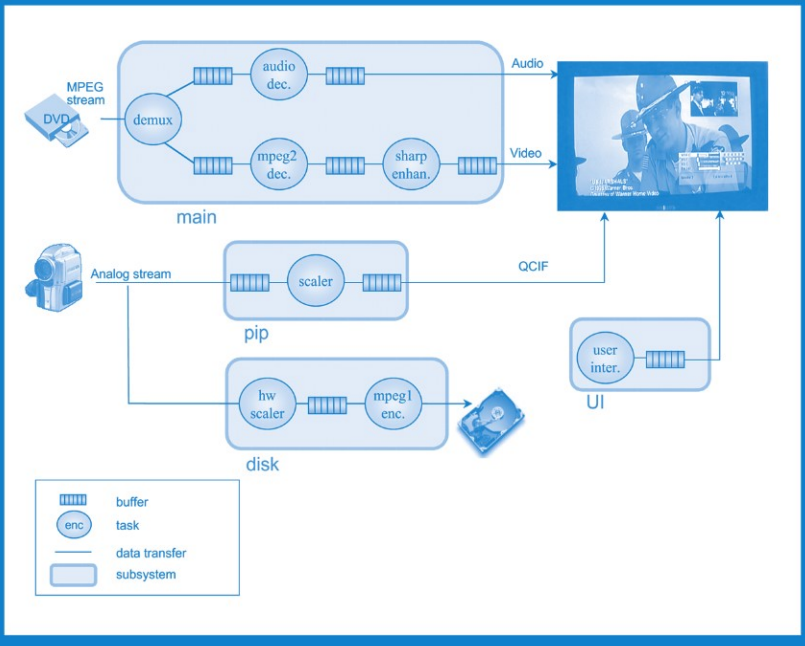
Philips Research Book Series Volume 3

Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices

Edited by
Peter van der Stok



Springer

Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices

Philips Research

VOLUME 3

Editor-in-Chief
Dr. Frank Toolenaar
Philips Research Laboratories, Eindhoven, The Netherlands

SCOPE TO THE *'PHILIPS RESEARCH BOOK SERIES'*

As one of the largest private sector research establishments in the world, Philips Research is shaping the future with technology inventions that meet peoples' needs and desires in the digital age. While the ultimate user benefits of these inventions end up on the high-street shelves, the often pioneering scientific and technological basis usually remains less visible.

This 'Philips Research Book Series' has been set up as a way for Philips researchers to contribute to the scientific community by publishing their comprehensive results and theories in book form.

Dr. Ad Huijser
Chief Executive officer of Philips Research

The titles published in this series are listed at the end of this volume.

Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices

Edited by

Peter van der Stok

Philips Research Laboratories, Eindhoven,
The Netherlands

Springer

A C.I.P. Catalogue record for this book is available from the Library of Congress.

ISBN-10 1-4020-3453-9 (HB) Springer Dordrecht, Berlin, Heidelberg, New York
ISBN-10 1-4020-3454-7 (e-book) Springer Dordrecht, Berlin, Heidelberg, New York
ISBN-13 978-1-4020-3453-4 (HB) Springer Dordrecht, Berlin, Heidelberg, New York
ISBN-13 978-1-4020-3454-1 (e-book) Springer Dordrecht, Berlin, Heidelberg, New York

Published by Springer,
P.O. Box 17, 3300 AA Dordrecht, The Netherlands.

Printed on acid-free paper

Printed in the Netherlands.

Table of Contents

Introduction, *Peter van der Stok*.....vii

Building predictable systems on chip: an analysis of guaranteed communication in the aethereal network on chip,
Om Prakash Gangwal, Andrei Rădulescu, Kees Goossens, Santiago González Pestana and Edwin Rijpkema.....1

Service-based design of systems on chip and networks on chip,
Kees Goossens, Santiago González Pestana, John Dielissen, Om Prakash Gangwal, Jef van Meerbergen, Andrei Rădulescu, Edwin Rijpkema, and Paul Wielage.....37

Cache-coherent heterogeneous multiprocessing as basis for streaming applications,
Jos van Eijndhoven, Jan Hoogerbrugge, Jayram M.N., Paul Stravers, and Andrei Terechko.....61

Dataflow analysis for real-time embedded multiprocessor system design,
Marco Bekooij, Rob Hoes, Orlando Moreira, Peter Poplavko, Milan Pastrnak, Bart Mesman, Jan David Mol, Sander Stuijk, Valentin Gheorghita, and Jef van Meerbergen.....81

Resource reservations in shared-memory multiprocessor SOCs,
Clara Otero Pérez, Martijn Rutten, Liesbeth Steffens, Jos van Eijndhoven, and Paul Stravers..... 109

Streaming in consumer products, beyond processing data,
Giel van Doren, and Bas Engel……..139

A robust component model for consumer electronic products
Hugh Maaskant……..167

Robust video streaming over wireless in-home networks,
Jeffrey Kang, Harmke de Groot, Peter van der Stok, Dmitri Jarnikov, Iulian Nitescu, and Felix Ogg……..193

Perceived quality of wirelessly transported videos,
Reinder Haakma, Dmitri Jarnikov, and Peter van der Stok.....213

Introduction

Consumer electronic (CE) devices are no longer the boxes, which operate on their own for a given fixed application (like TV watching). Instead, CE devices are becoming more complex, incorporate more functionality, and have to operate in an environment that is constantly changing. In addition, the CE devices are interconnected. Consequently, the possibility emerges to update them on the fly. Such flexibility can only be supported by realizing a large part of the CE functionality in software.

On the other hand, the latest chip technologies like Network on Chip (NoC) or Systems on Chip (SoC) incorporate more functionality on an ever-decreasing surface and volume. The CE industry is clearly at a crossroad: flexibility asks for software solutions, but performance asks for hardware solutions. Currently there is no general rule that will tell us whether a given functionality should be provided in software or hardware. This open choice is clearly reflected in this book, where both approaches are equally represented.

Within the context of ever-increasing complexity, one subject is becoming more important over the years: communication media and protocols. Gradually, homes now contain multiple PCs or digital appliances, which drives the emergence of home networks. The subject of streaming video between CE devices becomes important given the arrival of (wireless) home networks. The chips that constitute the CE devices are reaching such levels of complexity that networks are also used on chips within CE devices. Therefore, networks are used between CE devices as well as within the CE devices themselves.

Traditionally, a network operator manages the network to adapt to user wishes. In the home, no such operator is present and measures must be taken in chips and network to auto-manage the network.

Making such networks robust for user (re)configurations takes a large design effort. Both network and chips must be resilient against unexpected user behavior, perturbed communication, and unexpected inputs. The networks and chips must support a dynamic environment in which the user selects new videos, changes destinations or sources and generally does not want to be bothered by logistic issues in these networks.
This book provides a comprehensive overview of the challenges that face us. The book shows that there are many similarities between traditional networking and networks in the chip. However, different constraints lead to new trade offs and original solutions.
The book focuses on the robustness aspects of the chosen technologies in the area of video steaming. Management of resources such as memory, bandwidth, CPU cycles, bus cycles is an aspect that is prominent in many of the sections.

