide tight timing guarantees on the computation and communication tasks that are performed by the applications running on the platform. Chapter 3 presented a strategy to design CA-based MPSoC platforms which can satisfy the throughput constraints of multiple applications. The next step is to synthesize these platforms onto hardware. FPGAs are platforms of choice over ASICs (for low product volumes) due to low non-recurring engineering costs. Even for designs with large volumes, FPGAs are used for prototyping before implementing the functionality onto ASICs. However, it is very time consuming to map applications on these platforms due to the unavailability of *platform generation tools*. Furthermore, it is very difficult to *program* them as the user has to configure the communication infrastructure in addition to the application functionality.

Manual design efforts are error prone and consume a lot of time. FPGA ven-

dors have provided some intermediate languages which transform high level platform descriptions into Hardware Descriptive Languages (HDLs) [Xil11b]. However, the user has to add each component manually which slows the design process. Additionally the software for each processing element has to be written manually which makes the whole process very cumbersome. To worsen matters, most of these devices have a very short product life-cycle; shorter time-to-market for these systems poses a challenge for the designers. The designers have to verify each use-case. For example, Bluetooth 2.5 has to meet its specification during each combination of applications. It should perform while receiving a call or sending text messages or even taking a picture. So there is a need for automated tools which can reduce the design generation and prototyping time.

There are some multi-processor design tools like Compaan [SZT+04] and [JSKR05, LYBJ01, NSD06], but most of them lack support even for multiple applications let alone multiple use-cases, and require manual steps. There is a tool described in [KFH+08] that supports platform generation for multiple applications and their use-cases but it does not support *CA-based platforms*. Automated platform generation reduces errors in the design and thus saves time for design iterations.

In this chapter, we present a design flow (CA-MPSoC) that takes models of multiple applications synthesize CA-based multi-processor platform. The mapping of actors onto processors is determined with the help of our heuristic-based algorithm described in the previous chapter. As far as we know, this is the first design flow which can generate a CA-based platform. Following are the *key contributions* of this chapter.

An automated design flow that generates multi-processor systems, directly from the architecture aware application graphs. The flow also generates the communication infrastructure so that the designer does not have to worry about it. It generates a super-set hardware which can be used for all the use-cases. The software for each use-case is generated individually. This reduces the verification time of all use-cases of the applications.

Another contribution of this work is a definition of an interface for the tasks such that the semantics of SDF behaviour are maintained during execution. So when an application specification includes high-level language code corresponding to tasks in the application, the source code is automatically added to the desired processor.

The software for all the processors is automatically generated in the flow. Further, the required communication APIs is also generated. This includes configuration of communication channels, setting up connections, and management of memory used for communication. The programmer does not bother about these configurations and can concentrate on the functionality of the applications. A similar interface for non-CA based platforms has been presented in [Kum09]. We have extended this interface to incorporate CA and enable automatic CA-based platform generation.

The above contributions are essential to further research in design automation community since the embedded devices are increasingly becoming multi-featured.

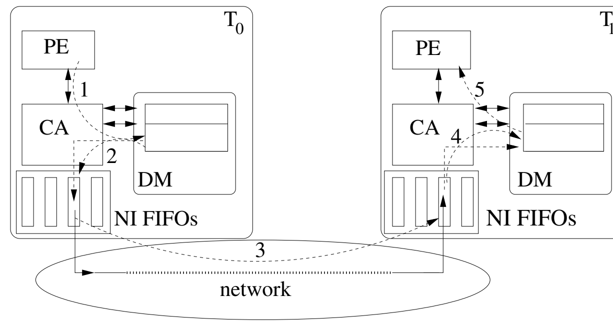


Figure 4.1: Proposed CA-based platform.

Our flow allows designers to synthesize an MPSoC platform that can satisfy the throughput constraints of multiple applications in given use-cases. In this chapter, platform generation for multiple uses-cases is evaluated with a mobile phone case study consisting of 6 applications. The merging of use-cases gives a platform which supports all the use-cases. The tool is made available on-line [CM09] for the benefit of the research community.

This chapter is organized as follows. Section 4.1 revisits the architecture template used in this thesis. Section 4.2 gives details of the steps performed in our design flow to generate the platform. Section 4.3 details the tool implementation. Section 4.4 presents results of the experiments performed to evaluate our design flow. Section 4.5 reviews the related work for automatic platform generation tool flows. Finally, Section 4.6 concludes the chapter and gives directions for the future work.

4.1 Architecture Template

The architecture template used in our platform is depicted in Figure 4.1. It consists of a processing element (PE), a communication assist (CA), Data memory (DM) and Network interface FIFOs (NI FIFO). The CA transfers data between the DM and the NI FIFO. The NI FIFOs are connected through a partial point-to-point network. The structure of the network is out of the scope of this chapter.

Scalability of partial point-to-point networks has been an issue as they require storage to deal with bursts. FSL bus from Xilinx is one example. However, the point-to-point networks used in our template do not require storage. This means that cost of a connection is not very high. The CAs can transfer the data directly from the data memory of the sending tile to the data memory of the receiving tile, i.e. they do not require storage in the point-to-point network itself.

4.1.1 Processing Element

The processing elements used in our template are simple RISC based processors. RISC processors are the processing element of choice for tile-based platforms [TKM+02]. No caches are attached to the processor to have predictable execution traces. The PE has local instruction and data memories. The instruction memory is connected to the PE through a bus whereas the access to the data memory is through the communication assist. Note that we chose Microblaze processors from Xilinx. Our synthesis flow is not restricted to any one processor type so choice of processor is not important.

The PE is non-pre-emptive and can execute only single thread. This simplifies the architecture of the PE. Preemption requires extra hardware and is costly in terms of area. Furthermore, non-pre-emptive scheduling algorithms are easier to implement as compared to their pre-emptive counter parts and have dramatically lower overhead at run-time [JSM91]. In high performance embedded processors (like SPEs in Cell Broad Band Engine and graphics processors), non-pre-emptive systems are preferred over pre-emptive systems.

4.1.2 Memories

We use a single-ported instruction memory, which is directly connected to the PE. The data memory (DM) used in our template is a dual-port memory as depicted in Figure 4.1. The CA has exclusive access to one port of this memory. The second port is connected to the PE through the CA. The choice of dual-port memory may seem expensive, however we use it to make the access of the memory to CA and PE as fast as possible. Single ported memory can introduce stall cycles for the processor which in-turn makes the execution time of the task executing on the processor unpredictable. Furthermore, many FPGAs contain dual-ported memory blocks.

4.1.3 Communication Assist

Our communication assist acts as an interface that provides a link between the NoC and the sub systems (PE and memory). It also acts as memory management unit that helps a processor keep track of its data structures. As a result, it decouples communication from computation and relieves the processor from data transfer functions. Our programmable CA uses a shared data and buffer memory. This leads to lower memory requirement for the overall system and to lower communication latency.

4.2 Design Flow

The previous chapter was devoted to our heuristic algorithm that generates the mappings of actors from multiple applications to meet the throughput constraints.

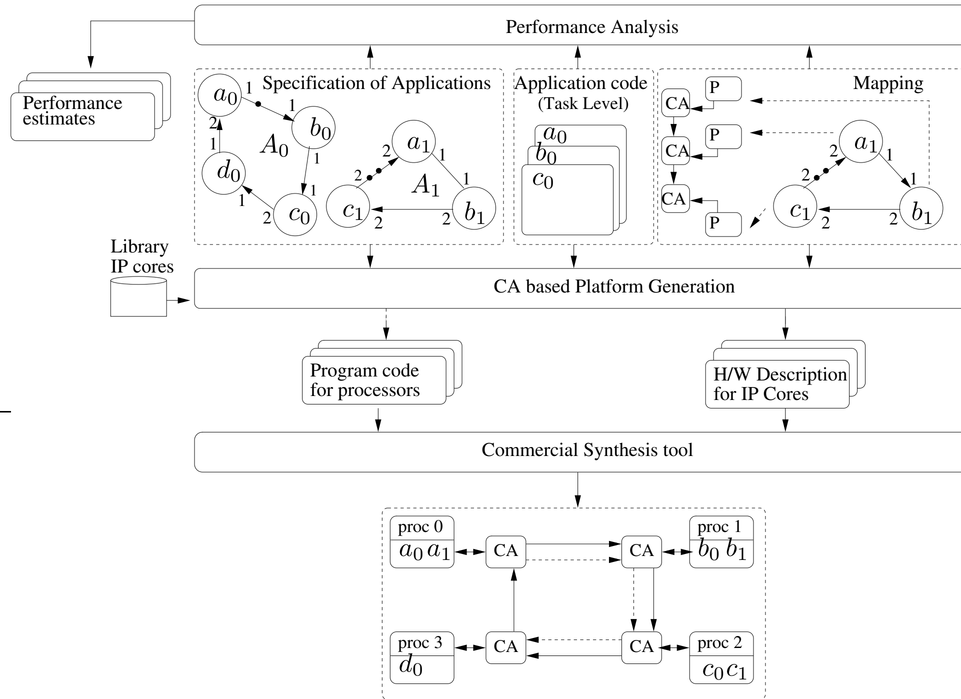


Figure 4.2: Design flow to generate MPSoC platforms using commercial FPGA tools.

```
<sdf name="jpeg" type="G">
  <actor name="CC" type="A0">
    <port name="in0" type="in" rate="128" datatype="char"/>
    <port name="out0" type="out" rate="64" datatype="char"/>
    <executionTime time="4446"/>
    <processor type="proc_0" default="true">
    <functionName funcname="CC"/>
  </actor>
  <actor name="DCT" type="A1">
   <port name="in0" type="in" rate="64" datatype="char"/>
   <port name="out0" type="out" rate="64"datatype="short"/>
   <executionTime time="20950"/>
   <processor type="proc_1" default="true">
   <functionName funcname="DCT"/>
  </actor>
   ...
<channel name="ch0" srcActor="CC" srcPort="out0"
dstActor="DCT" dstPort="in0"/>
   ...
```

Figure 4.3: Snippet of JPEG application specification.

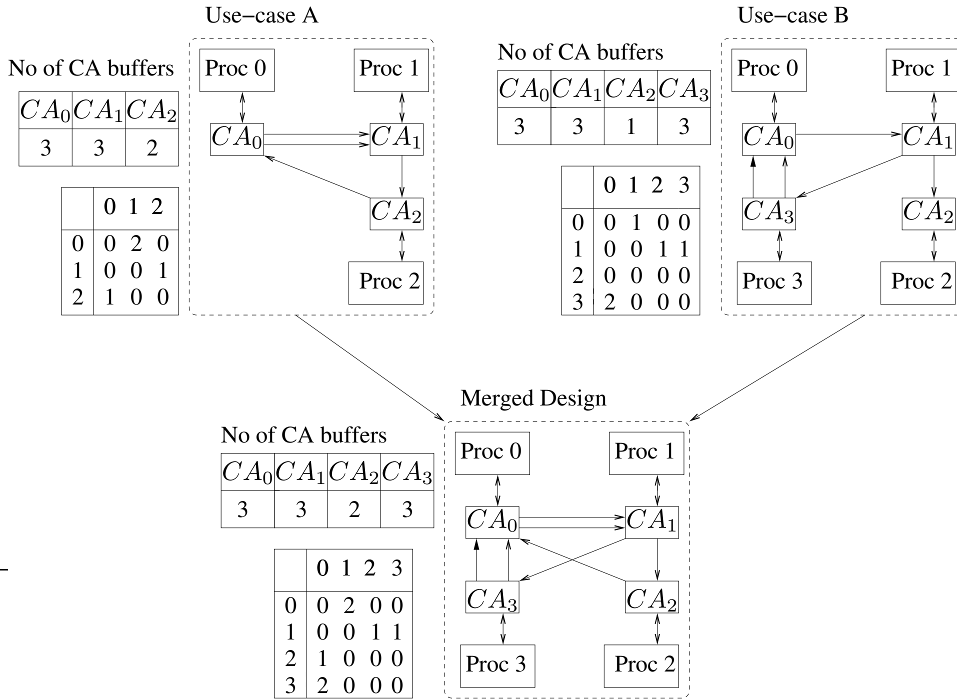
Once the user is satisfied with the performance analysis results, he/she can generate the complete CA-based platform using our design flow. We present CA-MPSoC, a design flow that takes in application(s) specifications and generates the entire CA-based MPSoC, specific to the input application(s) together with corresponding software projects for automated synthesis. This allows the design to be directly implemented on the target architecture. Figure 4.2 depicts our system design methodology. The application-descriptions are specified in the form of SDFGs, which are used to generate the hardware topology. Figure 4.3 shows an example application description. It forms an important part of the flow. While the specification shown in Figure 4.3 is obtained through application profiling, it is also possible to use tools to obtain the SDF description for an application from its code directly. Compaan [SZT+04] is one such example that converts a sequential description of an application into concurrent tasks. These can be then converted into SDFGs easily.

The application-descriptions, mapping information (actor-to-processor), and source code of each application are input to our tool. The source code is already partitioned, and each actor is in the form of a function call with arguments being the input and output to the actor.

4.2.1 H/W Generation

During hardware generation, the IP cores of the processor, CA, and memories are connected according to the mapping information. A CA is connected with each processor to take care of the communication between the processors. The number of NI FIFO channels and the number of buffers (the CA has to manage) are also generated according to edges in the architecture aware SDF graphs.

As the generated hardware supports multiple use-cases, so we employ the use-case merging technique [KFH+08] and modify certain parts to incorporate CA



CA_0	CA_1	CA_2
3	3	2

	0	1	2
0	0	2	0
1	0	0	1
2	1	0	0

CA_0	CA_1	CA_2	CA_3
3	3	1	3

	0	1	2	3
0	0	1	0	0
1	0	0	1	1
2	0	0	0	0
3	2	0	0	0

CA_0	CA_1	CA_2	CA_3
3	3	2	3

	0	1	2	3
0	0	2	0	0
1	0	0	1	1
2	1	0	0	0
3	2	0	0	0

Figure 4.4: An example showing how the combined hardware for different use-cases is generated. The corresponding communication matrix and number of buffers are also shown for each hardware design.

buffers. Each use-case requires a certain hardware topology to be generated. In addition to that, software is generated for each processor. Figure 4.4 shows an example of two use-cases that are merged. The figure shows two use-cases A and B, with different hardware requirements that are merged to generate the design with minimal hardware requirements to support both. The combined hardware design is a super-set of all the required resources such that all the use-cases can be supported. The reason to use a super-set hardware is the fact that while multiple applications are active concurrently in a given use-case, different use-cases are active exclusively.

The algorithm to obtain the minimal hardware to support all use-cases is described in Algorithm 5. The algorithm iterates over all use-cases to compute their individual resource requirements. This is, in turn, computed by using the estimates from the application requirements. While the number of processors and CA buffers needed are updated with a max operation (line 10 and line 11 in Algorithm 5), the number of CA channels is added for each application (indicated by line 13 in Algorithm 5). The total CA channel requirement of each application

is computed by iterating over all the buffers and adding a unique edge in the communication matrix for them. The communication matrix for the respective use-cases is also shown in Figure 4.4.

While there are in total three CA channels between CA_0 and CA_1, only two are used (at most) at the same time. Therefore, in the final design only two CA channels are between them. The number of CA buffers required is the maximum needed for all the use-cases. For example, CA_2 requires 2 buffers for use-case A and one in use-case B however, in the super-set hardware two buffers are reserved for CA_2. Note that the CA can use the same buffer as input or output. The configuration of CA binds a buffer in the memory with a NI FIFO channel. There are limits to the number of use-cases that can be mapped to hardware and to obey these limits certain heuristics have been proposed in [KFH$^+$08].

Algorithm 5 Generate communication matrix for CA channels and number of CA buffers.

Input: Applications $A_i \in \mathbb{G}$ and U_k use-case information i.e. active applications in a use-case
Output: N_{proc} Total number of processors needed
Output: X_{ij} the total number of CA channels needed
1: // Let X_{ij} denote the number of CA channels needed for processor P_i to P_j overall // SDF model of application A_i.
2: $X_{ij} = 0$ // Initialize the communication matrix to 0
3: $N_{proc} = 0$ // Initialize number of processors 0
4: $N_{ca-buffers} = 0$ // Initialize number of CA buffers 0
5: **for all** Use-cases U_k **do**
6: Y_{ij} // Y_{ij} stores the number of CA channels needed for U_k
7: $N_{proc,UseCase} = 0$ // Initialize processor count for use-case to 0
8: $N_{ca-buffers,UseCase} = 0$ // /Initialize CA buffers for use-case to 0
9: **for all** Applications A_l **do**
10: $N_{proc,UseCase} = \max(N_{proc,UseCase}, N_{proc,A_l})$ // Update processor count for U_k
11: $N_{ca-buffers,UseCase} = \max(N_{ca-buffers,UseCase}, N_{ca-buffers,A_l})$// Update CA buffers count for U_k
12: **for all** Channels c in A_l **do**
13: $Y_{c_{src}c_{dst}} = Y_{c_{src}c_{dst}} + 1$ // increment CA channel count
14: **end for**
15: **end for**
16: $N_{proc} = \max(N_{proc}, N_{proc,UseCase})$ // Update overall processor count
17: $N_{ca-buffers} = \max(N_{ca-buffers}, N_{ca-buffers,UseCase})$ // Update overall CA buffer count
18: **for all** i and j **do**
19: $X_{ij} = \max(X_{ij}, Y_{ij})$
20: **end for**
21: **end for**// $N_{ca-buffers}$ is now the total number of CA buffers needed

4.2.2 S/W Generation

Software generation includes configuration of buffers between the actors, data type declarations of the ports of the actors and code needed for SDF actor execution. The software project for each core is produced and the task files are copied into

```
// Functional definition of an SDF-actor
void <functionName>(datatype *in0, datatype *in1,
                    ....., datatype *inN,datatype *out0,
                    datatype *out1,....., datatype *outM){
...
...
}
```

Figure 4.5: The interface for specifying functional description of SDF-actors.

```
-------------------------------------------------------
//Code generated for Color conversion task
-------------------------------------------------------
char* in0;int size_out=rate_out*sizeof(char);
char* out0;int size_in=rate_in*sizeof(char);

Config(buffer_id0,base_addr_out,size_out,out,ni_fifo_id_out);
Config(buffer_id1,base_addr_in,size_in,in,ni_fifo_id_in);

out0=claimwritespace(buffer_id_0,size_out);
in0=claimreadspace(buffer_id_1,size_in);
CC(in0,out0);
releasewritespace(buffer_id0);
releasereadspace(buffer_id1);
-----------------------------------------------------------
//Code generated for DCT task
-----------------------------------------------------------
char* out0;int size_out=rate_out*sizeof(short);
char* in0;int size_in=rate_in*sizeof(char);

Config(buffer_id0,base_addr_out,size_out,out,ni_fifo_id_out);
Config(buffer_id1,base_addr_in,size_in,in,ni_fifo_id_in);

out0=claimwritespace(buffer_id0,size_out);
in0=claimreadspace(buffer_id1,size_in);
DCT(in0,out0);
releasewritespace(buffer_id0);
releasereadspace(buffer_id1);
```

Figure 4.6: Snippet of c-code generated from the architecture aware SDFG of the JPEG encoder.

the project folder. The *xml* file also specifies the processor on which the actor has been mapped. If an application specification also includes high-level language code corresponding to actors in the application, this source code can be automatically added to the desired processor. To realize this, we have defined an interface such that the SDF behaviour is maintained during execution. The number of input parameters of an actor function is equal to the number of incoming edges and the number of output parameters is equal to the number of output edges. The interface is shown in Figure 4.5. The array $*in_i$ is for input tokens consumed from i-th incoming edge. The array size is equal to the size of buffer associated with

the edge. Similarly, $*out_i$ is an array of output tokens that are written during one execution of the actor. The application *xml* file indicates the function name that corresponds to the application actor.