The first three chapters of the book discuss the essential features of the future Systems on Chips and their interconnecting Network on Chip. Chapter one describes mechanisms to guarantee throughput and latency in the NoC. The predictability of the underlying network is essential to provide predictable applications, and predictable and robust CE devices. Over-dimensioning of the chip's capacity is not always desirable because much cost is put in the prevention of conditions that happen rarely. Therefore the NoC supports both hard guarantees, and soft guarantees. The soft guarantees are provided with sufficient probability given the network load. The hard guarantees are always provided for a set of conditions, which are never violated. Chapter 2 goes in more detail into the offered communication services. It shows how the services match with the communication requirements of the possible application classes. The subject of chapter 3 is the use of memory in a SoC that contains a complete multiprocessing system. The programming of these complex devices is facilitated by providing memory models that obey a set of cache coherence requirements. The chapter explains the underlying mechanisms to guarantee the offered memory model.
The following chapter 4 complements the former three by concentrating on the analysis necessary during application design to shorten the design cycle. The chapter explains how simulation together with dataflow techniques increase the effectiveness of the analysis. Chapter 5 discusses the reduction of interference between the

individual applications. The key term is resource reservation and the guarantee that applications remain within their allotted resource bounds. Three resources are discussed in detail: the CPU, the shared cache, and the memory bus.
Chapters 6 and 7 concentrate more on the software architecture of consumer products. In chapter 6 the role of the SW platform and the required streaming infrastructure in these products is discussed. It will be shown that the tradeoffs in a streaming platform to come to a cost-effective solution puts an additional burden on the product architecture, next to the required innovative and differentiating features. From these tradeoffs, it is shown that the role of the streaming infrastructure goes beyond the traditional communication of data. Chapter 7 looks at a higher level to solve the composition problem when different versions of the same software functionality, or different software functionalities need to be combined in one product. The proposed software framework assists in reasoning on the composition and increases the probability that the final composition satisfies the global functional and extra-functional properties.
Chapters 8 and 9 tackle the problem of streaming video over a wireless medium. In section 8 it is explained how scalable video code prevents the occurrence of artifacts, which have their cause in the dependencies between the video frames generated during the encoding process. A controlled removal of parts of the video quality prevents artifacts while maintaining the playing of a lower quality enjoyable video. Section 9 investigates in more detail how the quality reduction is perceived by the users.

I want to thank all who have contributed to make this book a success. Jean Gelissen has initiated the writing of this book by pointing out the large amount of knowledge on video streaming shared by the members of the former IST sector. The support and encouragement by Eelco Dijkstra, Jaap van der Heijden and Albert van der Werf motivated the authors enormously and made writing this book feasible. Last but not least, I want to thank all authors who have put a large effort in writing original and valuable chapters.

Peter van der Stok

Chapter 1

BUILDING PREDICTABLE SYSTEMS ON CHIP: AN ANALYSIS OF GUARANTEED COMMUNICATION IN THE AETHEREAL NETWORK ON CHIP

Om Prakash Gangwal, Andrei Rădulescu, Kees Goossens,
Santiago González Pestana and Edwin Rijpkema
Philips Research Laboratories, Eindhoven, The Netherlands
{O.P.Gangwal, Andrei.Radulescu, Kees.Goossens,
Santiago.Gonzalez.Pestana, Edwin.Rijpkema}@philips.com

Abstract: As the complexity of Systems-on-Chip (SoC) is growing, meeting real-time requirements is becoming increasingly difficult. Predictability for computation, memory and communication components is needed to build real-time SoC. We focus on a predictable communication infrastructure called the Æthereal Network-on-Chip (NoC). The Æthereal NoC is a scalable communication infrastructure based on routers and network interfaces (NI). It provides two services: guaranteed throughput and latency (GT), and best effort (BE). Using the GT service, one can derive guaranteed bounds on latency and throughput. To achieve guaranteed throughput, buffers in NI must be dimensioned to hide round-trip latency and rate difference between computation and communication IPs (Intellectual Property). With the BE service, throughput and latency bounds cannot be derived with guarantees. In this chapter, we describe an analytical method to compute latency, throughput and buffering requirements for the Æthereal NoC. We show the usefulness of the method by applying it on an MPEG-2 (Moving Picture Experts Group) codec example.

Keywords: Networks-on-chip, Systems-on-chip, Time division multiplexing, Real-time systems, Predictable systems, Guaranteed throughput and latency connections, Best effort connections, Analysis and Verification of Networks-on-chip.

1. INTRODUCTION

As systems on a chip (SoC) grow in size and complexity, the current ways of system interconnect, such as buses and switches, cannot be used anymore,

P. van der Stok (ed.),
Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, 1-36.

because of, e.g., scalability and layout problems. For such complex systems, networks on chip (NoCs) have emerged as an interconnect solution. Some examples of NoC are SPIN (Adriahantenaina, Charlery, Greiner, Mortiez and Zeferino, 2003; Guerrier and Greiner, 2000), Æthereal (Goossens, van Meerbergen, Peeters and Wielage, 2002; Rădulescu, Dielissen, González Pestana, Gangwal, Rijpkema, Wielage and Goossens, 2005; Rijpkema, Goossens, Rădulescu, Dielissen, van Meerbergen, Wielage and Waterlander, 2003), Nostrum (Millberg, Nilsson, Thid and Jantsch, 2004), SoCBUS (Wiklund and Liu, 2003), QNoC (Bolotin, Cidon, Ginosar and Kolodny, 2004), aSOC (Liang, Swaminathan and Tessier, 2000), and others (Benini and De Micheli, 2001; Benini and De Micheli, 2002; Dally and Towles, 2001; Karim, Nguyen and Dey, 2002).

Most of the current interconnects, as well as NoCs have been built to offer best-effort (BE) communication services. BE communication infrastructures are not analyzable. Therefore, they require simulations to verify if the specified requirements are fulfilled. Because for complex chips, the interconnect is a central component in the system (Goossens, Gangwal, Röver and Niranjan, 2004), complete system simulations are required for system verification. Covering worst-cases for all configurations is not possible through simulations, because they are based on sample (demanding) inputs, which are never guaranteed to cover worst-case and corner cases. Problems that may appear during simulations are resolved by adjusting parameters in one or several of the many arbiters. If any change, system has to be resimulated again. There are three main problems with such systems: 1) long simulation times at each change, 2) numerous changes because of interdependences which lead to change side effects, and 3) worst-case behavior is not necessarily covered.

To solve these problems, we advocate the use of throughput and latency guarantees (Goossens et al., 2004; Goossens et al., 2002; Rijpkema et al., 2003). Each IP (Intellectual Property) module (i.e., computation and memories modules) can then be designed in isolation, because the interconnect requirements are made explicit. As the communication has a guaranteed behavior, the composed system will function according to the specifications provided all IP modules meet their specifications (correct by construction system) (Goossens et al., 2004). If IP modules have predictable behavior, the system behavior can be formally verified, without the need of simulations. If IP modules do not have predictable behavior, providing guarantees in the interconnect is still useful, because of the system compositionality resulted from offering guarantees: the system does not need to be simulated as a whole, but simulating only IP modules is enough. Moreover, there are no interdependencies, and, therefore, modifying parts of the system does not affect other parts of the system.

In this chapter, we focus on verifiable systems without a need of simulations. We define a model to characterize traffic of streaming IP modules, which

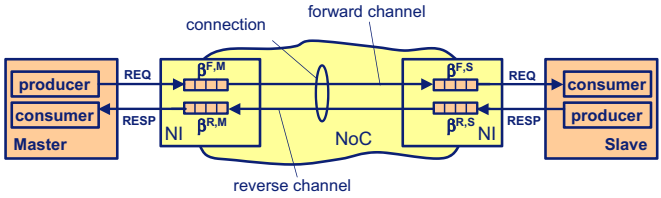


Figure 1-1. Connection example.

are characterized by the fact that they produce/consume data periodically. Using this and the Æthereal NoC's guaranteed-throughput service, we show how to compute the latency and throughput for the worst-case. We also compute the lower-bound sizes of the buffers between the NoC and the IP modules that guarantee the latency and throughput requirements are met. All these computations have been implemented in a verification tool, which is used by the Æthereal design flow (Goossens, Dielissen, Gangwal, González Pestana, Rădulescu and Rijpkema, 2005; Goossens, González Pestana, Dielissen, Gangwal, van Meerbergen, Rădulescu, Rijpkema and Wielage, 2005) to dimension and configure the NoC to satisfy the application requirements. We illustrate the use of the verification tool by applying it to an MPEG-2 (Moving Pictures Expert Group) codec example.

The chapter is organized as follows. In the next section, we describe the basics of the Æthereal NoC, focusing on the guaranteed-throughput and -latency communication services. In this section, we also define a communication model for the IP modules, and introduce some notation used in the chapter. In Section 3, we use these models to derive the throughput resulting from a given NoC for which the slots have been allocated. In Section 4, we compute the lower-bound sizes of the buffers between NoC and the IP modules. Further, in Section 5, we derive the latency that results from a given system consisting of a NoC and its attached IP modules. Our, throughput, buffer size, and latency formalizations and their implementation in a verification tool are shown in use by means of an MPEG-2 codec example in Section 6. We present our conclusions in Section 7. To ease reading, we also include in Section 8 a list of symbols used throughout the paper.

2. AN ANALYTICALLY VERIFIABLE SoC MODEL

In this section, we first describe the Æthereal NoC, focusing on the aspects that impact NoC analysis. Then, we list the conditions that IP modules need to satisfy to enable (sub)system analytical verification.

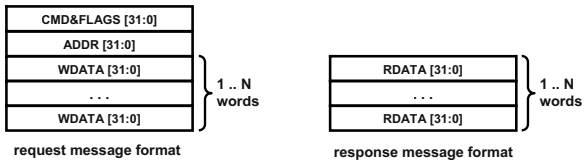


Figure 1-2. Example request and response message formats.

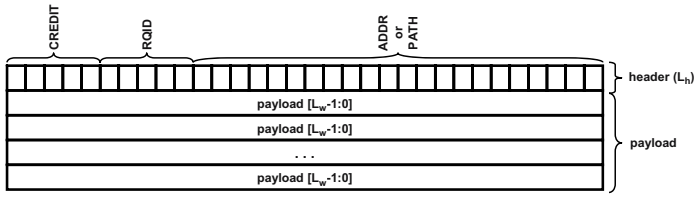


Figure 1-3. Æthereal packet format.

2.1 The Æthereal NoC Model

The Æthereal NoC (Rădulescu and Goossens, 2004; Rijpkema et al., 2003; Goossens et al., 2002) provides communication services on *connections* (Rădulescu and Goossens, 2004). As shown in Figure 1-1, connections are between two IP modules: one master, which is the module initiating the communication, and one slave which is the target module responding in the communication[1]. On each connection, we follow existing on-chip communication protocols, such as AXI, Advanced eXtensible Interface, (ARM Ltd., 2003), OCP, Open Core Protocol, (OCP International Partnership, 2003), or DTL, Device Transaction Level protocol, (Philips Semiconductors, 2002), and implement a *transaction*-based communication. That is, the masters issue *request messages*, consisting of a command (e.g., read/write), flags (e.g., burst length, mask, etc), address, and possibly write data (see Figure 1-2). Requests are transported via the NoC to the slave, which interprets and executes them, possibly issuing a *response message*, consisting of read data or acknowledgments/error flags (see Figure 1-2). In the current analysis, we use a simplified model where no acknowledgments/error flags are included.

From a NoC point of view, the request and response messages are just data which is packetized and transported over the NoC. The packet header (see Fig-

[1]More complex connections are possible, e.g., between one master and multiple slaves, but this is outside the scope of this chapter. For further information on the types of connections, please refer Rădulescu and Goossens, 2004

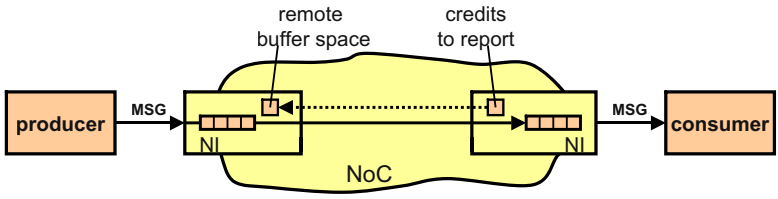


Figure 1-4. Credit-based flow control.

ure 1-3) added by the NoC has L_h words (in the current Æthereal implementation, $L_h = 1$), where the *word* is the unit of application data that is transferred on a single clock edge, and is measured in L_w number of bits (currently, in Æthereal a word consists of $L_w = 32$ bits)[2].

A connection consists of two *channels*: one *forward* channel, on which request messages are transferred, and one *reverse* channel, on which response messages are transferred. For each channel, there are two buffers that decouple IP modules from the NoC: one buffer in the network interface (NI) accepting messages in the NoC, and one buffer at the NI delivering messages to the destination IP module (see Figure 1-1). For a connection c_i, these buffers are denoted $\beta^{F,M}$, $\beta^{F,S}$, $\beta^{R,S}$ and $\beta^{R,M}$ for the buffers associated to the forward channel at master and slave sides, and those associated to the reverse channel at the slave and master sides, respectively. As shown in Figure 1-1, for each channel there is a producer (the master for a forward channel, and the slave for a reverse channel), and a consumer (the slave for a forward channel, and the master for a reverse channel).