Figure 4.3 shows an example for the DCT actor of JPEG encoder application. The function has an input channel from the CC module and the data produced during execution is written to the output channel to VLC module. Therefore the function definition of this actor only has one input and one output parameter as shown in Figure 4.6.

Figure 4.6 shows the automatically generated c-code from our tool. Both actors are executing on different processors. The data types specified in the *xml* file are used to determine the buffer space needed for the particular buffer. Buffers are configured for each channel. The size for each buffer inside the data memory is determined by multiplying the data type and rate associated with the port. For example, the size of output buffer in CC task is 64 bytes ($64 \times 1(tokensize)$). Configuration of buffer also includes the direction of the buffer, the NI_FIFO_ID number, and the physical address of the buffer inside the memory.

The *claimwritespace* command looks for available space in the output buffer. Similarly the *claimreadspace* checks whether the required number of tokens are available for processing. The buffers are identified by their *ids*. The reason to check the availability of output space before the input space is because our SDF model of execution is conservative. Both commands are non-blocking. So an actor might not be able to execute if any of its incoming buffers does not have sufficient tokens. The same holds when the output buffers of an actor are full. While this does not cause any problem when only one actor is mapped on the processor, in the case of multiple actors, the other possibly ready actors might not be able to execute while processor sits idle. To avoid this, *claimreadspace and claimwritespace* commands have been implemented as non-blocking so that if any of *claimspace* commands is unsuccessful, the processor is not blocked. This is also consistent with our MPSoC design algorithm (described in the previous chapter). The actors belonging to the same application are executed in static order. If an actor from one application is not ready to execute then an actor from another application is scheduled for execution and after finishing this execution, the processor again checks the same actor which was not executed previously. Note that the command overhead is fixed and is added to the execution time of the actors. It is implementation dependent and we explain more about it in Section 4.4.

After the function processing, the *releasewritespace* command indicates the CA to transfer the data to the receiving actor. The release commands update the read/write buffers so that they can be used for further receive/send operations.

4.3 Tool Implementation

Our tool flow targets Xilinx FPGA architecture. It can be easily modified to support other FPGA vendors. The processors in the CA-MPSoC are mapped to Microblaze processors [Xil11b]. The communication links are mapped onto fast simplex links (FSL). These are unidirectional point-to-point communication channels used to perform fast communication. The FSL depth is set to one as this is the minimum depth available for these buses. As explained earlier, we do not require any storage in the point-to-point networks in our proposed design. However, it is not possible to have FSL links with zero storage so it is an implementation dependent restriction.

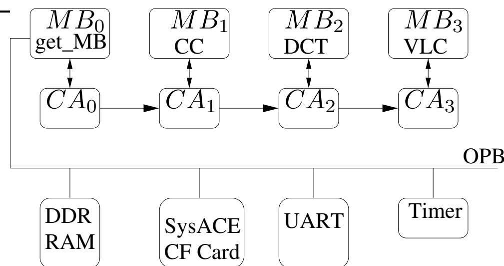


Figure 4.7: Generated hardware from the example xml file.

The generated architecture for the JPEG application is shown in Figure 4.7 according to the specification in Figure 4.3. It consists of several Microblaze processors with each actor mapped to a unique processor, with additional peripherals such as Timer, UART, SysACE, and DDR RAM. While the UART is useful for debugging the system, the SysACE compact flash card allows for convenient performance evaluation for multiple use-cases by running continuously without external user interaction. The timer module and DDR RAM are used for profiling the application and for external memory access, respectively.

In our tool, in addition to the hardware topology, the corresponding software for each processing core is also generated automatically. Routines for measuring performance, as well as sending results to the serial port and CF card on-board are also generated for MB_0.

Our software generation ensures that the tokens are read from (and written to) the appropriate FSL link in order to maintain progress and to ensure correct functionality. Writing data to the wrong link can easily throw the system in deadlock. XPS project files are also automatically generated to provide the necessary interface between hardware and software components.

Table 4.1: Realistic use-cases for mobile phone case study.

usecase number	H263 decoder	H263 encoder	JPEG decoder	modem	Phone call	mp3 decoder
1	1	-	-	-	-	-
2	-	1	-	-	-	-
3	1	1	-	-	-	-
4	-	-	1	-	-	-
5	-	-	-	1	-	-
6	1	-	-	1	-	-
7	-	1	-	1	-	-
8	1	1	-	1	-	-
9	-	-	1	1	-	-
10	-	-	-	-	1	-
11	1	-	-	-	1	-
12	-	1	-	-	1	-
13	1	1	-	-	1	-
14	-	-	1	-	1	-
15	-	-	-	1	1	-
16	1	-	-	1	1	-
17	-	1	-	1	1	-
18	1	1	-	1	1	-
19	-	-	1	1	1	-
20	-	-	-	-	-	1
21	-	-	1	-	-	1
22	-	-	-	1	-	1
23	-	-	1	1	-	1

4.4 Experiments and Results

Our tool is evaluated with a mobile phone case study consisting of 6 applications. In each use-case we enable a subset of these applications. We also show how our tool generates a super-set hardware that supports a large number of use-cases. The software for each use-case is generated at run-time, and enables us to verify these use-cases in a very short time.

Support for Multiple Use-cases & use-case merging

In this case study [Kum09], we consider 6 applications - video encoding (H.263) [Hoe05], video decoding [SGB06b], JPEG decoding [dK02], mp3 decoding [SGB06b], modem [BML99] and regular call. We first construct all possible use-cases giving 63 use-cases in total. However, some of these use-cases are not realistic. For example, JPEG decoding is unlikely to run simultaneously with video encoding or decoding, because when a user is recording or viewing video, it is not possible to browse through pictures. Similarly it is also not possible to listen to mp3 songs while talking to some body on phone. This gives us 23 realistic use-cases as shown in Table 4.1. Each active application in a use-case is represented with a “1” at its position.

In this experiment, our tool generates a platform that can support all of these 23 realistic use-cases. The platform consists of 5 Microblaze processors and 5 communication assists. The platform occupies 97% of the available FPGA resources.

Our approach is very fast and is further optimized by modifying only the relevant software and keeping the same hardware design for different use-cases. The software synthesis includes configuration of all CA channels, buffer sizes, and incorporation of appropriate task calls. Since software synthesis step takes only about 25 sec in our experiment, the entire experiment for 23 design points takes only about 9 minutes.

Manual design effort will involve separate hardware generation and software configuration for each use-case. In contrast, our tool takes a mere 100 ms to generate the complete hardware design. The Xilinx tool takes about 36 minute to generate the bit file together with the appropriate instruction and data memories for each core in the design. The time spent on the exploration is an important aspect when estimating the performance of big designs. The 6 application system is also designed by hand to estimate the time gained by using our tool. The hardware and software development took about 4 days in total to obtain an operational system.

Table 4.2: Time taken for platform generation in the experiment with ten applications.

	Manual Design	Generating Single Design hr:m:sec	Complete Experiment hr:m:sec
Hardware Generation	2 days	40ms	40ms
Software Generation	2 days	60ms	60ms
Hardware Synthesis	0:36:00	0:36:00	0:36:00
Software Synthesis	0:0:25	0:0:25	0:09:34
Total Time	4 days	0:36:25	0:45:34
Iterations	1	1	23
Average Time	4 days	0:36:25	0:1:59
Speedup	-	1	18.36

This hardware/software co-design approach results in a speed-up of about 18 when compared to generating a new hardware for each iteration. As the number of design points are increased, the cost of generating the hardware becomes negligible and each iteration takes only about 25 seconds. This study shows the usefulness of our use-case merging approach for problems like DSE for multi-processor systems.

4.5 Related Work

The problem of mapping an application to architecture has been widely studied in literature. One of the recent works most related to our research is ESPAM [NSD06]. This uses Kahn process networks (KPNs) [Kah74] for application specification. In our approach, we use SDFGs for application specification instead. Further, our approach supports mapping of multiple applications, while ESPAM is limited to a single application. This difference is imperative for developing modern embedded systems which support more than tens of applications on a single MPSoC. The same difference can be seen between our approach and the one proposed in [JSKR05], where an exploration framework to build efficient FPGA multi-processors is proposed.

The Compaan/Laura design flow presented in [SZT+04] also uses KPN specification for mapping applications to FPGAs. However, their approach is limited to a processor and coprocessor. Our approach aims at synthesizing complete MPSoC designs supporting multiple processors. Another approach for generating application-specific MPSoC architectures is presented in [LYBJ01]. However, most of the steps in their approach are done manually. Exploring multiple design iterations is therefore not feasible. In our flow, the entire flow is automated, including the generation of the final bit-file that runs on the FPGA. Yet another flow for generating MPSoCs for FPGAs has been presented in [KHHC07]. However, that flow focuses on generic MPSoCs and not on application-specific architectures. There is also a tool described in [KFH+08] that supports platform generation for multiple use-cases but it does not support *CA-based platforms*.

Xilinx provides a tool-chain as well to generate designs with multiple processors and peripherals [Xil11b]. However, most of the features are limited to designs with a bus-based processor-coprocessor pair with shared memory. It is very time consuming and error prone to generate MPSoC architecture and the corresponding software projects to run on the system. In our flow, MPSoC architecture is automatically generated together with the respective software projects for each core.

In [CCK+08], the authors present a design flow that generates a multi-core system for multimedia applications. Their work is quite similar to ours. However, there are some key differences. Firstly they use a mesh network for interconnection whereas we use point-to-point networks. Secondly, they use profiling to dimension their system. We on the other hand use static analysis techniques. Profiling based techniques are significantly slower than analysis based techniques. Also their synthesis flow generates platforms for average case performance whereas our flow can generate platforms for both worst case and average case performance. Lastly, our flow supports multiple applications concurrently executing on the platform while [CCK+08] is for single application only.

Finally, none of the above flows support a CA-based platform. In fact our flow is the first to generate CA-based multi-processor platforms. Communication plays an important role in the parallelization of applications. The communication

to computation ratio determines the justification of splitting task between the processors. Our CA in turn exposes more parallelism in the applications. We have seen that if large volumes of data needs to be transferred between communicating actors, our CA can speedup the transfer resulting in overall application speedup.

4.6 Conclusions

In this chapter, we presented a design flow to generate multi-processor platforms for multiple applications. We also provided analysis techniques to predict the performance of the applications before the generation of the platform. The design flow can cater for hard real-time applications. It is largely automated and requires minimal manual effort. It also generates the configuration software for the communication infrastructure and processing elements.

The design flow is evaluated with 6 applications from a mobile phone case study. The automated platform generation takes milliseconds in contrast to days needed for manual platform generation. The use-case merging evaluates all the 23 realistic use-cases of the case-study by using a single hardware platform. This results in a speed up of 18 when compared to the case where hardware for each use-case is generated individually and then evaluated. The tool is made available on line [CM09] for use by the research community.

Once the MPSoC platform has been synthesized into an FPGA, a run-time resource manager is required to deal with the dynamic situations. In the next chapter, we present run-time resource manager which can be used to handle dynamism in the applications mapped onto the MPSoC platform. The main focus is on scalability of this resource manager with increasing number of applications and processing elements.

CHAPTER 5

Distributed Resource Management

Modern multimedia systems for the consumer market are increasingly based on multi-processors due to stringent performance, power and cost constraints. These MPSoCs typically use (massive scale) instruction-, data- and task-level parallelism to compensate for a lower clock frequency in order to consume less energy while satisfying high compute requirements. These MPSoCs execute multiple applications concurrently that may exhibit dynamic behaviour. For example, MPSoC platforms are synthesized assuming worst-case task execution times. In reality, the execution times of the tasks may be less than their worst-case estimates. The designer cannot anticipate this variation in execution time at design time and run-time mechanisms are needed. Additionally, applications can be started or stopped by the user at run-time. Note that if the newly added application was not analyzed at design time then it is a new use-case and the resource managers discussed in this thesis cannot guarantee the performance of applications in such use-cases.

A Resource Manager (RM) is used to manage the resources of an MPSoC platform. The resource manager is needed to make trade-offs between the quality levels and compute requirements of the various applications, yielding even more dynamic application behaviour. It is also possible that the user wants to start an application at a certain quality level and the platform does not have sufficient resources to run the application at that quality-of-service (QoS) then the resource manager should be able to run the application at a lower QoS level provided the user agrees with it. The basic requirement to the RM is that all admitted concurrently executing applications must attain their specified performance constraints.

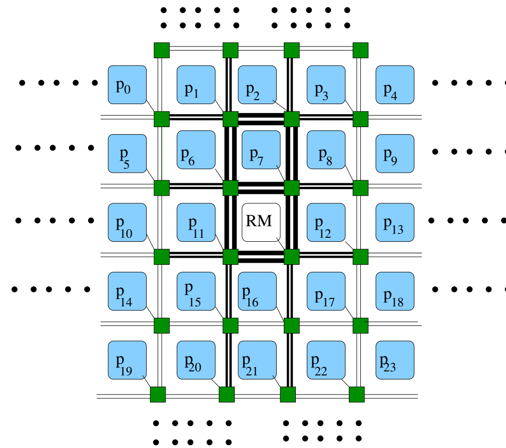


Figure 5.1: Management and communication bottleneck for centralized RM

The RM should also be able to handle dynamic conditions like admission of a new application, change in application performance constraints, etc.

Theoretically, compile-time analysis of all possible use-cases can provide performance guarantees, but the potentially large number of use-cases in a real system makes such a static analysis impractical [KMT+08], while not dealing with other types of dynamic behaviour, like data-dependent execution times (e.g. object-based vision processing). We therefore need to shift the burden from compile time analysis to run-time monitoring and intervention when necessary. This makes run-time resource management an essential part of MPSoCs.

A major requirement for run-time management approaches for these MPSoCs is that they no longer can assume pre-emption of the hardware platform; the (massively) parallel processing of multi-media data yields such a large amount of state in the processors, that it is no longer cost-effective to implement a context switch. Even if a context switch would be implemented, the time required to perform the context switch cannot be ignored, which excludes commonly used real-time scheduling approaches like Rate-Monotonic Scheduling, and other fixed-priority schedulers. The latest high-performance media MPSoCs for the consumer market, like the Cell BBE [CHKW08] and graphics processors, do not support pre-emption, thereby acknowledging the necessity of non-pre-emptive systems.

A resource manager has been presented in the literature [KMT+08] that performs management in a *central* way. It monitors the progress of applications, and enables and disables the applications at the smallest possible grain, i.e. individual actors (explained in more detail in Section 5.1). In this way, the central resource manager helps applications satisfy their throughput constraints. However, a central RM is not scalable with the number of applications and processors.

A centralized RM may create a hot spot in terms of management as shown in

Figure 5.1. All the information regarding application progress and QoS is fed to the centralized manager. The centralized manager has to process the information in a short time and take corrective measures. The compute requirement from the applications overwhelms the centralized manager.

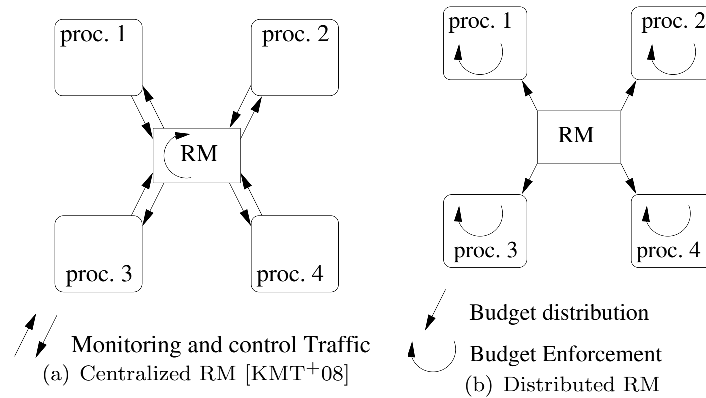
In this chapter, we propose two versions of resource managers which are scalable with the number of applications and processors. Our resource managers ensure that an application is only allowed to start if the platform can allocate the resources demanded by the application. Our distributed RMs are based on budget-based schedulers and differ in their budget enforcement protocols. The first type of RM– *Credit-based* can be used for applications which have strict constraints on their performance, i.e. their performance cannot be more than a fixed level even if resources are available to have better performance. Our second type of RM– *Rate-based* is suitable for applications which may run at higher performance than a minimum level if the compute resources are available. For example, streaming encoders are a good target for these type of RMs so that if there are resources available in the MPSoC platform, they can encode at a higher rate and finish the job quickly.

In this chapter, the distributed resource managers have been evaluated on the basis of their ability to satisfy the throughput constraints of multiple applications. We have also performed experiments by adding applications at run-time and studying their behaviour. The evaluation metrics used in this work include *deadline misses*, *buffer requirement*, and maximum *jitter*. A deadline miss occurs when an application fails to meet its deadline. The difference in successive finish times of application iteration should be fixed. Due to interference with other applications, the difference may not be constant. This variation in successive finish times is defined as the jitter. Other aspects like processor utilization and run-time variation in application throughput constraint have also been studied in this chapter.

The chapter is organized into the following sections. Section 5.1 describes our architecture model. Section 5.2 presents an example showing the scalability issues with centralized RM. In Section 5.3, we present our credit-based RM and rate-based RM. The evaluation and comparison of these RMs is presented in Section 5.4. Section 5.5 presents related work in this field and Section 5.6 concludes this chapter.

5.1 Application and Architecture Modelling

The resource managers presented in this chapter assume that the applications are represented as SDFGs. The architecture model consists of processors connected with each other through point-to-point/NoC interconnects. Our resource managers include an admission controller. The role of the admission controller is to evaluate the timing constraints of the new applications against available resources. If the available resources of the platform are less than the timing requirement of



(a) Centralized RM [KMT+08]

(b) Distributed RM

Figure 5.2: Resource Management on MPSoC: Centralized vs Distributed

the new application, the admission controller rejects the request and the application can request service at a lower quality level. A similar admission controller has been presented in [KMT+08]. To find the budgets, following information is required by the admission controller.

- SDF model of each application.
- Worst-case execution time estimates for each actor (in clock cycles).
- Desired performance of each application, e.g. frames/sec.
- Mapping of actors onto the platform is provided. Task migration is not supported, so the mapping remains fixed across all use-cases.
- Buffer sizes needed for edges in the graphs.
- Performance prediction of each application in isolation with the given mapping. This can be achieved using SDF^3 [SGB06a].