The NoC provides *credit-based end-to-end flow control* (Tanenbaum, 1996) for every channel in the NoC. This means that at the producer's NI, there is a counter ("remote buffer space" in Figure 1-4) tracking the available space in the buffer at the consumer NI. Initially, this counter is set to the size of the consumer NI's buffer. Whenever the producer NI sends a word, the counter is decremented, and, if it reaches zero, no data is allowed to be sent to prevent buffer overflow at the consumer NI. When the consumer's NI delivers messages to the consumer IP module, another counter ("credits to report" in Figure 1-4) is incremented. This credit value needs to be sent to the producer's NI, to let

[2]A glossary of symbols is provided at the end of this chapter.

the "remote buffer space" counter correctly follow the consumer's NI buffer empty space[3].

This implies that besides application messages, credit information is also transported for every channel in the network. In our implementation, and, hence, also in our model, the credit information is transported in the packet header (see Figure 1-3). If data is transported on the same connection in the same direction in which credits must be sent, the credit is piggybacked on the created packets in the header (see Figure 1-3). If there is no data to be sent, empty packets are sent (i.e., consisting of only headers with credit information). Because of implementation reasons (fixed number of bits in the header), there is a maximum amount of credits that can be sent with one packet: M_{FC}. Consequently, the total amount of credits that can be transported on the NoC is a function of the number of packet headers that are sent.

Throughput and latency guarantees are provided using *time-division multiplexed circuits* (Rijpkema et al., 2003). Communication streams are mapped to connections, for which time *slots* in a slot table are reserved. For each slot, a circuit is set up between the producer and the consumer that communicate with each other. These circuits are dedicated to only the producer and the consumer involved, and, therefore, any interference between different connections is prevented. By using time-division multiplexing, the circuits are changed at each time slot. This allows link bandwidth to be shared between multiple connections.

To improve link utilization even further, we implement *pipelined* circuits. That is, basic units of data (i.e., the amount of data that fits in a slot) are transferred across consecutive links in consecutive slots. For example, in Figure 1-5 we show links L1, L2, L3, L4 and L5, for which slots X1, X2 and Y are reserved. Each of the slots on consecutive links are allocated consecutively (e.g., X1 has reservations in slots 1, 2, 3 and 4 for L1, L2, L3 and L4, respectively). In the bottom part of the figure, we show how data is transferred in the network following the slot reservations.

All of the NoC components (routers and NIs) run at the same frequency f_{noc} (500 MHz for Æthereal), corresponding to a clock period of $T_{noc} = 1/f_{noc}$ (2 ns for Æthereal). As already mentioned, all links have the same link width L_w. This results in a raw link bandwidth of $B_L = L_w \times f_{noc}$ (16 Gbit/s, or 500 Mwords/s).

[3] An alternative way of preventing overflow at the consumer's NI buffers is to rely on the link-level flow control. This would be possible for best-effort communication, however, data could wait in the NoC if a consumer does not consume data fast enough, and NoC congestion and/or deadlock may also occur, disturbing NoC functionality. For guaranteed communication, there is no link-level flow control, and, hence, credit-based flow control is the only way to prevent buffer overflow in the case consumer's behavior is not fully known.

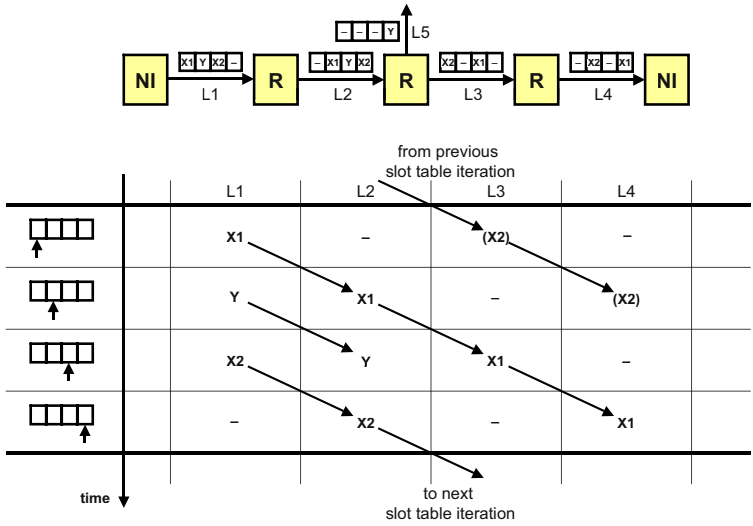


Figure 1-5. NoC with pipelined circuits using slots.

All NoC links have associated slot tables of equal sizes. We denote the slot sequence in a slot table with $\mathcal{S}$, and slot table size with $|\mathcal{S}|$. All the slots have an equal size: L_s (a slot has L_s words of L_w bits, which are transferred in $T_s = L_s \times T_{noc}$ seconds). For the Æthereal NoC, $L_s = 3$ words, and $T_s = 6$ ns. Note that a slot should be large enough to at least accommodate a complete packet header, because the packet header contains the routing information, and this information is needed to forward the slot to the correct destination ($L_s > L_h$).

Given a slot table size, we define $B_s = B_L/|\mathcal{S}|$ to be the bandwidth associated to a reserved slot, and $B_w = B_s/L_s$ to be the bandwidth associated to a reserved word.

The slot allocation and assignment of each connection c_i are stored in the network interfaces. For a connection c_i, there are two slot allocations: $\mathcal{S}_i^F \in \mathcal{S}$ and $\mathcal{S}_i^R \in \mathcal{S}$, for the forward and reverse channels, respectively.

The slot allocation for each link in the NoC is correct when (1) the number of allocated slots does not exceed the slot table size, (2) every slot of a link is allocated to at most one channel, and (3) when a channel traverses several links, the slots allocated for those links are consecutive.

2.2 The IP Module Model

On a connection c_i, IP modules are assumed to produce and/or consume data in bursts, distributed uniformly in time. That is, application bursts always come within a fixed-length periodic time interval T. As shown in Figure 1-6, we consider two cases:

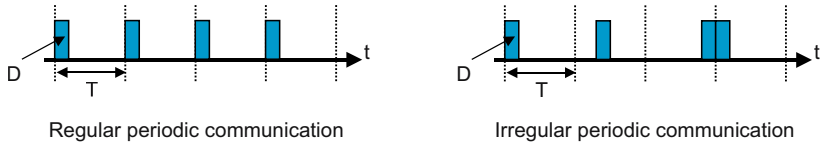


Figure 1-6. Periodic communication model.

regular, when the time between any consecutive data transfer is exactly T,

irregular, when data can be transferred at any time within a period T.

The IP module behavior is modeled by the data rate and burst size. The data rate for a connection c_i, measured in link-width words per second, is denoted for writes and reads by $R^{Wr}_{IP,i}$ and $R^{Rd}_{IP,i}$, respectively. This includes the application data (i.e., read or write data), but not the bandwidth required by the command, command flags, and address. The reason to specify only the application data is ease of specifications, as it allows to focus only on the way the application communicates, without being linked to any particular protocol or protocol implementation.

Burst sizes, denoted by $L^{Rd}_{DATA,i}$ and $L^{Wr}_{DATA,i}$ for a connection c_i, represent the amount of data that is transferred in a single read or write transaction (i.e., using a single read or write command), respectively. In our current model, a read transaction consists of a request containing a read command on the forward channel, and a response with read data on the reverse channel. A write transaction consists of a request containing a write command and write data on the forward channel, and no response on the reverse channel.