Figure 5.2 shows the functional diagram of the centralized resource manager presented by [KMT+08] and our distributed RM. The RM presented by [KMT+08] monitors the throughput of each application and compares it with its desired throughput. The centralized RM enables an application performing less than its desired throughput. Similarly the application having more than its desired throughput is suspended. The monitoring and control overhead limits the scalability of the centralized RM as it has to monitor and control all applications and perform QoS negotiations as well. To solve this problem, we present two versions of distributed RMs. These RMs seek to minimize the involvement of the central manager in the process and invest more intelligence in the local processor arbiters as shown in Figure 5.2(b). The budgets for each application are calculated centrally but they are enforced on all the processors locally. The RM

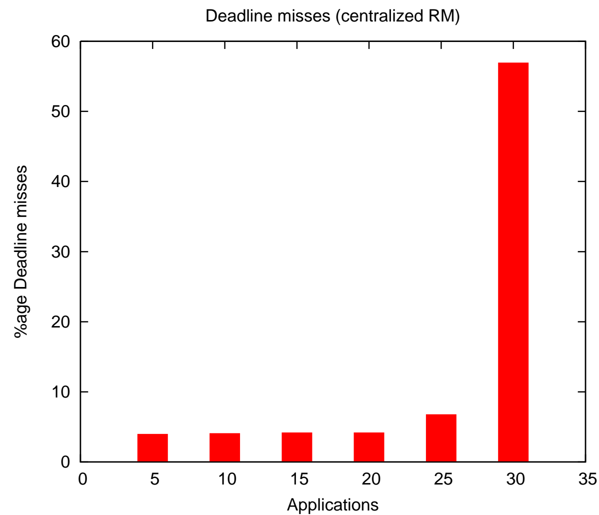


Figure 5.3: Increase in deadline misses with increase in number of applications

does not monitor each application so the scalability problem of [KMT+08] due to monitoring period is eliminated. In the next section, we present an example to further elaborate this problem.

5.2 Motivating Example

The central resource manager monitors the performance of each application. The monitoring period of central RM should be less than or equal to smallest period amongst all applications being monitored otherwise it will not be able to monitor the variations in that application. To study the scalability of central RM with increase in number of applications we use the model of central resource manager as described in [KMT+08]. The computation platform has 10 processors. The central resource manager performs the following operations for monitoring the performance of the applications:

- Receive the iteration completion message from each application.
- Increment the total execution count of the application.
- Find the current period of the application.
- Compare the current period with the desired period.
- Send enable/suspend signal to the application according to the result of the comparison.

We implemented the above functionality on a Microblaze processor [Xil11a] and measured the clock cycles required for one application. It took 240 clock cycles per application to perform above mentioned functions. We modelled our RMs with timing information. The length of a monitoring period determines the number of applications which can be controlled by the resource manager as the central processor must complete the above mentioned functionality within the monitoring period. Assume we have a monitoring period of 7,000 clock cycles then we can monitor $7000/240 = 29$ applications only. If we increase the number of applications to 30, the applications cannot meet their constraints as shown in Figure 5.3. In this experiment, we increased the number of applications and kept the monitoring period fixed at 7,000 clock cycles. The total number of deadline misses of these applications remains relatively low until 25 applications. For 30 applications, the total number of deadline misses increase tremendously as shown in Figure 5.3. The reason for this increase is the fact that we cannot monitor all 30 applications in the period of 7,000 cycles. During every monitoring cycle, some applications are not monitored thus resulting in deadline misses.

The other problem with centralized RM is jitter in application execution. If the monitoring period is large then an application will remain enabled/disabled during the whole monitoring period resulting in periods where the performance of application is more than desired performance and periods where it is less than the desired performance. To handle this jittery behaviour, large buffers are required to store the outputs of the applications so that the average behaviour is acceptable. These large buffers increase the cost of the system and are highly undesirable. We have illustrated this effect in Section 5.4.5.

Note that the scalability problem is made even worse when Quality-of-Service (QoS) negotiations are performed in order to deal with scarcity of computational resources. These negotiations burden the resource manager for each application (to set the quality level) as well each processor (monitoring and quality settings). We therefore expect that future MPSoC systems that enable QoS will profit even more from our distributed resource managers, which are optimized for scalability.

We propose two versions of the distributed resource managers that are motivated to address the scalability issues. The first type of resource manager is called *Credit-based* resource manager. Here the central admission controller distributes credits among the processors and they enforce these credits. This type of manager is useful where the throughput of applications has to be kept at a certain level. The second distributed RM is called *Rate-based* resource manager. This RM uses the same admission controller as used by the credit-based manager, but differs in the budget enforcement mechanism. This type of manager is useful for those applications which allow more than a certain level of performance. In the following section we explain both of these RMs in detail. Our resource managers are more scalable with increasing number of application and processors.

5.3 Proposed Resource Managers

Application admission control and budget enforcement are two parts of resource management. We first describe the admission control and credit calculation mechanism. The same mechanism is used in both types of RMs. In both versions of distributed resource managers, there is a central admission controller connected with the processing cores through a NoC. The central admission controller is an interface to the user and calculates the credits and these credits are distributed to the processors. The arbiters at the processors locally enforce these credits such that the throughput constraints of the applications are satisfied.

Algorithm 6 shows the method of calculating the credits. Each processor has a large replenishment interval of time with in which all tasks have to execute. The central controller finds the processing load imposed by each task on each processor. This load is calculated by multiplying the repetition vector of each actor with its execution time and the ratio of desired to predicted throughput. The size of the replenishment interval should be greater than the total processing load. This process is repeated for all the processors and an application is only admitted if all processors satisfy this condition. The credits are calculated as shown in

Algorithm 6 Admission Control and Credit calculation

Input: Applications $A_j \in \mathbb{G}$ and processor $p_i \in P$
Output: Credits($a_k \in \mathbb{G}$)

```
1: for all p_i ∈ P do
2:    load(p_i)=0;
3:    for all A_j ∈ G do
4:       processing_load=0;
5:       for all a_i ∈ G do
6:          if (mapping(A_j,a_k) == p_i) then
7:             processing_load(a_k) = γ(a_k) × α(a_k) × desired_throughput
8:             load(p_i)+=processing_load(a_k)
9:             if (load(p_i)/size_of_replenishment_interval(p_i)>1) then
10:               remove_load(A_j,p_i)// admission not possible, remove all actors of the appli-
                  cation from all processors
11:            end if
12:         end if
13:      end for
14:   end for
15: end for
16: for all applications do
17:    for all actors(application) do
18:       credits(a_k)=γ(a_k) × desired_throughput
19:       send(credits,a_k,p_i)
20:    end for
21: end for
```

line number 17 in Algorithm 6. Here γ(actor) and α(actor) are the repetition vector entry and execution time of the actor, respectively. We explain the credit calculation mechanism with the help of an example. Assume that we want to find the credits for the inverse quantization actor (IQ) from JPEG decoder. The JPEG

decoder graph used for this example decodes one macro-block in a single iteration. The IQ actor has to be called 6 times (4 times for luminance data and 2 times for chrominance). If the desired throughput constraint for JPEG decoder is set at 1 QCIF picture/sec then it is equivalent to 99 macro-blocks of JPEG encoded data and the number of credits for IQ are $6 \times 99 = 594$. Assuming a replenishment interval of one second the processing load imposed by IQ is $6 \times 2400 \times 99 =$ 1,425,600 clock cycles. Here, we assume that 2400 clock cycles are required by the IQ actor to perform inverse quantization function on one macro-block.

5.3.1 Credit-Based Resource Manager

The central admission controller sends the credits to the processors according to the mappings of actors onto processors. To enforce these credits, each processor has a kernel which loads these credits into counters. Each actor is repeated the number of times as specified in its counter in one replenishment interval. After completion of the interval, the counters are reloaded with their values as received by the central controller and this process continues. During the execution, if an actor is not ready then it is skipped and the processor is assigned to another actor so that the processor time can be used more efficiently.

Note that in contrast to TDMA, the replenishment interval of credit-based RM is not necessarily always equal to maximum replenishment interval. It might be possible that some applications are stopped by the user so the length of that replenishment interval will be smaller than the maximum replenishment interval.

Algorithm 7 Credit-based RM

Procedure: Credit_Based_RM()

1: **while** (size_of_replenishment_interval(p_i) $>$ 0) **do**
2:     **if** (a_k == ready $\wedge$ (credits(a_k) $>$ 0)) **then**
3:         execute(a_k)
4:         credits(a_k)- -
5:     **end if**
6:     a_k=next_actor_in_list
7: **end while**

5.3.2 Rate-Based Resource Manager

The dependence of the credit-based RM on the size of replenishment interval can be removed by the Rate-based RM. In the rate-based RM each processor has a local arbiter. The admission controller of the rate-based RM calculates the credits in the same way as the credit-based RM. The admission controller sends these credits to each distributed arbiter located at each processor. This arbiter receives these credits and executes these actors in such a way that the actor having the least achieved-to-desired execution ratio is given the highest priority. Note that

the rate-based RM allows the applications to execute continuously so long as the ratio is maintained. This means that applications can have more throughput than the desired throughput. To implement such an arbiter, a data structure for each actor is defined which contains the information required during the operation of the arbiter. This data structure contains registers to store the desired rates Rd_{a_i} (credits), achieved rates Ra_{a_i} and ratio of achieved-to-desired rates Rr_{a_i} for each actor a_i. The ratios of the achieved-to-desired rates are stored in non-decreasing order. Each time an actor executes, it achieved execution rate is incremented by one.

Algorithm 8 Rate-based distributed Manager

Procedure: Rate_Based_RM()
1: **for all** actors a_k **do**
2: Receive_rates()// Receive rates from admission controller
3: init(Rd_{a_k}, Ra_{a_k}, Rr_{a_k})
4: **end for**
5: **loop**
6: **for all** actors a_k **do**
7: **if** (a_k == ready && Rr_{a_k} == min_rate) **then**
8: Execute the actor
9: Ra_{a_k}++
10: $Rr_{a_k} = Ra_{a_k} / Rd_{a_k}$
11: min_rate = find_min_rate()
12: **end if**
13: **end for**
14: **end loop**

Each time a processor needs to execute an actor it finds the actor having least ratio and executes that actor if it is *ready* as shown in line 11 of algorithm 8. In SDF semantics, an actor is ready when its input data is available and there is space in its output buffer.

We explain the mechanism of rate-based RM by an example. Assume two applications a JPEG decoder and an H.63 decoder are to be executed on the platform. We assume that two actors, the inverse quantization actor (IQ) from JPEG, and the IDCT actor from H.263 decoder, are mapped onto a single processor. The constraint for JPEG decoder is 1 QCIF frame/sec as described in the previous example. We assume that the H.263 decoder has a performance constraint of 20 frames/sec then the credit for IDCT actor is 40. Assume that both actors have been executed twice then the achieved-to-desired ratios of IQ and IDCT are 2/594=0.003 and 2/40=0.05 respectively. The achieved-to-desired ratio of IQ is lower than that of IDCT so the IQ actor has the higher priority to use the processor as compared to IDCT. The platform will execute IQ actor and will update the achieved execution count of IQ (line 9-10 of Algorithm 8) and will calculate the priorities again.

Note that both RMs are very simple so a software/hardware implementation is not expensive in terms of run-time overhead. This makes them suitable for

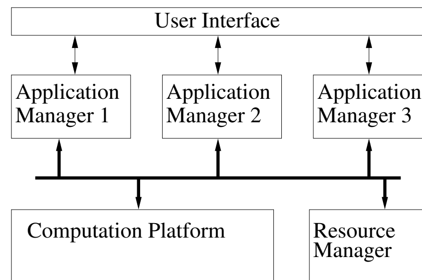


Figure 5.4: Overview of the system setup

run-time resource management of embedded systems.

5.4 Comparison Between the Resource Managers

In this section, we compare the centralized and two distributed resource managers. The bases of comparisons are different kinds of scenarios which are possible during run-time. These scenarios include:

- run-time admission of a new application
- run-time stoppage of an application
- run-time variation in execution time of the actors
- run-time variation in application throughput constraint

Figure 5.4 shows the overview of the experimental setup. It consists of a user interface, local application managers (one for each application), a resource manager, and the computation platform. The user-interface simulates input from and output to the user; for example, in case of mobile phone, input can come from a keypad, while the output can be in the form of screen display or sound. The simulation models of our distributed resource managers have been developed using the modelling language POOSL [vdPV97].

We have used two application models: JPEG decoder and H.263 decoder, as shown in Figure 5.5. The JPEG decoder consists of 6 actors namely, Variable Length Decoding (VLD), Inverse Quantization (IQ), Inverse Zigzag (IZZ), Inverse Discrete Cosine Transform (IDCT), Reorder (Re-order) and Color Conversion (CC). The H.263 decoder has four actors namely Motion Compensation (MC), IDCT, IQ, and VLD. The computation platform consists of 6 processors. The figure also shows the mapping of actors onto processors. The experiments in this section are performed to observe dynamic behaviour of our distributed resource manager. The behaviour of different resource managers is compared in these experiments. We also compare the buffer requirement and processor utilization of these RMs.

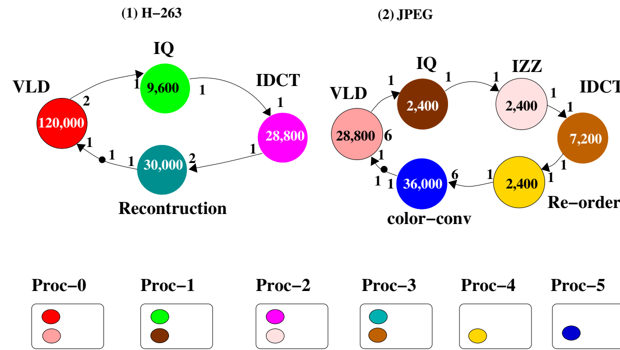


Figure 5.5: Application graphs and their mapping onto a 6-processor computation platform

In the experiments, the monitoring period for centralized RM is 40 million clock cycles. The processors in both credit-based and rate-based RM are executing at 40 MHZ. The replenishment interval for credit-based RM is 1 second so that the RMs are matched.

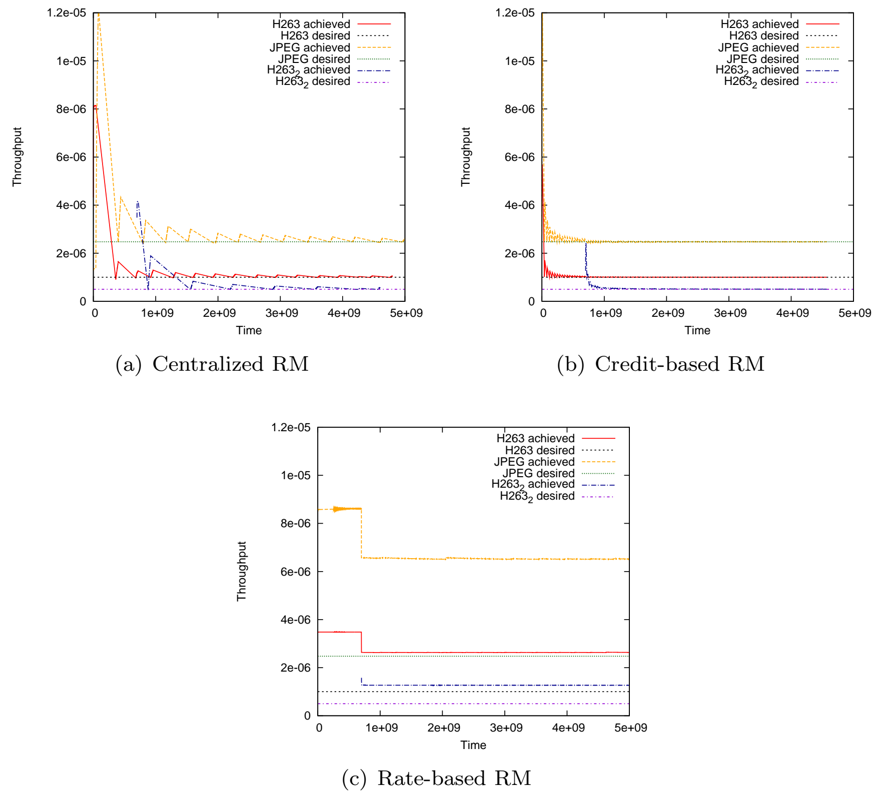
5.4.1 Admission of a New Application

In this experiment, a new application enters the MPSoC platform and requests for resources at (simulation) time 700 million clock cycles. The applications already executing on the platform are a JPEG decoder at 1 QCIF/sec and a H.263 decoder at 40 frames/sec. Another instance of H.263 decoder enters the platform and requires executing at 20 frames/sec.

Figure 5.6(a) shows the behaviour of a centralized RM against this dynamic situation. The centralized RM can handle this situation and the applications already executing do not show any impact on their performance. The jitter in the application execution is significant in case of a centralized RM. The reason for this jitter is the fact that the applications remain enable/disable across the monitoring period. The throughput of applications gets more than the desired throughput in monitoring period where they remain enabled. Similarly the application throughput decreases in the monitoring period where they remain disabled.

Our admission controller evaluates the resources needed for the application according to Algorithm 6 and allows it to execute. Figure 5.6(b) shows robustness of our credit-based RM. The applications already executing on the platform do not have any adverse effect on their performance due to virtualization of the resources. Moreover, the jitter in application execution is very small as compared to the centralized RM.

Figure 5.6(c) shows the response of the rate-based RM. The performance of JPEG and H.263 decoders decrease immediately as soon as the second instance of H.263 decoder is accepted by the RM. This is because of the fact that when



(a) Centralized RM

(b) Credit-based RM

(c) Rate-based RM

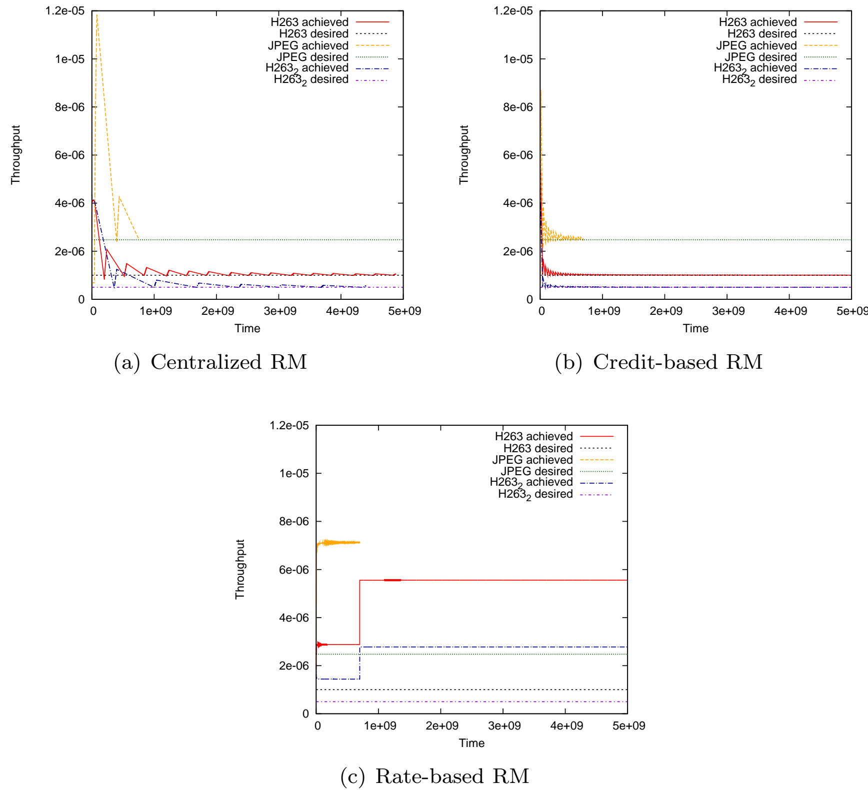
Figure 5.6: Example of run-time application admission at simulation time of 700 million clock cycles

the second instance of H.263 decoder enters, it has the lowest achieved to desired ratio so it gets preference and quickly gains the required performance level. As the time goes by other applications also get the compute resources and the system very quickly converges towards the new steady state.

5.4.2 Application stopped by the User

Run-time dynamism includes stopping of an application by user. This experiment starts with three applications executing on the platform. The JPEG decoder is executing at 1 QCIF/sec and two H.263 decoders are executing at 40 frames/sec and 20 frames/sec respectively. The JPEG decoder is stopped by the user at simulation time of 700 million clock cycles.

The behaviour of centralized RM under this dynamic condition is shown in Figure 5.7(a). The centralized RM can handle this situation and there is no effect



(a) Centralized RM

(b) Credit-based RM

(c) Rate-based RM

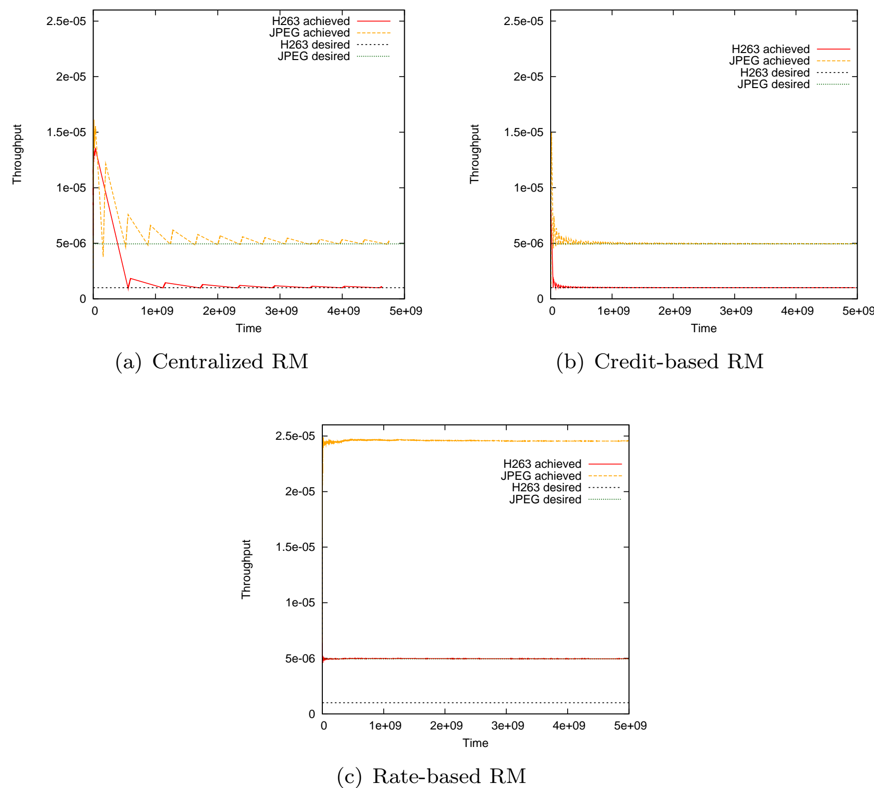
Figure 5.7: JPEG application stopped by the user at 700 million clock cycles

on other applications because of virtualization of resources. Figure 5.7(b) shows the behaviour of our credit-based RM. There is no effect on other applications as our credit-based distributed resource manager provides virtualized platform for each application. The applications execute concurrently but do not interfere with others for the resources.

Figure 5.7(c) depicts the response of our rate-based RM in the event of application stoppage. After the JPEG decoder has stopped, the second instance of H.263 gets the processor more often as its ratio is the lowest. The system goes into steady state and both H.263 decoders share the resources freed by the JPEG decoder. The performance of one of H.263 decoder is twice of the second instance of H.263 decoder and both are above their specified throughput constraints.

5.4.3 Variation in Actor Execution Times

The execution time of actors may vary due to different execution paths through the code. In this experiment, the execution time of the actors is varied randomly between 1 clock cycles to the actual execution time of the actor. We use uniform random number generation for this experiment. Figure 5.8(a) shows that the variation in the execution time has no effect on the throughput for centralized RM. Similarly our credit-based RM has no effect on the throughput of the applications as shown in Figure 5.8(b). The reason for this result is the fact that the credits in our distributed resource managers are calculated as the number of iterations each actor has to execute for satisfying the application throughput constraints. This property makes it independent of the variations in the actor execution times.



(a) Centralized RM

(b) Credit-based RM

(c) Rate-based RM

Figure 5.8: Variation in actor execution times

However, the rate-based distributed RM is slightly affected by the variation in actor execution times. The variation in execution time affects the achieved-

to-desired ratios and consequently the throughput of applications observe some variation but the magnitude of this variation is pretty small as shown in Figure 5.8(c).

5.4.4 Variation in Application Throughput Constraint

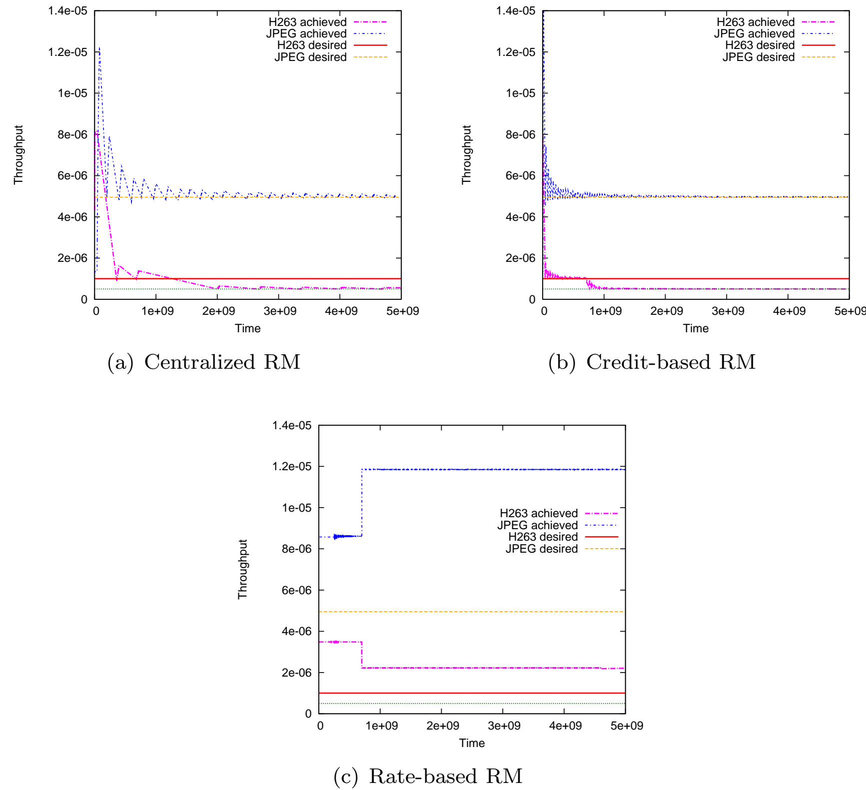
In this experiment, the H.263 decoder is required to decode 40 frames/sec and the constraint for the JPEG encoder is 2 QCIF frames/sec. The distributed credit-based resource manager finds the credits based on this information. The arbiters in the processors enforce these credits as shown in Figure 5.9(b). Now at 700 million clock cycles, the distributed resource manager receives a request from the user to decrease the frame rate of H.263 decoder from 40 frames/sec to 20 frames/sec. This implies that the distributed resource manager has to recalculate the credits based on the new application constraints and it re-sends them to the processors. Figure 5.9(b) shows that the new credits are enforced by the processors and the new throughput constraint is met successfully. Further, there is no effect on the throughput of the other application.

For the same experiment with the centralized RM, there is performance degradation for the JPEG decoder when the throughput constraint of H.263 decoder is varied. The reason for this degradation is the monitoring period of centralized decoder. Figure 5.9(a) shows the throughput of both applications. It is clear that the response of centralized RM is slow as compared to that of credit-based RM. Figure 5.9(c) shows the same experiment with rate-based RM. The response for application constraint variation is quite fast for rate-based RM. The resources freed by the second instance of H.263 are consumed by the H.263 decoder executing at higher rate and the system achieves steady state.

5.4.5 Buffer Requirement

The throughput constraints for this experiment have been assumed to be 2 QCIF frames/sec for JPEG decoder and 40 frames/sec for H.263 decoder. This translates to 4.95×10^{-06} iterations/clock cycle for JPEG and 1.0×10^{-06} iterations/clock cycle for H.263 decoder. In this experiment, we study the jitter in application execution introduced by the RMs. The jitter results in large buffers at the output of the applications to store the frames/macro-blocks decoded by the applications. The desired buffer space shown in the Figure 5.10 is the ideal buffer space required in order to remove all jitter in application execution.

The monitoring period for centralized RM is 40 million cycles. The central resource manager continuously monitors the throughput of all the applications and sends messages to disable the applications having more throughput than the desired throughput. Figure 5.10(a) shows that the control is not as smooth and the platform needs higher buffer space because when an application achieves more throughput than the desired throughput then the frames decoded are to be stored in a buffer memory. The jitter in the application execution is introduced



(a) Centralized RM

(b) Credit-based RM

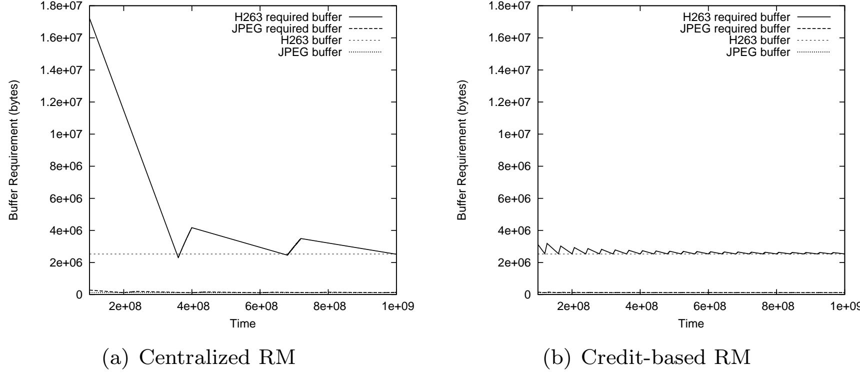
(c) Rate-based RM

Figure 5.9: Run-time change in application constraints of H.263 decoder

due to monitoring period. Figure 5.10(a) shows the buffer requirement of both applications. The JPEG decoder has to decode 198 macro-blocks in one second e.g. 2 QCIF frames/sec. H.263 decoder has to decode 99 $\times$ 40 macro-blocks/sec e.g. 40 frames/sec. This is equal to almost 3 Mbytes of buffer space. The extra buffer space needed for centralized RM is quite large as compared to ideal buffer space requirement.

Figure 5.10(b) shows the budget requirement for credit-based RM. The processors are operating at clock frequency of 40 MHZ and the size of replenishment interval is 1 sec. The actors from the applications are executed according to their budgets and this process repeats each second. It is evident that our credit-based RM requires only a very small amount of extra buffer space as compared to the ideal buffer space.

Table 5.1 compares the jitter (clock cycles) in application execution for the three type of RMs. The maximum jitter for centralized RM is the highest. This is due to the fact that the applications in centralized RM can be executing or dis-



(a) Centralized RM (b) Credit-based RM

Figure 5.10: Comparison of buffer requirement

Table 5.1: Comparison of Jitter (in clock cycles) between three RMs

App.	Centralized based		Credit-based		Rate-based	
	Avg. jitter	max. jitter	Avg. jitter	max. jitter	Avg. jitter	max. jitter
JPEG	256,528	1.59×10^8	43,200	123,600	355	93,600
H.263	281,766	2.79×10^8	1.72×10^5	1.98×10^6	730	1.72×10^5

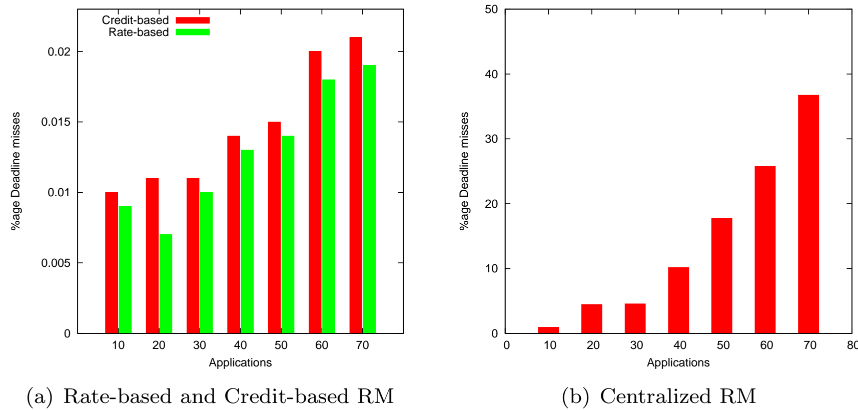
abled for longer periods of time. Hence the time difference between two successive executions of applications can be very large. In Credit-based RM, the maximum jitter is smaller than the centralized RM. This results in low buffer requirement. The maximum jitter for Rate-based is the lowest. The average jitter of distributed RMs is also lower than the centralized RM. Low jitter is also an indication for low buffer requirement.

5.4.6 Processor Utilization of Resource Managers

Table 5.2: Processor utilization of RMs.

Centralized	Credit-Based	Rate-based
0.1672	0.1625	0.8074

The processor utilization of the three RMs is shown in Table 5.2. The applications and their constraints are the same as used in the experiment of subsection 5.4.5. The processor utilization is the ratio of time spent on the applications to the total processor time. The platform consists of 6 processors and the table shows the average processor utilization of the RMs. The processor utilization of



(a) Rate-based and Credit-based RM

(b) Centralized RM

Figure 5.11: Scalability of RMs with increasing number of applications and processors

Rate-based RM is highest of all RMs. This is due to the fact that the applications execute continuously and try to use the compute resources to the maximum.

5.4.7 Scalability of RMs

The scalability of our RMs is evaluated with increasing number of processors and applications. Two platforms models consisting of 10 and 20 processors are built. The platform with 10 processors is used to simulate up to 30 applications and from 40-70 applications, the platform with 20 processors is used. Figure 5.11(a) shows that both RMs are scalable with number of applications and processors. The figure also shows that the deadline miss rate of Rate-based RM are lower than Credit-based RM. Figure 5.11(b) shows the poor scalability of centralized RM with same experiment settings. The deadline misses for centralized RM are more than rate and credit-based RMs.

5.5 Related Work

Research on multi-processor real-time scheduling has mainly focused on pre-emptive systems [DD86, BCPV96]. Non-pre-emptive scheduling has received considerably less attention. It was shown by Jeffay et al. [JS91] and further strengthened by Cai and Kong that the problem of determining whether a given periodic task system is non-pre-emptively feasible even for a single processor is already intractable [CK96]. Recently, more work has been done on non-pre-emptive scheduling for multi-processors. S. Baruah [Bar06] presented sufficient conditions for a periodic task system to be feasible on multi-processor platform. In short, non-pre-emptively scheduling periodic and non-periodic tasks on multi-processor systems is NP-hard. There are many heuristic based solutions to this problem.

There are a number of budget schedulers [MMB07, SBW09, MST94] described in the literature. These schedulers assign a fixed time for each task in a replenishment interval. Steine [SBW09] has added priorities on top of the budgets (PBS). The priorities of all tasks have to be defined at design time and one task can have its priority defined during run-time. Their technique can be used at design time whereas our technique is for run-time resource management. The work by [MST94] is for single processor scheduling.

TDMA is also considered as budget based scheduler. Computing worst-case waiting time taking resource contention into account for round-robin [Hoe05] and TDMA [SBGC07] (requires pre-emption) scheduling has also been analyzed. However, a potential disadvantage of these approaches is that the analysis can be very pessimistic. In [CM08], internal and external contention in communication streams is considered, but their region forming approach is targeted at homogeneous meshed platforms, and is not suitable for heterogeneous or irregular architectures. In [MMB07], an architecture driven approach is used to map tasks first on virtual tiles, which are in turn clustered on elements connected to the same router. They use TDMA schedulers at the processors for budget enforcement. Task switching in TDMA is pre-emptive while our RMs use non-pre-emptive scheduling. The distributed approach of [AFKH08] uses a static mapping algorithm inside its clusters. This approach requires hardware support for cluster management, while it poses more constraints on the size and structure of applications.

In [KMT+08], a resource manager has been proposed which shifts the burden from compile time analysis to run-time monitoring and intervention. They advocate the fact that compile-time analysis of all possible use-cases can provide performance guarantees, but the potentially large number of use-cases in a real system makes such analysis infeasible [KMT+08]. However their resource manager is centralized and not scalable. We have shown this with the help of experiments in Section 5.2. We studied the factors affecting the scalability of resource manager and proposed two distributed resource managers aimed to be more scalable with increasing number of applications and processing elements.

StarSs [BPBL06] is a programming model and run-time manager from Barcelona Super Computing Center. It is an extension of OpenMP [CDK+01]. It consists of a “C” to “C” compiler which converts a sequential C-code into a C-code that can execute on the PPE and SPEs of the Cell processor. The run-time manager executes on the PPE and distributes the jobs between the SPEs whenever it finds a free SPE. The run-time manager is centralized and communicates with the SPEs using the mailboxes. Nanos++ [ABC+09] is another centralized resource manager that consists of a CPU manager and a scheduler. The CPU manager acts as an admission controller and all applications send requests to the CPU manager. The CPU manager selects a scheduling policy and computes the required number of processors for each application and assigns them. The CPU manager monitors the performance of applications through file systems of the processors. Both StarSs and Nanos++ monitor the performance of applications centrally, in contrast to

the resource managers presented in this chapter; they monitor the performance of the tasks of applications locally to increase the scalability of distributed RMs.

5.6 Conclusions

We have presented two versions of a distributed resource manager (RM) for multiprocessor Systems-on-Chip, and compared them to a centralized resource manager. Experiments show that the credit-based RM is very effective for enforcing throughput constraints, and the rate-based RM is effective for obtaining high resource utilization in the context of applications that can profit from the availability of more resources. Both distributed RMs can cope with a large number of processors as well as large number of concurrently executing applications compared to a centralized RM. Furthermore, our experiments demonstrate that they deal better with application and user dynamics, and require less buffering. We conclude that our approach is effective for controlling the computational resources in a multi-processor platform, can deal with data dependencies and dynamically varying execution times that characterize modern media applications, without requiring support for pre-emption. We can therefore fill the gap left by existing techniques like rate-monotonic scheduling that cannot satisfy the abovementioned requirements.

In the next chapter, a simulation framework is presented that can provide average-case performance results of multiple applications executing on an MPSoC platform.