Together with the number of words needed to encode the command and its address $L^{Rd}_{CMD,i}$ and $L^{Wr}_{CMD,i}$ (2 words for the Æthereal NoC for both read and write commands), we can fully characterize the messages. For convenience, we use the command to data ratio, $\gamma^{Rd}_i = L^{Rd}_{CMD,i}/L^{Rd}_{DATA,i}$ and $\gamma^{Wr}_i = L^{Wr}_{CMD,i}/L^{Wr}_{DATA,i}$. Knowing γ^{Rd}_i and γ^{Wr}_i, we can compute the command and address rate as $R^{Rd}_{CMD,i} = R^{Rd}_{IP,i} \times \gamma^{Rd}_i$ and $R^{Wr}_{CMD,i} = R^{Wr}_{IP,i} \times \gamma^{Wr}_i$.

As an example, let us consider a connection c_i with $R^{Rd}_{IP,i} = 12$ Mwords/s, and $L^{Rd}_{DATA,i} = 16$ words. Then $\gamma^{Rd}_i = L^{Rd}_{CMD,i}/L^{Rd}_{DATA,i} = 2/16 = 0.125$. The resulting command and address rate is $R^{Rd}_{CMD,i} = R^{Rd}_{IP,i} \times \gamma^{Rd}_i = 12 \times 0.125 = 1.5$ Mwords/s.

From IP rate and burst sizes, we derive the period with which the IP module produces and/or consumes data. For a connection c_i, we assume these periods are identical for the master and slave IP modules attached to c_i. To capture the possible difference in read and write patterns on a connection, the IP periods are defined separately for reads and writes as $T^{Rd}_{IP,i} = L^{Rd}_{DATA,i}/R^{Rd}_{IP,i}$ and $T^{Wr}_{IP,i} = L^{Wr}_{DATA,i}/R^{Wr}_{IP,i}$, respectively.

Flow control is sent in the opposite direction compared to data (be it commands, addresses or application data). The credit stream must be enough to compensate for the data stream.

2.3 Notation

In this section, we further define operations and notation to help in our analysis. We use the channel type *ch* (i.e., F=forward and R=reverse) and computation type *comp* (i.e., P=producer, C=consumer, M=master, S=slave) as superscripts, and attributes to specialize the symbol *attr* (e.g. CMD=command, DATA=data, I=input, O=output) and connection index *conn_id* as subscripts for a given symbol:

$$Symbol^{ch,comp}_{attr,conn\_id} \tag{1-1}$$

We introduce two new operators: $\oplus$ and $\ominus$ for addition and subtraction modulo (denoted as %) $|\mathcal{S}|$, respectively, as follows:

$$s \oplus s' = (s + s')\%|\mathcal{S}| \tag{1-2}$$

$$s \ominus s' = (|\mathcal{S}| + s - s')\%|\mathcal{S}| \tag{1-3}$$

For each connection c_i, we define for both forward and reverse channels a set $\mathcal{F}_i^{ch}$ containing the blocks of contiguous slots allocated for that connection. A block from slots s to s' (including s and s' inclusive) is described by a tuple containing the first slot s and the block length:

$$\begin{aligned}\mathcal{F}_i^{ch} = \{\langle s, ((s' \ominus s) + 1)\rangle \quad | \quad & s, s' \in \mathcal{S}_i^{ch} \wedge \\ & (s \ominus 1) \notin \mathcal{S}_i^{ch} \wedge \\ & (s' \oplus 1) \notin \mathcal{S}_i^{ch} \wedge \\ & \forall s'',\ (s' \ominus s) \geq (s'' \ominus s), s'' \in \mathcal{S}_i^{ch}\}\end{aligned} \tag{1-4}$$

We define an "empty" set $\mathcal{E}_i^{ch}$ containing the contiguous blocks of slots not allocated to each channel of a connection c_i:

$$\begin{aligned}\mathcal{E}_i^{ch} = \{\langle s, ((s' \ominus s) + 1)\rangle \quad | \quad & s, s' \notin \mathcal{S}_i^{ch} \wedge \\ & (s \ominus 1) \in \mathcal{S}_i^{ch} \wedge \\ & (s' \oplus 1) \in \mathcal{S}_i^{ch} \wedge \\ & \forall s''\ (s' \ominus s) \geq (s'' \ominus s), s'' \notin \mathcal{S}_i^{ch}\}\end{aligned} \tag{1-5}$$

Using $\mathcal{F}_i^{ch}$, we can also define $\mathcal{H}_i^{ch}$ as the set of slots which contain headers in the case there is data sent at full rate. As in Æthereal packets correspond to blocks of slots (Rădulescu et al., 2005), at each block of slots a header is introduced:

$$\mathcal{H}_i^{ch} = \{s_h \in \mathcal{S}_i^{ch} \quad | \quad \langle s_h, \lambda\rangle \in \mathcal{F}_i^{ch}\} \tag{1-6}$$

3. THROUGHPUT ANALYSIS

The available throughput for a connection depends on the slot allocation for sending data and flow control information and the size of buffers in NIs. For this analysis, we assume that slot allocation for a connection is given. To fully utilize the available bandwidth, buffers must be dimensioned based on the analysis of Section 4 and there must be enough bandwidth available for sending flow control information.

To derive available throughput, for a given connection c_i, network specific overheads (e.g., packet header) and transaction specific overheads (e.g., command and address) need to be subtracted from the raw bandwidth based on the slot allocation, $\mathcal{S}_i$.

We first consider the case of a producer connected to a consumer through a channel *ch* with a slot allocation of $\mathcal{S}_i^{ch}$. We call $N_{h,i}^{ch}$ the number of headers introduced in one slot table iteration of a channel *ch*.

$$N_{h,i}^{ch} = |\mathcal{H}_i^{ch}| \tag{1-7}$$

Where $\mathcal{H}_i^{ch}$ denotes a set of allocated slots, where a header will be sent, for the channel *ch*.

We define $W_{r,i}^{ch}$ as the total number of words reserved for the channel in a slot table iteration. These words are divided in two categories, the first one is used to carry headers $W_{h,i}^{ch}$, and the second one is used to carry payload data[4] $W_{p,i}^{ch}$ (see Figure 1-3).

$$W_{r,i}^{ch} = |\mathcal{S}_i^{ch}| \times L_s \tag{1-8}$$

$$W_{h,i}^{ch} = N_{h,i}^{ch} \times L_h \tag{1-9}$$

$$W_{p,i}^{ch} = W_{r,i}^{ch} - W_{h,i}^{ch} \tag{1-10}$$

Based on the bandwidth associated to a reserved word B_w, we define raw bandwidth $B_{r,i}^{ch}$, header bandwidth $B_{h,i}^{ch}$, and payload bandwidth $B_{p,i}^{ch}$ for a given channel *ch*.

$$B_{r,i}^{ch} = W_{r,i}^{ch} \times B_w \tag{1-11}$$

$$B_{h,i}^{ch} = W_{h,i}^{ch} \times B_w \tag{1-12}$$

$$B_{p,i}^{ch} = W_{p,i}^{ch} \times B_w \tag{1-13}$$

The maximum flow control value that can be sent in one packet header is denoted by M_{FC} and the rate to send flow control words by $\Theta_{FC,i}^{ch} = N_{h,i}^{ch} \times B_w$.