CHAPTER 6

Distributed Simulation on FPGA

MPSoC platforms for real-time applications are designed for worst-case task execution time estimates. It has been shown by Kumar that the average-case performance is often two fold better than worst-case estimated performance [Kum09]. Kumar performed the experiment by assuming virtualized resources for each application and then executing the applications onto a platform and measured their performance. This effect becomes more evident with increasing number of applications. Therefore, knowing the average-case performance is also important for the system designer. In chapters 3 and 4, we presented a design approach which can synthesize MPSoC platforms to meet the throughput constraints of multiple applications in all use-cases using worst-case execution time estimates. This chapter describes a simulation framework that can report average-case performance of applications executing on these synthesized MPSoC platforms.

Simulation in software is often used for performance evaluation. Unfortunately, the accuracy of simulation is often directly proportional to the time spent on it. Further, existing techniques for performance evaluation are limited to single-application designs [Stu07, HPB08]. Hardware acceleration is sometimes also used to speed simulation. However, it generally requires a high design-time effort to build a simulation model in hardware. Some techniques do exist that provide automated flows, but they only simulate the system from the perspective of architecture (functional units) and not that of applications [BI+95, DRCO05, CWB05]. Therefore, an automated hardware simulation design synthesis approach is needed that can deal with the large number of applications and use-cases in modern multi-processor systems.

Further, the applications share the MPSoC platforms with the help of scheduling polices. Dynamic arbitration methods are used as schedulers and it is very difficult to analyze the effect of these arbiters on the performance of these concurrently executing applications. Simulation can be employed to predict the performance of applications under dynamic arbitration polices.

In this chapter, an automated FPGA-based simulation methodology (MAMP-SIM) for performance evaluation of multiple applications executing concurrently on multi-processor systems is presented. The architecture description is specified including the desired mapping of tasks to processors, and the arbiter type for each processor. The target multi-processor platform is generated where each processor is simulated using a Xilinx Microblaze processor in the hardware. The properties of each task are preserved during software generation for each processor. The desired arbiter for each processor is also generated automatically.

Further, it is observed that dynamic arbiters in a processor may lead to causality errors in simulation. In order to prevent this, parallel discrete event simulation (PDES) principles are used [CM79b]. Typically, PDES is used to accelerate sequential program execution on parallel machines. In our approach, PDES is used for simulating multiple applications – each consisting of parallel tasks – executing on multiple processors. Most PDES approaches fall under one of the two categories – *conservative* and *optimistic*. It is proposed to use a *smart conservative* approach that is intelligent to figure out when the sequential program execution can be set aside for improved efficiency. A mechanism has been developed which on every simulation step checks whether proceeding with the simulation on incomplete information can result in a causality error. By default, conservative PDES is used and as soon as it is found that causality errors can be avoided with non-sequential execution, the simulation proceeds and does not follow sequential execution. This mechanism is called as *smart conservative*.

This chapter is organized as follows. In the next section, we describe our simulation platform generation methodology. Section 6.2 extends PDES for multiple applications. In Section 6.3, the implementation of the methodology on Xilinx FPGA and a case study is discussed. Section 6.4 presents a case study of simulation of dynamic scheduling polices.Section 6.5 explores the related work in this domain and Section 6.6 discusses the conclusions inferred from this work.

6.1 Simulation Platform Generation

In a refined MPSoC architecture, where processors are already selected and hardware and software components have been defined, a global simulation model can be used for performance analysis. MPSoC architectures are composed of multiple processors, hardware IPs, memories, and peripherals. Evaluation of individual components is not sufficient to analyze the system performance. The situation is further complicated if the platform has to support multiple applications.

Our simulation platform is a network of nodes/processors connected to each

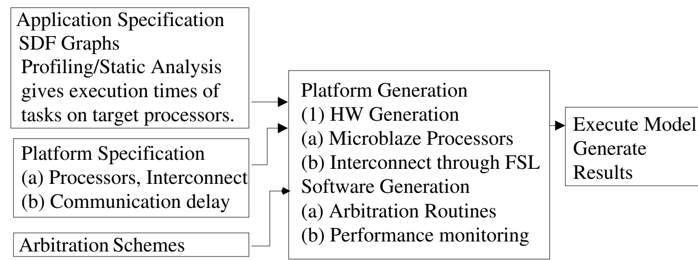
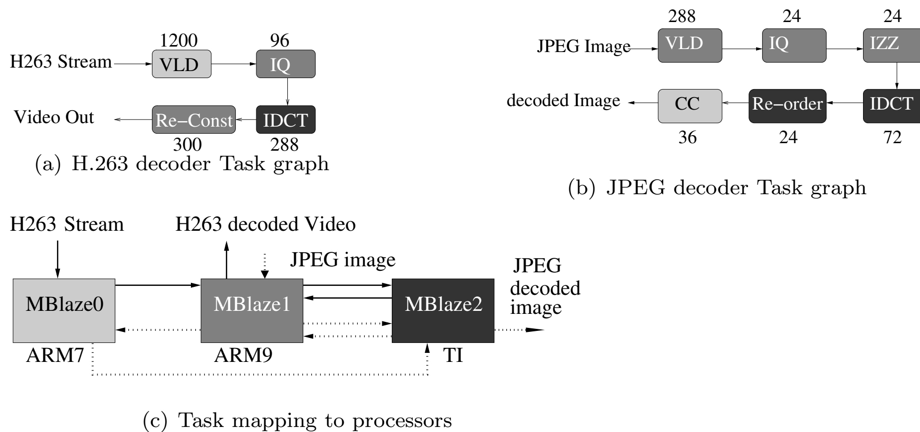


Figure 6.1: simulation methodology

other through First-in-First-out (FIFOs) buffers. The nodes mimic the behaviour of SDF actors and continuously check their input ports and wait for arrival of data units called "tokens". The nodes then process these tokens and forward the results to connected nodes. FIFOs are placed between the nodes so that synchronization signals between the producing and consuming nodes are not required. Our simulation methodology is shown in Figure 6.1.

A system of K applications having N_K actors is simulated. For example "$a_0, a_1, ..., a_{N_{a-1}}$" are "$N_A$" actors from application A and "$b_0, b_1, ..., b_{N_{b-1}}$" are "$N_B$" actors from application B. Actors from the same application communicate with each other exclusively through messages going through edges "c". We assume that the execution times of the actors on the target processors are known by profiling or by static analysis. Along with the application information, the platform information including the number of processors, their interconnect topology, and communication delay is also required. At present the implementation supports point-to-point networks only but the methodology also supports shared networks.



(a) H.263 decoder Task graph

(b) JPEG decoder Task graph

(c) Task mapping to processors

Figure 6.2: Simulation Platform Generation

Table 6.1: Processor-Actor Assignment.

Processor	H263	JPEG	Scheduling policy
ARM7	VLD	CC	FCFS
ARM9	IQ,Re-const	VLD,IQ,IZZ	RRWS
TI	IDCT	IDCT,Re-order	FCFS

Figure 6.2 shows an example of our simulation technique. H263 and JPEG are two applications to be simulated on a three processor platform. Assume that ARM7, ARM9 and a micro-controller from TI constitute a multi-processor platform. The task graphs of applications are shown in Figure 6.2(a) and Figure 6.2(b). Table 6.3 shows mapping of tasks from the two applications on each processor and the task scheduling policy at each processor. The platform is simulated on a 3-Microblaze system which has one to one correspondence with the target platform as shown in Figure 6.2(c). Each processor can have actors from different applications and maintains a local time-stamp "ts_p" of its progress. In addition each actor also carries with it its own time-stamp "ts_a". The time-stamps indicate how far in simulation time the processor or the actors have progressed. For a processor, the time-stamps are updated when it executes an actor or when it is idle. The same goes for actors. The time at which an actor is ready, is determined by the time-stamp of the latest available token from all input edges of an actor. Following are important definitions used in our approach.

Definitions:

- The **time-stamp** of a processor ts_p is equal to the finishing time of the last actor executed on the processor.

$$ts_{p_{new}} = max(ts_p, ts_a) \tag{6.1}$$

- Before an actor can fire, it has to ensure that no other inputs can cause any change in the decision to execute the actor. This is also called **safe to execute**. When an actor executes, its execution time $\alpha(a)$ is added to maximum of the processor and actor time-stamp.

$$ts_{a_{new}} = max(ts_a, ts_p) + \alpha(a) \tag{6.2}$$

- For processors having a single-actor, the processor can fire the actor when it is ready.

- Each FIFO buffer has a time-stamp associated with it indicating the minimum time to which it has progressed. When empty, it is the time-stamp of the last data item read from the FIFO.

Theorem 1 (S)*uccessive time-stamps* ts_{pi}*,* $i \in \mathbb{N}$*, on a FIFO (edge) are guaranteed to be non-decreasing i.e.* $ts_{pi} \geq ts_{pj}$ *for* $i > j$*.*

Proof. Let p_0 and p_1 be two processors connected to each other through a FIFO edge. Let t_{p1} and t_{p2} be time-stamps of successive tokens sent from p_0 to p_1. We know that the token received by the FIFO is the processor time of the processor producing this token, and according to Equation 6.1 it is obtained by taking the maximum value of processor time and actor time, so $t_{p2} \geq t_{p1}$. Hence time-stamps on a FIFO edge are guaranteed to be non-decreasing.

Algorithm 9 achieves the time update for each actor. The algorithm reads the input tokens from all the incoming edges of the actor. Since we are only simulating the performance, we are only interested in the time-stamps of the data when it was produced. The actor is ready to fire at the time the last token is available. Since we are sure the tokens in any FIFO are in non-decreasing time-stamps, we can simply check the time-stamp of the last token read on all the edges to determine the ready time of the actor.

Algorithm 9 Determining the ready time-stamp of all the actors.

1: // ts_a is ready time-stamp of an actor a.
2: **for all** Incoming edges c where $sink_c = a$ **do**
3: Read required input tokens on c
4: Let ts_c be the time-stamp of the last token on c
5: $ts_{a_{new}} = \max(ts_a, ts_c)$
6: **end for**

The same is illustrated by means of an example in Figure 6.3. The actor has two incoming edges and the number of tokens needed from each edge is shown on the respective FIFO. The time-stamps of the last tokens read on the left and on the top edge are 10 and 9 respectively. This implies that the actors can only start executing at time 10. Since the actor needs 3 time-units to execute, the tokens produced on the edge have the time-stamp of 13. Right side of Figure 6.3 shows the state of the FIFO buffers after the execution of actor a_0. This assumes that there is no contention on the processing node and that the actor can start execution as soon as it is ready.

Every processor contains software scheduler, responsible for selecting the next actor according to the scheduling policy. For static scheduling methodologies like Round Robin (RR) and static order [KMT+08], the actor execution order is fixed and does not depend on the arrival of actor time-stamps. In such deterministic cases, there is no problem of ordering of actor execution in a simulation model. For Dynamic arbitration schemes like FCFS and Round Robin with Skipping (RRWS), the simulation model should behave the same way as the physical system. However, this can lead to deadlock in the system. To remove deadlocks, we

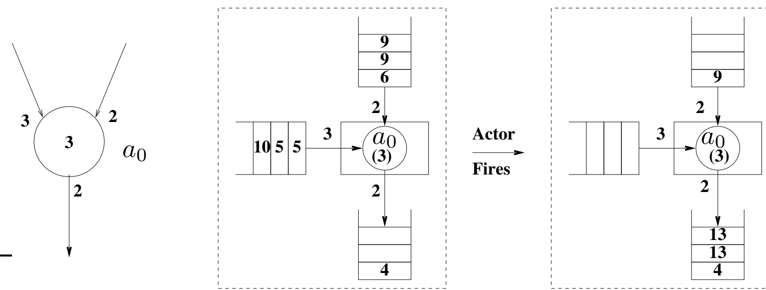


Figure 6.3: FIFO contains time-stamps of the tokens and determine when an actor fires.

have implemented the well-known technique used in PDES. More details are in the next section.

6.2 PDES For Multiple Applications

PDES is often used to accelerate the simulation with the help of a parallel computer. In this work, we employ PDES to simulate the execution of multiple applications on an MPSoC platform. Simulation on a single processor is easy, as the global information is available. When simulation is distributed, it is difficult to decide which data should be transmitted globally so that there is no deadlock situation.

6.2.1 Deadlocks

An important issue in distributed simulation is deadlock. According to [CM79a], a deadlock condition is defined as

1. Not all the processes in a network of processes have terminated and
2. No process is executable.

In dynamic scheduling like FCFS, the ready actor with the lowest time-stamp is executed. However if one of the input FIFO of the actor having lowest time-stamp is empty; the actor blocks. If a cycle of interdependent empty FIFOs arises that has lowest time-stamps, then each actor in that cycle must block and the simulation deadlocks. Figure 6.4 shows one such situation. In this example, both processors P_0 and P_1 have two actors each from different applications A and B. Their respective actor execution times (2 units each) are also shown in the figure. We assume that at the beginning of simulation, actors a_0 and b_1 execute due to initial tokens at their inputs queues. Processor times of both P_0 and P_1 are incremented to 2 as shown in Figure 6.4. Now assume actors a_0 and b_1 receive

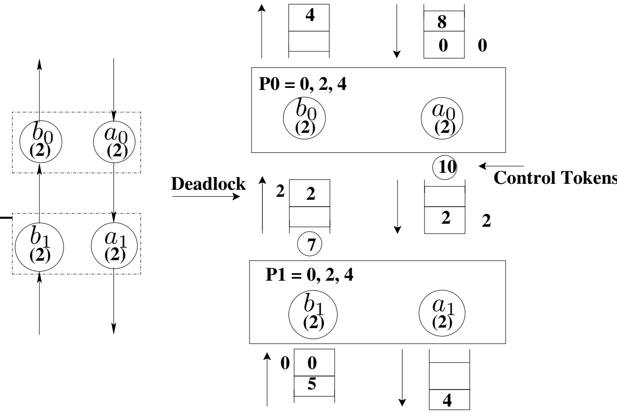


Figure 6.4: Left, System deadlock. Right, Control tokens remove deadlock from the system.

time-stamps 8 and 5 at their queues, respectively. Note that tokens removed from the queues during previous execution are shown beside the queues (Figure 6.4). Actors a_1 and b_0, both have data in their input queues and their time-stamps are lower than that of a_0 and b_1 (2 each for a_1 and b_0 against time-stamps 8 and 5 of a_0 and b_1, respectively). So both a_1 and b_0 fire resulting time stamps of 4 each, as shown in their output queues (Figure 6.4). The time-stamps of the tokens at input FIFOs of actor a_1 and b_0 are lowest (2 each for a_1 and b_0 against 8 and 5 of a_0 and b_1 respectively) but they are waiting for their data. There is a cycle of dependencies between all four actors and the simulation deadlocks and cannot proceed any further. To rectify this problem, we forward the expected arrival time of tokens to the next processor. We call these tokens "control tokens". These control time-stamps require the system to be predictable, i.e. we are able to predict the time-stamp values of an actor from the knowledge of its previous executions. In case of SDF graphs we can predict the finishing time of an actor execution by Equation 6.3. These deadlocks can be avoided by a mechanism of look ahead (in our case we call them "control tokens") described in [CM79b]. Readers can read [CM79b] for further details and proofs.

$$t_{cntrl}(a) = \alpha(a) + max(ts_p, ts_a) \tag{6.3}$$

where t_{cntrl} is the time-stamp of the control token to be sent to the next processor. The execution time of actor is represented by $\alpha(a)$. This time-stamp is a message to the receiving processor that it will not receive any output from sending processor before time t_{cntrl}.

In case of above example, if actor b_1 forwards control token (encircled in Figure 6.4) having time-stamp of 7 and actor a_0 forwards control token of time-stamp 10 (according to Equation 6.3)), the deadlock resolves and simulation continues.

Control time-stamps are forwarded only if their sent time is less than the processor time.

6.2.2 Smart Conservative PDES

PDES, also called distributed simulation, refers to the execution of a single discrete event simulation program on a parallel computer. All algorithms for parallel simulation fall into two class, either **conservative** or **optimistic**.

- Parallel simulations are conservative if they satisfy the property that no process receives information from any other process that **predates** the current simulation time of the receiving process [Fuj89].
- They are optimistic if the processes can act on incomplete information thus admitting the case where messages may arrive **in the past**.

Optimistic methods exploit more parallelism; however they require some sort of roll back mechanism to an earlier valid state. To achieve this synchronization, each process must checkpoint its state and event information, which requires storage space [Fuj89]. In multi-application systems for which dynamic schedulers are used, an optimistic approach generates a lot of correction traffic due to close dependence of time-stamps on the successive actors in an application. This increased traffic will affect performance monitoring of the application.

So we propose a new PDES approach which we name as **smart** conservative PDES. Our approach is different from conservative PDES as we develop a mechanism to find out those cases for which violation of event list order cannot produce the causality errors. In these cases we proceed with the simulation and do not wait unnecessarily for information which will not affect the simulation order. If our mechanism determines that there is a possibility of causality error, we switch to conservative PDES. An example in the next subsection will further clarify our approach. Our approach borrows many concepts from PDES work by Chandy [CM79b] as we are using event driven simulation. PDES simulates a single program onto a multi-processor platform. However, we have used PDES to simulate multiple applications modelled as SDFGs. So not only the actors of an application are to be executed in a sequence, scheduling of actors from different applications should also follow some scheduling policy. Our methodology does not require any central scheduler to synchronize the processing nodes, which makes our approach more scalable than those one with a central scheduler.

6.2.3 Motivating Example

In this subsection, we show that for multi-application simulation, our smart conservative PDES provides an efficient solution. Figure 6.5 shows two actors from two applications mapped onto one processor P_0. Their corresponding SDF graphs are shown on top of the figure. Actor a has two incoming edges and actor b has

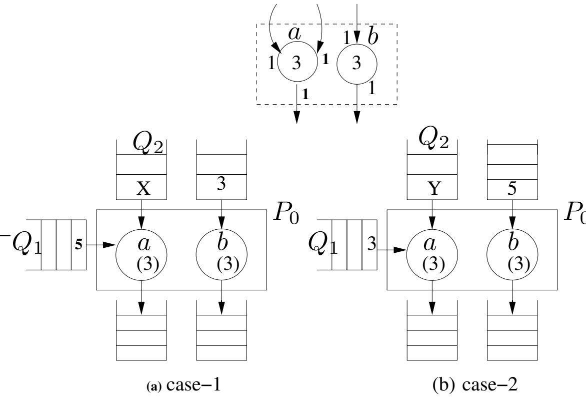


Figure 6.5: Smart conservative PDES example

one. Rate at all edges is 1. Figure 6.5(a) shows the case when actor b has a lower time-stamp than actor a and second edge of actor a is empty (so the token X in queue Q_2 has not yet arrived). For this case, if these actors are scheduled using the FCFS, the token at Q_2 can be ignored as actor time of a is $ts_a = max(5, X)$ (according to Algorithm 9). Even if the time-stamp of the received token at Q_2 (which is X), is lower than 3, b should be executed (as for FCFS we are looking for the actor having minimum time-stamp value given by $min(max(5, X), 3)$. This property of not waiting for the arrival of token “X” at Q_2 is an example of a **smart conservative** scheduling decision.

Figure 6.5(b) shows the case, when one of the queues of a has a lower time-stamp than that of b and the other queue of a is empty. Let “Y” be the expected time-stamp value of token at Q_2. The value of Y can effect the scheduling decision by the relation $min(5, max(3, Y)$. It can be lower or higher than 5, in this case we use conservative approach and wait for the arrival of time-stamp at Q_2.

6.2.4 Dynamic Actor Arbitration

Algorithm 10 shows our smart conservative FCFS arbitration. The algorithm waits for any actor to get ready for execution (line 4). Then it checks the minimum time of the actors. Next, the algorithm verifies that the processor time is not less than the minimum actor time. This can happen if all the actors get ready during the time the processor was idle. In this case we increase our processor time to minimum actor time (line 12). The actor with smallest time stamp is executed if it is ready. Control time-stamps are sent only if the actor is not ready or its time-stamp is not the minimum.

Algorithm 11 shows the pseudo code for RRWS arbitration under PDES. Like FCFS, the algorithm ensures that if the processor time is less than all the actor times then it sets the processor time to minimum actor time. Then it picks an

Algorithm 10 FCFS arbitration algorithm for PDES.

Procedure: Check_FCFS()

Input: actors $a \in$ Application A and Execution time of actor a $\alpha(a)$
Output: FCFS

```
1: min_time = 1
2: // Check which of the actors are ready to execute.
3: for all actors a do
4:    actor_ready[a] = is_actor_ready(a)
5: end for
6: for all actors a do
7:    if (ts_a[a] < min_time) then
8:       min_time = ts_a[a]; actor = a; flag_min_time = 1;
9:    end if
10: end for
11: // If all the actors get ready after the processor was last idle then we set the //ts_p equal
    to the minimum time of the actors.
12: ts_p=min_time
13: if (actor_ready[a] && flag_min_time == 1) then
14:    execute(a)
15: else
16:    send_control_tokens(max(ts_p, ts_a) + α(a)); return Check_FCFS()
17: end if
```

actor from list of actors and checks if it is ready (line 9). If the actor is ready then it determines if it is safe to execute this actor; e.g. if the processor time is less than the actor time, then we cannot execute this actor, because there is a possibility that an actor from other application may arrive before the ready time of this actor; so we skip the actor (line 12). If this is not the case, we execute the actor and pick the next actor from the list. On the other hand if the actor is not ready yet and processor time is more than this actors' ready time, then we will have to wait for the arrival of tokens for this actor and we cannot skip it, as shown on line 19 of Algorithm 11. Similarly, if the actor is not ready and its time is greater than the processor time then the tokens for this actor will arrive in the future and we can safely skip to the next actor (line 22).

6.3 FPGA implementation, Experiments and Results

The simulation platform has been implemented onto a Xilinx FPGA board (XUP). The tasks are mapped to Microblaze processors. The FIFO links are mapped to FAST Simplex Links (FSL). Additional peripherals such as timer, UART, and SysAce are also used in the design. UART is useful for printing debugging information of the system. Performance results of each use-case are stored in the SysAce Compact Flash card. A Timer is used for profiling the application.

Our implementation platform is the Xilinx XUP Virtex II Pro Development Board with an xc2vp30 FPGA on-board. Xilinx EDK 8.2i and ISE 8.2i were used for synthesis and implementation. All tools run on a Pentium dual core at 2.0GHZ with 2.0GB of RAM.

Algorithm 11 RRWS arbitration algorithm for PDES.

Procedure: Check_RRWS()

Input: actors $a \in$ Application A and Execution time of actor a $\alpha(a)$
Output: RRWS

```
1:  min_time=1
2:  for all actors a do
3:      if (ts_a[a] < min_time) then
4:          min_time = ts_a[a]
5:      end if
6:  end for
7:  // If all the actors get ready after the processor was last idle then we set the //ts_p equal
    to the minimum time of the actor
8:  ts_p= min_time.
9:  // Check the next actor from the list if it is ready
10: if (is_actor_ready(a)) then
11:     if (ts_p < ts_a[a]) then
12:         Skip the actor; return Check_RRWS();
13:     else
14:         Execute(a)
15:     end if
16: else
17:     if (ts_p >= ts_a[a]) then
18:         send_control_tokens(max(ts_p, ts_a[a]) + α(a));
19:         Wait for the actor to get ready;
20:         Check_RRWS()
21:     else
22:         Skip the actor; return Check_RRWS();
23:     end if
24: end if
```

6.3.1 DSE Case Study

To predict the performance of applications for different buffering options, time spent during hardware synthesis is the limiting factor. The solution is to synthesize a super-set hardware for all use-cases in a typical design space exploration problem, and only change the software for each point in the design space as shown in Figure 6.6. Interested readers are requested to read [KFH+08] for more details about this technique.

We present a case-study to use our simulation methodology for performing a design space exploration to compute the optimal buffer requirement for two applications running concurrently on a multi-processor platform. Minimizing buffer size is an important objective when designing embedded systems. We explore the trade-off between the buffer-size used and throughput obtained for multiple applications. Increasing buffer space exploits more parallelism in the platform. For single applications, the analysis is easier and has been presented earlier [SGB06b]. For multiple applications it is non-trivial to predict resource usage and performance because multiple applications cause interference when they compete for resources [KFH+08].

The case study is performed for JPEG and H.263 applications. Both applica-

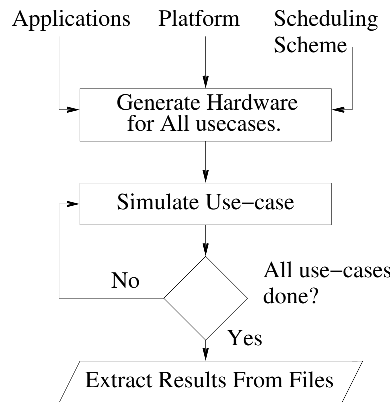
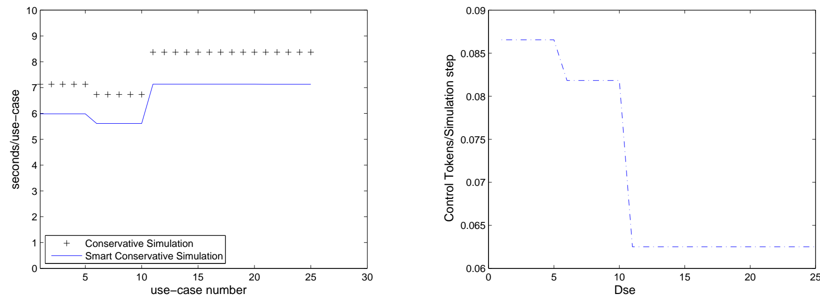


Figure 6.6: Software for each use-case is loaded one by one

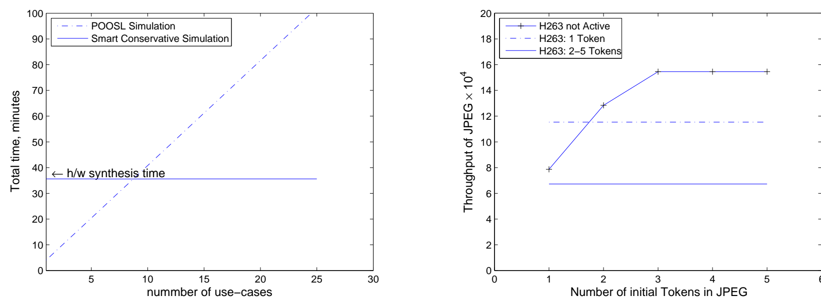
tions are mapped onto a 3-processor platform. Figure 6.2 showed the task graphs and actor mappings of both applications. The actors are mapped on processors by equally distributing the computation load. In this case study the buffer size has been modelled by the initial tokens present on the incoming edge of the first actor. The higher this initial-token count, the larger the buffer needed to store the output data. In case of H.263, each token corresponds to an entire decoded frame, while for JPEG, it is the complete image. The design space consists of 25 points, e.g. for both applications the number of initial tokens are varied from 0 to 4. Figure 6.8(b) shows how the throughput of JPEG decoder varies with increasing number of tokens in the SDF graph. When the number of tokens (i.e. buffer-size in the real application) is increased, the throughput also increases until a certain point after which it saturates. When the JPEG decoder is the only application executing (obtained by setting the initial tokens in H.263 to zero), we observe that its throughput increases almost linearly till 3 initial tokens. We further observe that increasing the initial tokens of H.263 worsens the performance of JPEG. Hardware synthesis time of the whole design on the FPGA was about 35 minutes. This is a one-time overhead and after that we only compile the software for each design point and download that to get the results. We compare the performance of our tool with a POOSL [vdPV97] model. POOSL is a very expressive modelling language with a small set of powerful primitives and completely formally defined semantics. It furthermore serves as a basis for performance analysis.

In [KMT+08], authors have created a tool to find application throughput and buffer requirements using POOSL. We compare our FPGA simulation platform results with this POOSL model. Total time taken for POOSL simulation is 95 minutes whereas the FPGA simulation took 41 minutes only (including the hardware synthesis time). The speed up gained for FPGA simulation platform for two applications is 16 (excluding the hardware synthesis time, 41-6 = 35 minutes) as



(a) Conservative vs smarter conservative (b) Number of control tokens/simulation step

Figure 6.7: Comparison of smarter conservative with conservative PDES.



(a) POOSL vs smarter conservative (b) Effect of varying initial Tokens on JPEG throughput

Figure 6.8: Comparison of Smart conservative PDES with POOSL simulation.

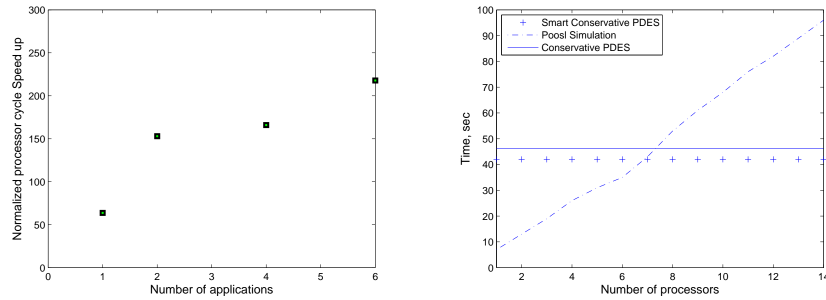
shown in Figure 6.8(a) .

Figure 6.7 presents results from this case-study. Figure 6.7(a) shows the time taken by smart conservative approach at each design point. Smart conservative approach is 15% faster than the conservative PDES. This seems a small improvement. However in DSE problems, the user has to perform thousands of such tests and improvements like these are evident in overall results. For example, if we have to perform same DSE for 4 applications, the design space will contain 625 points. To explore this huge design space, our smart conservative saves 15% simulation time. Figure 6.7(b) shows the number of control tokens per simulation step at each DSE point. Techniques like [PVA+08] require one control token at each simulation step whereas in our case the required number of control tokens/simulation step is very low. Figure 6.7(b) shows a decrease in number of tokens with each DSE point. This is because from use-case 1-5, initial H263 tokens are zero so only one application is executing in the platform. This results in more control tokens. As the number of tokens of H263 is increasing from use-case (6-25), both applications are active and fewer control tokens are required due to the increased exploitable parallelism.

6.3.2 Scalability

Figure 6.9(b) shows the scalability of our FPGA simulation as compared to POOSL. In this experiment, we increased the number of processors in a single application by one processor at each experiment step. We simulated 500,000 iterations of each resulting graph. It is evident from Figure 6.9(b) that for fewer processors, POOSL simulation is faster than our FPGA implementation. It should be noted that our FPGA platform is operating at 50MHz and POOSL at 2.0 GHz. However, as we increase the number of processors, POOSL simulation gets slower and the simulation time keeps on increasing. On the other hand, FPGA simulation takes almost the same time, as we keep on increasing the number of processors. We increased the number of processors up to 14. This is the highest number of Microblaze processors which can be synthesized on Xilinx xc2vp30 FPGA used for our experiments. FPGA hardware synthesis time is not included for the scalability results.

Figure 6.9(a) shows the normalized processor cycle speedup of FPGA simulation against the POOSL simulation. POOSL and FPGA simulation is running at different frequency using different type of processors, so to have a comparison we convert their simulation time to normalized processor cycles. Our POOSL simulation is running on Intel dual core 2.0 GHZ processor (only one of the processor being used) and our FPGA simulation is mapped on 6 Microblaze processors running at 50 MHz each, as shown in Table 6.2. Normalized processor cycles are obtained by first multiplying the simulation time, number of processors, and processor frequency for both FPGA and POOSL simulation and then dividing the corresponding products of POOSL by the FPGA products. Our normalized processor cycle speed up for 6 applications is as high as 218.



(a) Speedup with number of applications (b) Scalability with number of processors.

Figure 6.9: Figures showing scalability of our simulation technique.

Table 6.2: Normalized processor cycles.

# of Appl.	cycles POOSL $\times 10^{12}$	cycles FPGA $\times 10^{9}$	# of proc. POOSL	# of proc. FPGA	Tot. Cycles POOSL $\times 10^{12}$	Tot. Cycles FPGA $\times 10^{9}$	Speed up in FPGA
1	0.28	0.73	1	6	0.28	4.39	64
2	0.83	.90	1	6	0.83	5.45	153
4	1.12	1.125	1	6	1.12	6.75	165
6	1.71	1.307	1	6	1.71	7.84	218

6.4 Simulation of Dynamic Scheduling Policies

We perform DSE to find the throughput of applications for different actor-to-processor mappings and scheduling policies i.e. FCFS, RRWS, and RR. The case study is performed for JPEG and H.263 decoder applications. Figure 5.5 shows the SDF models of these applications. H.263 is mapped on 4 processors and JPEG on 6. So in total we use 6 processors for mapping these two applications. Four processors are shared by H.263 and JPEG where as two processors have actors from JPEG only.

Our tool is provided with one set of mappings and mapping constraints. It automatically generates the possible combinations from the mapping constraint. The mapping constraint assumes that each processor can have at most two actors. This assumption limits the search space to 28 possible mappings. These 28 different mappings of actors onto processors are simulated. We generate the hardware for all these mappings only once and the software is generated for each mapping individually. At least 500,000 iterations of each graph are executed to find the processor utilization (PU) and application throughput.

Figure 6.10(a) shows the plot of processor utilization and application throughput for each mapping point under FCFS scheduling. Processor utilization (PU) is defined as "the time during which the processing node is doing computation as compared to the total time".

$$PU = \sum_{a=1}^{N} (\alpha(a) \times iters(a) \times rep(a))/Sim_time \tag{6.4}$$

Equation (6.4) is used to calculate the processor utilization for each processor. Here *iters(a)* is number of times an actor a is executed. *rep(a)* is the repetition vector entry for actor a , N is the total number of actors per processor and *Sim-time* is the total simulation time. The throughput graphs suggest that throughput of the applications is highly dependent on their mappings, in case of FCFS. FCFS lacks fairness in distribution of resources between the actors. On the other hand, RR scheduling distributes the compute resources more evenly as shown in Figure 6.10(b).

Figure 6.10(c) shows processor utilization and application throughput results for RRWS. Processor utilization and application throughput are highest for RRWS. In RRWS, if an actor is not ready the processor moves to the next actor and executes it if it is ready. So waiting time is reduced for RRWS scheduling. Where as in case of FCFS, the processor may keep on waiting for minimum time-stamp actor to get ready and then execute, so processor utilization for FCFS is lower than RRWS. Highest processor utilization for RRWS is 0.4421. Highest processor utilization for RR and FCFS are 0.4301 and 0.4254 respectively. The bars in the figures also show the mapping points where JPEG has the highest throughput. The user can choose the mapping point according to desired throughput from each application. Processor utilization guides us about the application load on

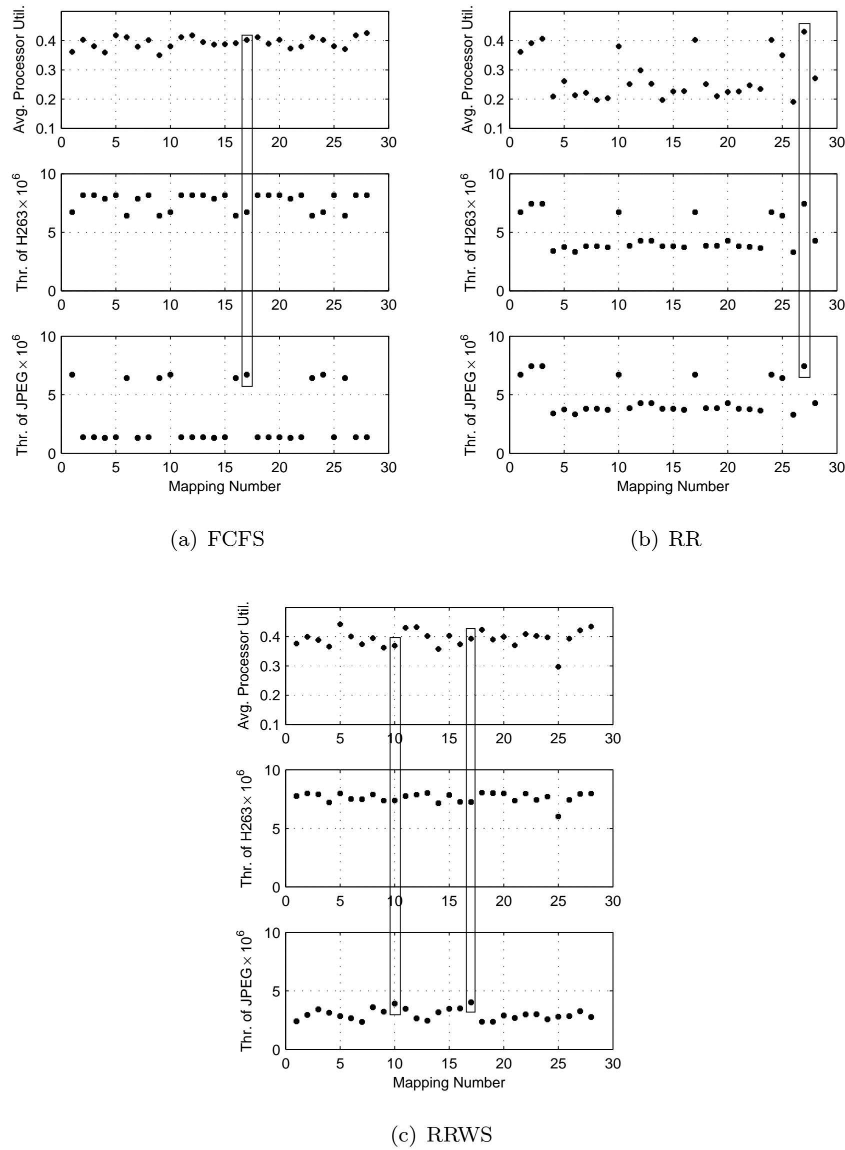


Figure 6.10: Processor utilization and application throughput plotted against mapping number for different schedulers.

the processor and how many actors from other applications can be mapped onto the processors.

Table 6.3 shows the actor to processor mappings, which resulted in maximum performance for JPEG. Mapping number 17 achieves highest JPEG throughput for both FCFS and RRWS. Table 6.3 also shows the corresponding scheduling scheme responsible for the desired application performance. The table also shows the actors mapped on to the corresponding processors P_0-P_5.