[4]Recall that in the payload data, we include command, address, and application data (see Section 2.1 for explanation).

For a correct operation, the amount of credits sent (using flow control headers) on opposite channel (e.g. R) must be greater than or equal to the amount of data consumed by the consumer on a channel *ch* (e.g. F).

$$\Theta^{F}_{FC,i} \times M_{FC} \geq R^{R,M}_{i} \quad (1\text{-}14)$$

$$\Theta^{R}_{FC,i} \times M_{FC} \geq R^{F,S}_{i} \quad (1\text{-}15)$$

where $R^{R,M}_{i}$ and $R^{F,S}_{i}$ are the data rate of the consumer on the reverse channel and forward channel, respectively.

In the following sections, we derive exact formulas for throughput of write-only, read-only and read-write connections. Furthermore, we also provide formulas to check whether the given slot allocation meets the flow control requirements or not.

3.1 Throughput for write-only connections

For a write connection c_i, all data (including command, address, write data) are sent on the forward channel and reverse channel is used only for sending flow control data. The available data throughput for write data is derived from Equation (1-13):

$$B^{F}_{p,i} = W^{F}_{p,i} \times B_w \quad (1\text{-}16)$$

The available data $R^{Wr}_{DATA,i}$ and command $R^{Wr}_{CMD,i}$ throughput for write data (excluding packet overhead), for a given command to data ratio γ^{Wr}_{i}, is:

$$R^{Wr}_{DATA,i} = \frac{B^{F}_{p,i}}{(1+\gamma^{Wr}_{i})} \quad (1\text{-}17)$$

$$R^{Wr}_{CMD,i} = \gamma^{Wr}_{i} \times R^{Wr}_{DATA,i} \quad (1\text{-}18)$$

The specified data rates $R^{Wr}_{IP,i}$ are met when $R^{Wr}_{DATA,i} \geq R^{Wr}_{IP,i}$.

On the reverse channel, no data is sent for write transactions but flow control information (i.e., amount of data removed from buffer of the NI of consumer) needs to be sent. For a correct operation, the amount of credits sent (using a flow control header in a packet) must be greater than or equal to the amount of data consumed by the consumer. By substituting values for channel type (i.e., R) and the specified data rates in Equation (1-15), the condition is:

$$\Theta^{R}_{FC,i} \times M_{FC} \geq (1+\gamma^{Wr}_{i}) \times R^{Wr}_{IP,i} \quad (1\text{-}19)$$

3.2 Throughput for read-only connections

For a read connection c_i, commands and flow control information for read data is sent on the forward channel and on the reverse channel read data and flow control information for commands is sent.

The available data throughput for sending commands (excluding packet header overhead) is derived from Equation (1-13). As we only send commands through the forward channel, the available data throughput $B^F_{p,i}$ is fully used for commands $R^{Rd}_{CMD,i}$

$$R^{Rd}_{CMD,i} = B^F_{p,i} = W^F_{p,i} \times B_w \tag{1-20}$$

The available data throughput for sending read data (excluding packet header overhead) is derived from Equation (1-13). As we only send data through reverse channel, the available data throughput $B^R_{p,i}$ is fully used for sending data $R^{Rd}_{DATA,i}$

$$R^{Rd}_{DATA,i} = B^R_{p,i} = W^R_{p,i} \times B_w \tag{1-21}$$

For a command to data ratio of γ^{Rd}_i, the conditions when the specified data $R^{Rd}_{IP,i}$ and command rates are met are:

$$R^{Wr}_{DATA,i} \geq R^{Rd}_{IP,i} \tag{1-22}$$

$$R^{Wr}_{CMD,i} \geq \gamma^{Rd}_i \times R^{Rd}_{IP,i} \tag{1-23}$$

For a correct operation, the amount of credits sent (using flow control headers) for the forward (reverse) channel must be greater than or equal to the amount of data (command) sent in the reverse (forward) channel. By substituting the values for the data rates in Equations (1-14) and (1-15), the conditions are:

$$\Theta^F_{FC,i} \times M_{FC} \geq R^{Rd}_{IP,i} \tag{1-24}$$

$$\Theta^R_{FC,i} \times M_{FC} \geq \gamma^{Rd}_i \times R^{Rd}_{IP,i} \tag{1-25}$$

3.3 Throughput for read-write connections

For the forward path, which is used for read commands, write commands, write data, and end-to-end flow control for read data, the data rate is defined as:

$$R^{F,RdWr}_i = R^{Rd}_{CMD,i} + R^{Wr}_{CMD,i} + R^{Wr}_{DATA,i} \tag{1-26}$$

For the reverse path, read data and flow control for forward data are sent. The reverse data rate is defined as:

$$R^{R,RdWr}_i = R^{Rd}_{DATA,i} \tag{1-27}$$

For a read-write connection, the conditions when the specified data rates are met are:

$$R^{F,RdWr}_i \geq (1 + \gamma^{Wr}_i) \times R^{Wr}_{IP,i} + \gamma^{Rd}_i \times R^{Rd}_{IP,i} \tag{1-28}$$

$$R^{R,RdWr}_i \geq R^{Rd}_{IP,i} \tag{1-29}$$

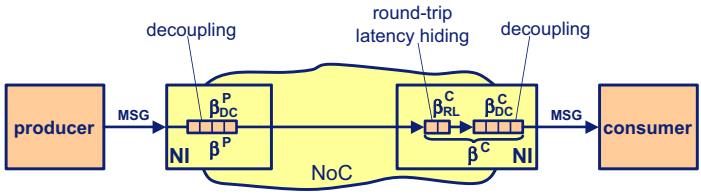


Figure 1-7. Buffers are logically split in (1) decoupling buffers β^P_{DC} and β^C_{DC}, and (2) flow-control round-trip latency hiding buffers β^C_{RL}.

For a correct operation, the amount of credits sent (using flow control headers) for the forward (reverse) channel must be greater than or equal to the amount of data (command and/or data) sent in the reverse (forward) channel. By substituting the values for the data rates in Equations (1-14) and (1-15) the conditions are:

$$\Theta^F_{FC,i} \times M_{FC} \geq R^{Rd}_{IP,i} \quad (1\text{-}30)$$

$$\Theta^R_{FC,i} \times M_{FC} \geq (1 + \gamma^{Wr}_i) \times R^{Wr}_{IP,i} + \gamma^{Rd}_i \times R^{Rd}_{IP,i} \quad (1\text{-}31)$$

4. BUFFER SIZE ANALYSIS

As shown in Figure 1-1, an Æthereal connection consists of two channels: one forward channel, and one reverse channel. Each channel has one buffer at the producer side (forward buffer at the master side, and reverse buffer at the slave side), and one buffer at the consumer side (forward buffer at the slave side, and reverse buffer at the master side). Both buffers at the producer and consumer side are used to decouple the IP blocks from the NIs, namely to hide the differences in operating frequency and communication pattern of the IP blocks and NI. Moreover, the consumer-side buffer is also used to hide the round-trip latency of reporting the flow-control credits.

For analysis purposes, we split the buffer at the consumer in two: one part for flow-control round-trip latency hiding (β^C_{RL}), and the other for decoupling (β^C_{DC}) (see Figure 1-7). In an actual implementation, for efficiency reasons, these two parts should be merged into a single buffer $\beta^C = \beta^C_{RL} + \beta^C_{DC}$. In the following two sections we describe how to compute the worst-case size for these two kinds of buffers.

4.1 Decoupling Buffers

The decoupling-buffer size computation relies on the fact that modules exchanging data exhibit a particular behavior. In our context, consisting of real-time audio/video applications, it is safe to assume that modules transfer data periodically, with an upper bound on the amount of data transferred per period.

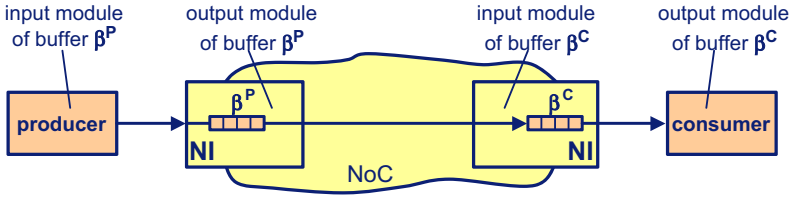


Figure 1-8. For each buffer, there are an input module (filling the buffer) and an output module (emptying the buffer).

This is, the traffic generated or consumed by a module is characterized by the following parameters (see Section 2.2 and Figure 1-6):

Period (T) is the minimum period in which a constant amount of data is sent. For a connection c_i of an IP module , T corresponds to $T^{Rd}_{IP,i}$, and/or $T^{Wr}_{IP,i}$ for read and write transactions, respectively. For an Æthereal NoC with arbitrary slot allocation on a connection c_i's channel, the period T is equal to the duration of a complete slot table rotation $|\mathcal{S}| \times T_s$. When slots are allocated equidistantly in blocks of k contiguous slots, T is taken $(|\mathcal{S}| \times T_s \times k)/|\mathcal{S}^{ch}_i|$. As shown further in this section and Section 5, a smaller period T implies smaller buffers and shorter worst-case latencies.

Data amount (D) is the upper bound on the transferred data in the given period. For the IP modules, D corresponds to the messages, and is equal to $L^{Wr}_{CMD,i} + L^{Wr}_{DATA,i}$ for write requests, $L^{Rd}_{CMD,i}$ and $L^{Rd}_{DATA,i}$ for read requests and read responses sent on a connection c_i, respectively. For a NoC with arbitrary slot allocation on a connection c_i's channel, D is equal to the number of payload words $W^{ch}_{p,i}$ transferred in a complete slot rotation. In case of equidistantly allocated blocks of k contiguous slots, $D = W^{ch}_{p,i} \times k/|\mathcal{S}^{ch}_i|$.

Regular or irregular to specify if an IP module transfers data in the same or in an arbitrary position within the interval T, respectively. IP modules can be either regular or irregular. An Æthereal NoC, however, always transfers data in the reserved slots, which do not change from a slot table rotation to another. For this reason, the NoC is always periodic over a complete slot table rotation period $|\mathcal{S}| \times T_s$. For connection c_i's channel with equidistantly allocated blocks of slots, there may be jitter in the slot allocation, and, if there is jitter, NoC will be periodic irregular over $|\mathcal{S}| \times T_s \times k/|\mathcal{S}^{ch}_i|$.

We use the same method of computing the buffer size for all the decoupling buffers, at producer and consumer sides, and for all connections. To simplify

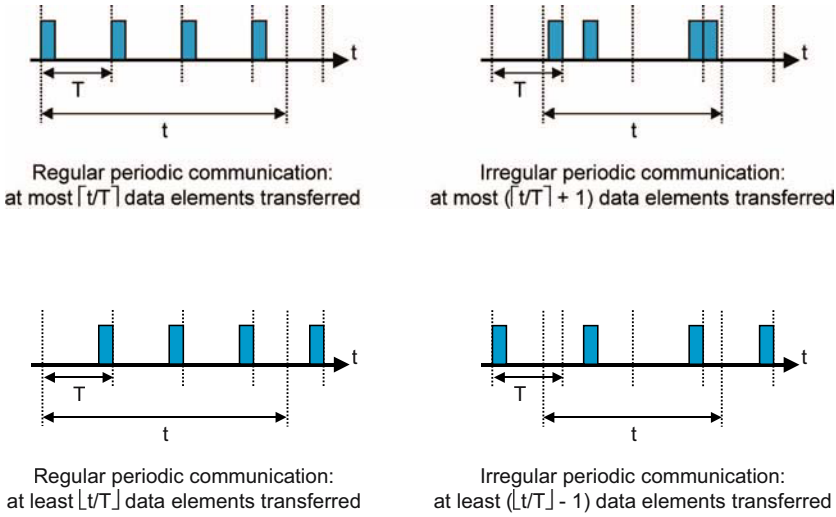


Figure 1-10. Lower bounds for periodic data transfer.

the discussion, we call the module attached to the input of the buffer the input module (producer, and NI at the consumer side), and the module attached to the output of the buffer the output module (NI at the producer side, and consumer), as shown in Figure 1-8.

Given an input module and an output module, let their periods be T_I and T_O, and the data amount per period be D_I and D_O, respectively. The maximum buffer size required between an input module M_I and an output module M_O is given by the maximum difference between the data produced by M_I and the data consumed by M_O over any time interval.

To compute this maximum difference for an arbitrary time interval of duration t, we must consider the worst-case for data production and consumption, respectively. We consider the two cases presented in Section 2.2: regular and irregular.

The worst-case data to be buffered is when the amount of produced data over an arbitrary time interval t is maximized. As shown in Figure 1-9, for the regular case, the amount of produced data is bounded by the minimum number of periods T_I that covers the time interval considered t:

$$\phi_I^R(t) \leq \left\lceil \frac{t}{T_I} \right\rceil \times D_I \tag{1-32}$$

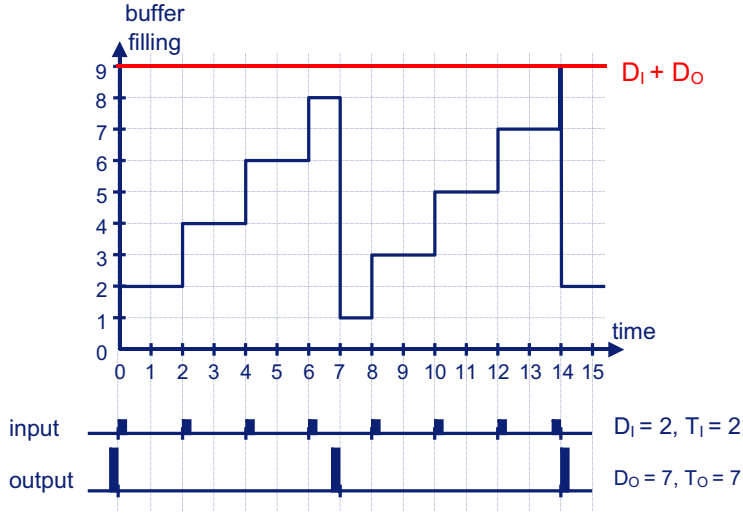


Figure 1-11. Example buffer filling for almost regular production and consumption of data.