Table 6.3: Actor-to-Processor mappings, resulting in higher JPEG performance.

Application	Mapping #	Scheduling Policy	P_0	P_1	P_2	P_3	P_4	P_5
H.263 JPEG	17	FCFS & RRWS	IDCT re-order	Reconst. CC	VLD VLD	IQ IQ	IZZ	IDCT
H.263 JPEG	10	RRWS	IQ IDCT	IDCT re-order	Reconst. CC	VLD VLD	IQ	IZZ
H.263 JPEG	27	RR	VLD IQ	IQ IZZ	IDCT IDCT	Reconst. re-order	CC	VLD

6.5 Related Work

FPGAs are often used as simulation platforms; the reason being their flexibility, relatively low cost, and their property of being re-configurable. Today, FPGAs are fast and big enough to provide scalable alternative to software simulation. FPGA simulation efforts such as RPM [BI$^+$95] have produced a system level multi-processor emulator. Designers can choose from a library of architectural components and evaluate their performance. FAST [DRCO05] is an FPGA-based platform for modelling multi-processor system with MIPS cores. It is suitable for MPSoC memory system research. RAMP [CWB05] is a cycle accurate, distributed concurrent event simulator. It claims to provide the user with the flexibility to configure all components of multi-processor systems like processing elements, communication infrastructure, programming model etc. ATLAS [WCN$^+$07] uses the BEE2 boards from RAMP and its main emphasis is software research for transactional memory. The prototype runs the GNU/linux operating system and runs multi-threaded applications that use transactional memory.

All the above mentioned platforms simulate architectural components inside a processor/multi-processor. On the contrary, our platform performs performance evaluation of multiple applications on a multi-processor fabric. There are also some analysis tools like SDF3 [Stu07] for single applications, and they use heuristics to find the actor-processor mappings. In [HPB08], authors have presented a scheduler for SDF graph simulation on multi-processor platforms; however, it does not support multiple applications. As far as we know, our approach is the first to use FPGAs as simulation platform for performance evaluation of multiple applications running concurrently on MPSoC platforms.

In [KMT+08], the authors have presented a simulation model for multiple applications. The tool helps in finding the throughput of applications, but it is not scalable and like any software simulation environment it gets slower and slower as we increase the number of simulated processors. Simflex [WWF+06] is a micro-architectural simulator for multi-processors. It accelerates simulation by exploiting homogeneity of application behaviours that repeat millions of times. It analyzes the application and chooses the size of sample in such a way to capture the behaviour of the model completely. This technique has a very high overhead due to creation of complete state before execution of each sample. MAMPS [KFH+08] is a tool flow for mapping multi-media applications on FPGA. It evaluates the performance of applications by executing their models on the FPGA fabric, however we only forward the time-stamps of execution. Forwarding time stamps takes fewer cycles as compared to execution of the program model. This makes our approach faster than MAMPS.

A-ports [PVA+08] is another architectural simulator on FPGA. Like our approach, it is also a graph of connected nodes. A node may execute a simulation cycle when all of its inputs are ready. However, there are some key differences. A-Ports nodes are required to send a message every cycle, even if the message indicates that no change in the state of node has happened while we only forward the control tokens if there is a chance of deadlock in simulation.

MPARM [BBB+05] is an environment for MPSoC design space exploration using SystemC. It is complete platform solution for MPSoc simulation composed of processor models (ARM), bus models (AMBA), memory models, hardware support for SMP (hardware semaphores), and a software development toolset including a C compiler and an operating system. It provides several performance statistics, such as cache miss/hit rate and bus contention and average waiting time. In [OWB+07] authors have presented a methodology for performance analysis of processor at different abstraction levels, offering different trade-offs between estimation speed and accuracy. Neural networks are used to provide software performance estimation. The neural networks are first trained on benchmarks of specific processors. The methodology is targeted towards processor selection.

6.6 Conclusions

In this chapter, a novel technique has been presented that accelerates simulation of multiple applications on FPGAs. The technique can be used to measure the average-case performance of applications executing onto an MPSoC platform. The technique is scalable as larger FPGAs are available to simulate designs with a large number of applications and use-cases. The largest Virtex-4 device [Xil11b] from Xilinx can be configured to have about 97 Microblaze processors. Support for such a large number of processors makes our approach very attractive. It is also shown that for two applications the simulation technique is at least 16 times faster than an efficient software solution. This speedup further increases by

increasing the number of applications, use-cases, and processors.

Another contribution of the work is highlighting and solving the problems identified during the PDES simulation for multiple applications. Efficient algorithms for three scheduling policies –RRWS, RR and FCFS have been presented, in combination with a new smart PDES simulation. In the future, we intend to extend the work by supporting more scheduling techniques. It is also planned to include hardware synthesis and performance evaluation options to the framework. This will allow evaluating performance of hardware modules like accelerators for multi-media applications. HDL descriptions will be integrated in the simulation platform to get feedback on performance of hardware modules.

CHAPTER 7

Conclusions and Future Work

In this chapter, the major conclusions from this thesis are presented, together with several issues that remain to be solved.

7.1 Conclusions

The role of modern multimedia systems has changed considerably since their emergence. These systems have to support large number of applications. The applications come from different domains and hence exhibit contrasting compute requirements. Beside functional requirements, the embedded systems also have to meet non-functional constraints like energy and hardware area. Both these constraints affect the cost of the system. These applications execute in different combinations. Further, the market of these systems is expanding all the time. The average life for these devices can be as low as 1 year since a new device with advanced features becomes available in the market. The designers of these systems have to come up with new design techniques so that these systems can be designed in cost effective and efficient way.

Predictability in design techniques shortens the design cycle. We presented a communication assist that introduces predictability in MPSoC platforms. The CA relives the processor from transferring data duties and the processor has to perform the computation only. This decouples the communication from computation. The CA assumes that the interconnect can provide guarantees on the transfer. This assumption is quite valid as most of interconnects provide timing guarantees on their operations. The CA implements the input/output semantics

of SDFGs e.g. checking of available space in the output edges and then verifying the presence of the tokens at input edges, before the actual execution of the actor. We presented the SDF model of the CA so that the performance of applications can be predicted on MPSoC platforms containing CA.

We also developed an MPSoC platform generation technique that generates CA-based predictable platforms. The heuristics based technique presented in this thesis generates MPSoC platforms that can support multiple applications and their use-cases. The technique analyzes the use-cases and shares the resources so that the MPSoC platform can be dimensioned to consume minimum resources. The platforms generated consist of processors interconnected with point-to-point interconnect. Further in the thesis, a platform synthesis technique (CA-MPSoC) is presented that can synthesize the generated platforms onto FPGAs with the help of commercial synthesis tools. The platform synthesis technique automatically writes the communication APIs for CAs in the platform. The application functions are automatically inserted the processor codes. The software is generated for each use-case of the applications.

The synthesized platform needs resource managers to handle dynamism in applications. Traditionally centralized resource managers are used in MPSoC platforms. Centralized resource managers monitor application performance at regular intervals and take corrective measures. If an application is not achieving its desired performance then more compute resources are allocated to this application so that it meets its performance constraints. These centralized resource managers do not scale with increasing number of applications and processing elements. The thesis presents two distributed resource managers. The budgets are calculated centrally and distributed to the processing elements. The processing elements enforce these budgets locally. The distributed resource managers are scalable with the number of applications and processing elements.

Simulation is extensively used for knowing the difference between average-case and worst-case performance. In many design problems, simulation is inevitable and the designers are interested in fast simulation techniques. In this thesis, we have presented a fast parallel simulation technique that is used to accelerate the speed of simulation. We have employed the principles of Parallel Descrete Event Simulation (PDES). PDES is used to simulate multiple applications on FPGA platforms. A simulation methodology (MAMPSIM) is presented which automatically generates a simulation platforms consisting of Microblaze processors connected through FSL links. The methodology also implements the scheduling policy for the platform as selected by the user. Efficient algorithms for three scheduling policies –RRWS, RR and FCFS have been presented, in combination with a new smart PDES simulation. It is also shown that for two applications the simulation technique is at least 16 times faster than an efficient software solution. This speedup further increases by increasing the number of applications, use-cases and processors. Another contribution of the work is highlighting and solving the problems identified during the PDES simulation for multiple applications.

Using above contributions, a designer can quickly design and implement pre-

dictable multi-processor based systems capable of satisfying throughput constraints of multiple applications in given set of use-cases, and employ resource management strategies to deal with dynamism in applications.

7.2 Future Work

The thesis presents solutions to design of MPSoC platforms. There are still issues which should be addressed. These issues are summarized below.

SDF Model Extraction

Throughout this thesis, we assume that the SDF model of an application is already provided. This modelling is mostly done manually and can be very time consuming. There is no significant work on automatic extraction of SDF models from a given source code. SUIF [HAA+96] is a compiler that is used to extract parallelism from the application C-code. The tools Compaan [KRD00] and PNgen [VNS07] automatically convert an application into a parallel KPN. These tools require a lot of tweaking from the user to get any kind of performance from them. The result of this problem is a very slow extraction process and Design Space Exploration strategies become even more slower. A possible future work can be a fully automatic SDF model extraction tool from C-code.

Other Models of Computation

All of the techniques developed in this thesis work on SDF. SDF is considered as a static model of computation although CSDF [Bil96] and SADF [TGB+06a] have been developed to model dynamism in applications. It would be interesting to apply the techniques developed in thesis to other models of computation like CSDF, SADF, and KPN, which can express the dynamic behaviour of applications in more efficient way.

Support for NoC Based Platform

In this thesis, the techniques presented do not place any constraint on the type of interconnect between the processors and the experiments have been performed on point-to-point interconnects. The techniques assume that the interconnect can provide guarantees on the transfer and this is a valid assumption as there are NoCs and buses available in the literature which provide bounds on their data transfer. An extension of this work can be to apply these techniques on platforms where the communication network does not provide guarantees on the transfer. It will be interesting to think how the CA will handle such communication networks.

MPSoC Parallel Simulation

A possible extension to this work can be to extend the work by supporting more scheduling techniques like EDF. It is also a possibility to include hardware module (accelerators) synthesis and performance evaluation options to the framework. This will allow to evaluate performance of hardware modules like accelerators for multi-media applications. HDL descriptions will be integrated in the simulation platform to get feedback on performance of hardware modules.

Distributed Resource Managers

The distributed Resource Managers presented in this thesis relieve the central controller in centralized resource managers from the responsibilities of monitoring. The central controller only distributes the credits to the processors which are locally enforced by each processor. This removes the bottle neck created by the monitoring. One extension to the work can be to perform the admission control operation also in a distributed fashion. This will make the resource management more scalable with the number of processors and applications. Further, the resource management techniques presented in this thesis cannot guarantee the performance of applications which are not analyzed during design time. The resource manager will try its best such that the application meets its constraints but cannot provide guarantees. A possible extension to this work can be development of a technique which can guarantee the performance of a newly added application in the MPSoC platform which was not statically analyzed.

Bibliography

[ABC+09] Eduard Ayguade, Rosa M. Badia, Daniel Cabrera, Alejandro Duran, Marc Gonzalez, Francisco Igual, Daniel Jimenez, Jesus Labarta, Xavier Martorell, Rafael Mayo, Josep M. Perez, and Enrique S. Quintana-Ortí. A proposal to extend the openmp tasking model for heterogeneous architectures. In *IWOMP*, pages 154–167. Springer-Verlag, 2009.

[AFKH08] Mohammad Abdullah Al Faruque, Rudolf Krist, and Jörg Henkel. Adam: run-time agent-based distributed application mapping for on-chip communication. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 760–765, New York, NY, USA, 2008. ACM.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[ARM] ARM. Arm primecell^{TM} DMA controller, `http://www.arm.com/armtech/PrimeCell?OpenDocument`.

[Bar06] Sanjoy K. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Syst.*, 32(1-2):9–20, 06.

[BBB+05] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. Mparm: Exploring the multiprocessor soc design space with systemc. *J. VLSI Signal Process. Syst.*, 41(2):169–182, 2005.

[BBjS08] T. Bijlsma, M. Bekooij, P. jansen, and G. Smit. Communication between nested loop programs via cicular buffers in an embedded multiprocessor system. In *Proc. of 11th SCOPES*, volume 296, pages 33–42, 2008.

[BCPV96] Sanjoy K. Baruah, N. K. Cohen, C. Greg Plaxton, and Donald A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[Bil96] G. Bilsen. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44:397–408, 1996.

[BML99] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *J. VLSI Signal Process. Syst.*, 21(2):151–166, 1999.

[BMSM96] Thomas F. Baldwin, D. Stevens Mcvoy, Chrles W. Steinfield, and D. Stevens Mcvoy. *Convergence: Integrating Media, Information and Communication*. Saga Publications, 1996.

[BPBL06] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *ACM/IEEE Conference on Supercomputing*, page 86. ACM, 2006.

[CCK+08] Myong Hyon Cho, Chih-Chi Cheng, Michel Kinsy, G. Edward Suh, and Srinivas Devadas. Diastolic arrays: throughput-driven reconfigurable computing. In *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 457–464, Piscataway, NJ, USA, 2008. IEEE Press.

[CDK+01] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[CFR99] Ricardo C. Corrêa, Afonso Ferreira, and Pascal Rebreyend. Scheduling multiprocessor tasks with genetic algorithms. *IEEE Trans. Parallel Distrib. Syst.*, pages 825–837, 1999.

[CHKW08] Catherine H. Crawford, Paul Henning, Michael Kistler, and Cornell Wright. Accelerating computing with the cell broadband engine processor. In *Proceedings of the 5th conference on Computing frontiers*, pages 3–12, New York, NY, USA, 2008. ACM.

[CK96] Yang Cai and M. C. Kong. Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems. *Algorithmica*, 15(6):572–599, 1996.

[CM79a] K. M Chandy and Jayadev Misra. Deadlock absence proofs for networks of communicating processes. Technical report, Austin, TX, USA, 1979.

[CM79b] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *Software Engineering, IEEE Transactions on*, SE-5(5):440 – 452, sept. 1979.

[CM08] Chen-Ling Chou and Radu Marculescu. User-aware dynamic task allocation in networks-on-chip. In *DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1232–1237, New York, NY, USA, 2008. ACM.

[CM09] CA-MPSoC. Please email at a.kumar@tue.nl for username and password, 2009.

[CSG99] D. Culler, J. Singh, and A. Gupta. In *Parallel Computer Architecture: a hardware/software approach*. Morgan Kaufmann Publishers, Inc., 1999.

[CWB05] Chen Chang, John Wawrzynek, and Robert W. Brodersen. Bee2: A high-end reconfigurable computing system. *IEEE Design and Test of Computers*, 22:114–125, 2005.

[DD86] S. Davari and S. K. Dhall. An on line algorithm for real-time tasks allocation. In *Real Time Systems Symposium*, 1986.

[DJ97] Robert P. Dick and Niraj K. Jha. Mogac: a multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems. In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, ICCAD '97, pages 522–529, Washington, DC, USA, 1997. IEEE Computer Society.

[dK02] E. A. de Kock. Multiprocessor mapping of process networks: a jpeg decoding case study. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 68–73, New York, NY, USA, 2002. ACM.

[DRCO05] John D. Davis, Stephen E. Richardson, Charis Charitsis, and Kunle Olukotun. A chip prototyping substrate: the flexible architecture for simulation and testing (fast). *SIGARCH Comput. Archit. News*, 33(4):34–43, 2005.

[EE10] Stijn Eyerman and Lieven Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. *SIGARCH Comput. Archit. News*, 38:362–370, June 2010.

[EEP03] Cagkan Erbas, Selin C. Erbas, and Andy D. Pimentel. A multiobjective optimization model for exploring multiprocessor mappings of process networks. In *CODES+ISSS '03:*, pages 182–187, 2003.

[FSV99] Alberto Ferrari and Alberto Sangiovanni-Vincentelli. System design: Traditional concepts and new paradigms. In *Proc. of ICCD*, pages 2–12, 1999.

[Fuj89] R. M. Fujimoto. Parallel discrete event simulation. In *WSC '89: Proceedings of the 21st conference on Winter simulation*, pages 19–28, New York, NY, USA, 1989. ACM.

[GHF+06] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[GK10] Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53:58–66, November 2010.

[GNL01] Om Prakash Ganwal, Andre Niewland, and Paul Lippens. A scalable and flexible data synchronization scheme for embedded HW-SW shared memory systems. In *Proc. of ISSS*, pages 1–6, 2001.

[GSBC05] Stefan Valentin Gheorghita, Sander Stuijk, Twan Basten, and Henk Corporaal. Automatic scenario detection for improved wcet estimation. In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pages 101–104, New York, NY, USA, 2005. ACM.

[Gsw06] M. Gswind. Chip multi-processing and the cell broad band engine. In *Proc. of CCF*, pages 1–8, 2006.

[HAA+96] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29:84–89, 1996.

[HCY+07] Wei-Hsuan Hung, Yi-Jung Chen, Chia-Lin Yang, Yen-Sheng Chang, and Alan P. Su. An architectural co-synthesis algorithm for energy-aware network-on-chip design. In *SAC '07:*, pages 680–684, 2007.

[HDT07] K. Huang, D.Grunert, and Lothar Thiele. Windowed FIFOs for FPGA-based multiprocessor systems. In *Proc. of IEEE 7th ASAP*, pages 36–41, 2007.

[Hen03] Jörg Henkel. Closing the soc design gap. *Computer*, 36(9):119–121, 2003.

[HFK+07] Christian Haubelt, Joachim Falk, Joachim Keinert, Thomas Schlichter, Martin Streubühr, Andreas Deyhle, Andreas Hadert, and Jürgen Teich. A systemc-based design methodology for digital signal processing systems. *EURASIP J. Embedded Syst.*, 2007:15–15, January 2007.

[HG11] Andreas Hansson and Kees Goossens. *On-Chip Interconnect with aelite*. Springer, 2011.

[HGR05] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. A unified approach to constrained mapping and routing on network-on-chip architectures. In *CODES+ISSS '05:*, pages 75–80, 2005.

[HLTW03] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of wcet tools. 91(7):1038–1054, 2003.

[HM03] J. Hu and R. Marculescu. Energy-aware mapping for tile-based noc architectures under performance constraints. In *ASP-DAC '03:*, pages 233–239, 2003.

[HM08] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41:33–38, July 2008.

[Hoe05] Rob Hoes. Predictable dynamic behaviour in noc-based multiprocessor system-on-chip. Master's thesis, Eindhoven University of Technology, Eindhoven (The Netherlands), 2005.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[HPB08] Chia-Jui Hsu, José Luis Pino, and Shuvra S. Bhattacharyya. Multithreaded simulation for synchronous dataflow graphs. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 331–336, New York, NY, USA, 2008. ACM.

[Huf52] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[HW96] Junwei Hou and Wayne Wolf. Process partitioning for distributed embedded systems. In *Proceedings of the 4th International Workshop on Hardware/Software Co-Design*, CODES '96, pages 70–, Washington, DC, USA, 1996. IEEE Computer Society.

[HWO98] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 32(5):58–69, 1998.

[IB10] Harold Ishebabi and Christophe Bobda. Heuristics for flexible CMP synthesis. *IEEE Transactions on Computers*, 59:1091–1104, 2010.

[IC10] STMicroelectronics INc. and CEA. Platform 2012: A many-core programmable accelerator for ultra-efficient embedded computing in nanometer technology. White paper, November 2010. Available online (26 pages).

[IKKM07] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):186–197, 2007.

[Ins11] Texas Instruments. http://www.ti.com/ww/en/omap/omap5/omap5-omap5430.html., 2011.

[ITR05] ITRS. International techniology reoad map for semiconductors 2005. interconnect., 2005.

[ITR07] ITRS. International techniology reoad map for semiconductors 2007. system drivers., 2007.

[ITR10] ITRS. International techniology reoad map for semiconductors 2010. system drivers., 2010.

[JS91] Kevin Jeffay and Donald F. Stanat. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the 12th Real-Time Systems Symposium*, pages 129–139, 1991.

[JSKR05] Y. Jin, N. Satish, and K. Keutzer K. Ravindran. An automated exploration frame work for fpga based soft multiprocessor systems. In *Proc. of 3rd CODES+ISSS CA, USA*, volume 3199, pages 273–278, 2005.

[JSM91] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. pages 129–139, 1991.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of IFIP Congress*, pages 471–475. North-Holland Publishing Co., 1974.

[KFH+08] Akash Kumar, Shakith Fernando, Yajun Ha, Bart Mesman, and Henk Corporaal. Multiprocessor systems synthesis for multiple use-cases of multiple applications on fpga. *ACM Trans. Des. Autom. Electron. Syst.*, 13(3):1–27, 2008.

[KHHC07] Akash Kumar, Andreas Hansson, J. Huisken, and Henk Corporaal. An fpga design flow for reconfigurable network-based multiprocessor systems on chip. In *Proc. of Design Automation and Test in Europe*, page 117122. Los Alamitos, CA. IEEE Computer Society, 2007.

[KMT+08] Akash Kumar, Bart Mesman, Bart Theelen, Henk Corporaal, and Yajun Ha. Analyzing composability of applications on mpsoc platforms. *J. Syst. Archit.*, 54(3-4):369–383, 2008.

[KRD00] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: deriving process networks from matlab for embedded signal processing architectures. In *CODES*, pages 13–17, 2000.

[KRH+03] Sankaralingam Karthikeyan, Nagarajan Ramadass, Liu Haiming, Kim Changkyu, Huh Jaehyuk, Burger Doug, Keckler Stephen W., and Moore Charles R. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. *SIGARCH Comput. Archit. News*, 31(2):422–433, 2003.

[KTN+00] Mai Ken, Paaske Tim, Jayasena Nuwan, Ho Ron, Dally William J., and Horowitz Mark. Smart memories: a modular reconfigurable architecture. *SIGARCH Comput. Archit. News*, 28(2):161–171, 2000.

[Kum09] Akash Kumar. *Analysis, Design and Management of Multimedia Multiprocessor Systems*. PhD thesis, Eindhoven University of Technology, Eindhoven (The Netherlands), 2009.

[LK03] Tang Lei and Shashi Kumar. Algorithms and tools for network on chip based system design. *Integrated Circuit Design and System Design, Symposium on*, page 163, 2003.

[LM87] E.A Lee and D. G. Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. In *Proc. of IEEE Transactions on Computers*, volume 36, pages 24–35. IEEE, jan 1987.

[LSG+09] Martin Lukasiewycz, Martin Streubühr, Michael Glaß, Christian Haubelt, and Jürgen Teich. Combined system synthesis and communication architecture exploration for mpsocs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 472–477, 2009.

[LYBJ01] D. Lyonnard, S. Yoo, A. Baghdadi, and A. Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *Proc. of Design Automation and Test in Europe*, page 518523. ACM Press, New York, 2001.

[MBB+05] A. Moonen, R. V. Berg, M. Bekooij, H. Bhullar, and Jef van Meerbergen. A multi-core architecture for in-car digital entertainment. In *Proc. of GSPx Conference*, 2005.

[MBC07] T. Marescaux, E. Brockmeyer, and H. Corporaal. The impact of higher communication layers on noc supported mp-socs. In *NOCS '07: Proceedings of the First International Symposium on Networks-on-Chip*, pages 107–116, Washington, DC, USA, 2007. IEEE Computer Society.

[MBvdBvM07] Arno Moonen, Marco Bekooij, Rene van den Berg, and Jef van Meerbergen. Decoupling of computation and communication with a communication assist. In *Proc. of DSD*, pages 63–68, 2007.

[MEP07] Sorin Manolache, Petru Eles, and Zebo Peng. Fault-aware communication mapping for NoCs with guaranteed latency. *Int. J. Parallel Program.*, pages 125–156, 2007.

[MMB07] Orlando Moreira, Jacob Jan-David Mol, and Marco Bekooij. Online resource management in a multiprocessor with a network-on-chip. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1557–1564, New York, NY, USA, 2007. ACM.

[MMBM05] Orlando Moreira, Jan-David Mol, Marco Bekooij, and Jef van Meerbergen. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *RTAS '05:*, pages 332–341, 2005.

[MNTJ04] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 20890, Washington, DC, USA, 2004. IEEE Computer Society.

[Moo98] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82 –85, jan. 1998.

[MSC07] Zhe Ma, Daniele Paolo Scarpazza, and Francky Catthoor. Run-time task overlapping on multiprocessor platforms. In *ESTImedia*, pages 47–52, 2007.

[MST94] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *International Conference on Multimedia Computing and Systems*, pages 90–99, 1994.

[NAE+08] Vincent Nollet, Prabhat Avasare, Hendrik Eeckhaut, Diederik Verkest, and Henk Corporaal. Run-time management of a MP-SoC containing FPGA fabric tiles. *IEEE Trans. Very Large Scale Integr. Syst.*, pages 24–33, 2008.

[NKG+02] A. Niewland, J. Kang, O. Gangwal, R. Sethuraman, N. Busa, K. Goosens, R. Peset Llopis, and P. Lippens. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. In *Proc. of DAC*, pages 233–270, 2002.

[NSD06] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Multi-processor system design with ESPAM. In *Proc. of CODES+ISSS*, pages 211–216, 2006.

[Nvi11] Nvidia. http://www.nvidia.com/object/product-quadro-6000-us.html., 2011.

[OH02] Hyunok Oh and Soonhoi Ha. Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints. In *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, pages 133–138, New York, NY, USA, 2002. ACM.

[OWB+07] M. Oyamada, F. R. Wagner, M. Bonaciu, W. Cesario, and A. Jerraya. Software performance estimation in mpsoc design. In *ASP-DAC '07: Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pages 38–43, Washington, DC, USA, 2007. IEEE Computer Society.

[PD80] David A. Patterson and David R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8(6):25–33, 1980.

[PVA+08] Michael Pellauer, Muralidaran Vijayaraghavan, Michael Adler, Arvind, and Joel Emer. A-ports: an efficient abstraction for cycle-accurate performance models on fpgas. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 87–96, New York, NY, USA, 2008. ACM.

[RDG+04] Andrei Radulescu, John Dielissen, Kees Goossens, Edwin Rijpkema, and Paul Wielage. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network configuration. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 20878, Washington, DC, USA, 2004. IEEE Computer Society.

[SB00] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., NY, USA, 2000.

[SBGC07] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *DAC '07:*, pages 777–782, 2007.

[SBV98] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 521–532, New York, NY, USA, 1998. ACM.

[SBW09] Marcel Steine, Marco Bekooij, and Maarten Wiggers. A priority-based budget scheduler with conservative dataflow model. In *DSD '09: Proceedings of the 2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pages 37–44, Washington, DC, USA, 2009. IEEE Computer Society.

[SCT09] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Power-aware mapping of probabilistic applications onto heterogeneous mpsoc platforms. In *RTAS '09:*, pages 151–160, 2009.

[SGB06a] Sander Stuijk, Marc Geilen, and Twan Basten. SDF^3 : SDF for free. In *Application of Concurrency to System Design, ACSD 06, Proceedings. IEEE*, pages 276–278, 2006.

[SGB06b] Sander Stuijk, Marco Geilen, and Twan Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous data flow graphs. In *Proc. of Design Automation Conference*, volume 3199, pages 899–904. ACM press, New York, USA, 2006.

[SHS05] Lodewijk T. Smit, Johann L. Hurink, and Gerard J. M. Smit. Run-time mapping of applications to a heterogeneous SoC. In *in Proceedings of SoC05*, pages 78–81, 2005.

[SIAM+04] Han Sang-Il, Baghdadi Amer, Bonaciu Marius, Chae Soo-Ik, and Jerraya Ahmed A. An efficient scalable and flexible data transfer architecture for multiprocessor soc with massive distributed memory. In *DAC*, pages 250–255, New York, NY, USA, 2004. ACM.

[SKC00] Youngsoo Shin, Daehong Kim, and Kiyoung Choi. Schedulability-driven performance analysis of multiple mode embedded real-time systems. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 495–500, New York, NY, USA, 2000. ACM.

[SSK+10] Ahsan Shabbir, Sander Stuijk, Akash Kumar, Bart D. Theelen, Bart Mesman, and Henk Corporaal. A predictable communication assist. In *Conf. Computing Frontiers*, pages 97–98, 2010.

[Stu07] Sander Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Eindhoven University of Technology, 2007.

[SVM01] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. In *IEEE Design and Test of Computers*, pages 23–33, Nov/Dec 2001.

[SZT+04] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette. System design using kahn process networks: The compan/laura approach. In *Proc. of Design Automation and Test in Europe*, pages 340–345, 2004.

[TGB+06a] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Proceedings of MEMOCODE, pp 185194*, pages 185–194. IEEE Computer Society Press, 2006.

[TGB+06b] B.D. Theelen, M.C.W. Geilen, T. Basten, J.P.M. Voeten, S.V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 185–194, july 2006.

[TKD04] A. Turjan, B. Kienhuis, and E. Deprettere. An integer linear programming approach to classify the communication in process networks. In *Proc. of SCOPES*, pages 62–76, 2004.

[TKM+02] Michael Bedford Taylor, Jason Sungtae Kim, Jason E. Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matthew Frank, Saman P. Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[vB09] C. H. van Berkel. Multi-core for mobile phones. In *DATE*, pages 1260–1265, 2009.

[vdPV97] Petrus Henricus Antonius van der Putten and Jeroen Peter Marie Voeten. *Specification of reactive hardware/Software systems*. PhD thesis, Eindhoven University of Technology, 1997.

[VNS07] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. pn: A tool for improved derivation of process networks. *EURASIP J. Emb. Sys.*, 2007, 2007.

[WAHE03] Dong Wu, Bashir M. Al-Hashimi, and Petru Eles. Scheduling and mapping of conditional task graphs for the synthesis of low power embedded systems. In *DATE '03:*, pages 90–100, 2003.

[WCN+07] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. A practical fpga-based framework for novel cmp research. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 116–125, New York, NY, USA, 2007. ACM.

[Wol05] Wayne Wolf. Multimedia applications of multiprocessor systems-on-chips. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 86–89, Washington, DC, USA, 2005. IEEE Computer Society.

[WWF+06] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. Simflex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[Xil11a] Xilinx. *Microblaze Processor Reference Guide*, 2011.

[Xil11b] Xilinx. Resource pages[online], 2011.

[YC03] Peng Yang and Francky Catthoor. Pareto-optimization-based run-time task scheduling for embedded systems. In *CODES+ISSS '03:*, pages 120–125, 2003.

[YWM+01] Peng Yang, Chun Wong, Paul Marchal, Francky Catthoor, Dirk Desmet, Diederik Verkest, and Rudy Lauwereins. Energy-aware runtime scheduling for embedded-multiprocessor SOCs. *IEEE Design and Test of Computers*, pages 46–58, 2001.

[BI+95] Koray ner, Luiz A. Barroso, Sasan Iman, Jaeheon Jeong, Krishnan Ramamurthy, and Michel Dubois. The design of rpm: An fpga-based multiprocessor emulator. In *Proceedings of the 3rd ACM International Symposium on Field-Programmable Gate Arrays*, pages 60–66, 1995.

Glossary

Acronyms and abbreviations

ASIC	Application specific integrated circuit
ILP	Integer Linear Programming
CF	Compact flash
CSDF	Cyclo static dataflow
DCT	Discrete cosine transform
DSE	Design space exploration
DSP	Digital signal processing
FCFS	First-come-first-serve
FIFO	First-in-first-out
FPGA	Field-programmable gate array
FSL	Fast simplex link
HSDFG	Homogeneous synchronous dataflow graph
IDCT	Inverse discrete cosine transform
IP	Intellectual property
JPEG	Joint Photographers Expert Group
KPN	Kahn process network
LUT	Lookup table
MAMPS	Multi-Application Multi-processor Simulation
MB	Microblaze
MoC	Models of Computation
NoC	Network-on-chip
MCM	Maximum cycle mean
MPSoC	Multi-processor system-on-chip

POOSL	Parallel object oriented specification language
QoS	Quality-of-service
RAM	Random access memory
RISC	Reduced instruction set computing
RM	Resource manager
RR	Round-robin
RRWS	Round-robin with skipping
APG	Acyclic precedence graph
SADF	Scenario aware dataflow
SDFG	Synchronous dataflow graph
SMS	short messaging service
TDMA	Time-division multiple access
VLC	Variable length coding
VLD	Variable length decoding
VLIW	Very long instruction word
WCET	Worst case execution time
WCRT	Worst case response time
XML	Extensible markup language
DMA	Dynamic Memory Access

Terminology and definitions

Actor	A program segment of an application modeled as a vertex of a graph that should be executed atomically.
Control token	Some information that controls the behaviour of actor. It can determine the rate of different ports in some MoC (say SADF and BDF), and the execution time in some other MoC (say SADF and KPN).
Multimedia systems	Systems that use a combination of content forms like text, audio, video, pictures and animation to provide information or entertainment to the user.
Output actor	The last task in the execution of an application after whose execution one iteration of the application can be said to have been completed.
Rate	The number of tokens that need to be consumed (for input rate) or produced (for output rate) during an execution of an actor.
Response time	The time an actor takes to respond once it is ready i.e. the sum of its waiting and its execution time.
Scenario	A mode of operation of a particular application. For example, an MPEG video stream may be decoding an I-frame or a B-frame or a P-frame. The resource requirement in each scenario may be very different.

Scheduling	When multiple objects share a resource, a mechanism is required that shares the resource between the objects. This mechanism is called scheduling.
Task	A part of program that is executed atomically.
Token	A data/control element that is consumed or produced during an actor-execution.
Use-case/mode	This refers to a combination of applications that may be active concurrently.
Work-conserving schedule	This implies if there is work to be done (or task to be executed) on a processor, it will execute it and not wait for some other work (or task). A schedule is work-conserving when the processor is not idle as long as there is any task waiting to execute on the processor.

Curriculum Vitae

Ahsan Shabbir was born in Sialkot, Pakistan on December 11, 1974. He obtained his initial education from Sialkot and proceeded to Bangladesh for engineering studies. In 1998, he completed Bachelors in Electrical engineering (First Class) from the Bangladesh Institute of Technology Rajshahi Bangladesh (BIT). He joined National Engineering and Scientific Commison (NESCOM) as Assistant Manager in 1999. During his work in NESCOM, he also completed Masters in Computer Engineering from University of Engineering & Technology Taxila in 2003.

In 2007, he began working towards his Ph.D. degree from TUe in the Electronic Systems group. His research was funded by NESCOM. It has led, among others, to several publications and this thesis. He is also involed in EVA project at TUe. He has developed several IPs (Camera link, DDR3 controller) for EVA.

List of Publications

Journals and Book Chapters

- Ahsan Shabbir, Akash Kumar, Bart Mesman and Henk Corporaal. Enabling MPSoC Design Space Exploration on FPGAs. In *Wireless Networks, Information Processing and Systems, Communications in Computer and Information Science*, Vol. 20, pp. 412-421, ISSN: 1865-0929. Springer, 2009. doi:10.1007/978-3-540-89853-5_44.

- Ahsan Shabbir, Akash Kumar, Sander Stuijk, Bart Mesman and Henk Corporaal. CA-MPSoC: An Automated Design Flow for Predictable Multiprocessor Architectures for Multiple Applications. In: *Journal of Systems Architecture- Embedded System Design.* Vol 56, Issue 7, 2010, pp. 265-277. http://dx.doi.org/10.1016/j.sysarc.2010.03.007.

Conference Papers

- Ahsan Shabbir, Akash Kumar, Bart Mesman and Henk Corporaal. Performance Evaluation of Concurrently Executing Parallel Applications on Multi-processor Systems. In *International Conference on Embedded Computer Systems, Architecture, Modeling, and Simulation (ICSAMOS)*, 2009. pp.100-107, SAMOS, Greece.

- Ahsan Shabbir, Akash Kumar, Bart Mesman, Henk Corporaal. Distributed Simulation on FPGA for Performance Evaluation of Multiple Applications. In *Proceedings of 15th Conference of the Advanced School for Computing*

and Imaging (ASCI). Jun 2009, ISBN: 978-90-8108849-4-9. Zeewolde, The Netherlands, 2009.

- Ahsan Shabbir, Sander Stuijk, Akash Kumar, Bart D. Theelen, Bart Mesman and Henk Corporaal. A Predictable Communication Assist. In *Proceedings of Conference on Computing Frontiers (CCF)*, May 2010. pp. 97-98, ISBN: 978-1-4503-0044-5, Bertinoro, Italy, May 17-19, 2010.

- Ahsan Shabbir, Sander Stuijk, Akash Kumar, Henk Corporaal and Bart Mesman. An MPSoC Design Approach for Multiple Use-cases of Throughput Constrainted Applications. In *Proceedings of Conference on Computing Frontiers (CCF)*, May 2011. ISBN: 978-1-4503-0698-0, Ischia, Italy, May 3-5, 2011.

- Ahsan Shabbir, Akash Kumar, Bart Mesman and Henk Corporaal. Distributed Resource Management for Concurrent Execution of Multimedia Applications on MPSoC Platforms. In *International Conference on Embedded Computer Systems, Architecture, Modeling, and Simulation (IC-SAMOS)*, 2011.

Conference Papers not covered in the thesis

- Muhammad Nadeem, Stephan Wong, Georgi Kuzmanov, Ahsan Shabbir. A High-throughput, Area-efficient Hardware Accelerator for Adaptive Deblocking Filter in H.264/AVC. In *Proceedings of the 7th IEEE Workshop on Embedded systems for Real-time Multimedia (ESTIMedia 2009)*, Oct 2009. pp. 18-27. ISBN: 978-1-4244-5170-8. Grenoble, France, 15-16 October 2009.

- Muhammad Nadeem, Stephen Wong, Georgi Kuzmanov, Ahsan Shabbir, Muhammad Faisal Nadeem and Fakhar Anjam. Low-power, High-throughput Deblocking Filter for H.264/AVC. In *Proceedings of International Symposium on System-on-Chip (SoC)*, Sept 2010. pp. 93-98, ISBN: 978-1-4244-8279-5, Tampere, Finland, 29-30 Sept, 2010.

Reader's Notes