For the irregular case, one more data item needs to be added because of the arbitrary position that the data may take inside the period:

$$\phi_I^I(t) \leq \left(\left\lceil \frac{t}{T_I} \right\rceil + 1 \right) \times D_I \tag{1-33}$$

Similarly, the worst-case data consumption occurs when the amount of consumed data over a time interval t is minimized. As shown in Figure 1-10, for the regular case, the amount of data is minimized by the number of periods T that fit in the time interval t:

$$\phi_O^R(t) \geq \left\lfloor \frac{t}{T_O} \right\rfloor \times D_O \tag{1-34}$$

For the irregular case, one more data item needs to be subtracted because of the arbitrary position that the data may take inside the period:

$$\phi_O^I(t) \geq \left(\left\lfloor \frac{t}{T_O} \right\rfloor - 1 \right) \times D_O \tag{1-35}$$

The buffer filling is a function of time t, and is less than or equal to the difference of the amount of data sent by M_I (i.e., $\phi_I(t)$), and the amount of data that can be received by M_O (i.e., $\phi_O(t)$) (see Figure 1-11 for an example). In the case M_O can consume more data than the M_I can produce, the buffer filling

is zero:

$$\phi(t) \leq \max\{\phi_I(t) - \phi_O(t), 0\} \tag{1-36}$$

For both regular production and regular consumption of data, the worst-case buffer size requirements are:

$$\phi^R R(t) \leq \max\{\phi_I^R(t) - \phi_O^R(t), 0\} \tag{1-37}$$

From Equations (1-32), (1-33), (1-34) and (1-35), and assuming the input rate is at most the output rate of the considered buffer ($D_I/T_I \leq D_O/T_O$):

$$\begin{aligned} \phi_I^R(t) - \phi_O^R(t) &\leq \left\lceil \frac{t}{T_I} \right\rceil \times D_I - \left\lfloor \frac{t}{T_O} \right\rfloor \times D_O \\ &\leq \left(\frac{t}{T_I} + 1 \right) \times D_I - \left(\frac{t}{T_O} - 1 \right) \times D_O \\ &\leq D_I + D_O + t \times \left(\frac{D_I}{T_I} - \frac{D_O}{T_O} \right) \\ &\leq D_I + D_O \end{aligned} \tag{1-38}$$

From (1-37) and (1-38):

$$\phi^{RR}(t) \leq D_I + D_O \tag{1-39}$$

In other words, the maximum buffer filling for regular periodic input and output modules is equal to the sum of the data produced D_I and consumed D_O in the periods T_I and T_O, respectively.

In the case of a NoC-based system, the buffers always reside in between an IP module (master or slave) and the NoC. The Æthereal NoC provides guaranteed-bandwidth data transfers using slot reservations in slot tables. Consequently, the NoC behavior is always regular and periodic.

In this chapter, we address the cases in which the IP module behavior is periodic and can be either regular or irregular. As the NoC is always regular and periodic, Equation (1-39) covers the case in which the IP module is periodic. For an irregular consumer, the worst-case buffer size requirements are given by:

$$\phi^{RI}(t) \leq D_I + 2 \times D_O \tag{1-40}$$

and, for an irregular IP module producer, the worst-case buffer size requirements are given by:

$$\phi^{IR}(t) \leq 2 \times D_I + D_O \tag{1-41}$$

These two equations are derived similarly to Equation (1-39).

For a connection c_i for which the throughput is guaranteed, there are a number of slots reserved in the slot table for the forward and reverse channels. As explained in Section 3, the bandwidth reserved with these slots is split in bandwidth for headers ($W_{h,i}$) and bandwidth for payload ($W_{p,i}$). In the interval given by the slot table period ($|\mathcal{S}| \times L_s/B_L$), the data is produced/consumed regularly. The NoC acts as an output module at the master side for the forward channel, and at the slave side for the reverse channel, and as an input module at the master side for the reverse channel and at the slave side for the forward channel.

Let us first consider the case when the master and slaves produce and consume data regularly. For a non-acknowledged write-only connection c_i, the decoupling buffer sizes (measured in words of L_w bits) for forward master and slave, and reverse slave and master are given by:

$$\beta_{DC,i}^{F,M} = (L_{DATA,i}^{Wr} + L_{CMD,i}^{Wr}) + W_{p,i}^{F} \quad (1\text{-}42)$$

$$\beta_{DC,i}^{F,S} = W_{p,i}^{F} + (L_{DATA,i}^{Wr} + L_{CMD,i}^{Wr}) \quad (1\text{-}43)$$

$$\beta_{DC,i}^{R,S} = 0 \quad (1\text{-}44)$$

$$\beta_{DC,i}^{R,M} = 0 \quad (1\text{-}45)$$

respectively[5].

For a read-only connection, buffer sizes are given by:

$$\beta_{DC,i}^{F,M} = L_{CMD,i}^{Rd} + W_{p,i}^{F} \quad (1\text{-}46)$$

$$\beta_{DC,i}^{F,S} = W_{p,i}^{F} + L_{CMD,i}^{Rd} \quad (1\text{-}47)$$

$$\beta_{DC,i}^{R,S} = L_{DATA,i}^{Rd} + W_{p,i}^{R} \quad (1\text{-}48)$$

$$\beta_{DC,i}^{R,M} = W_{p,i}^{R} + L_{DATA,i}^{Rd} \quad (1\text{-}49)$$

For a read-write connection, buffer sizes are a sum of the buffers for the write-only and read-only cases:

$$\beta_{DC,i}^{F,M} = (L_{DATA,i}^{Wr} + L_{CMD,i}^{Wr} + L_{CMD,i}^{Rd}) + W_{p,i}^{F} \quad (1\text{-}50)$$

$$\beta_{DC,i}^{F,S} = W_{p,i}^{F} + (L_{DATA,i}^{Wr} + L_{CMD,i}^{Wr} + L_{CMD,i}^{Rd}) \quad (1\text{-}51)$$

$$\beta_{DC,i}^{R,S} = L_{DATA,i}^{Rd} + W_{p,i}^{R} \quad (1\text{-}52)$$

$$\beta_{DC,i}^{R,M} = W_{p,i}^{R} + L_{DATA,i}^{Rd} \quad (1\text{-}53)$$

For the case the master and/or slave produce or consume data irregularly in their periods, the buffer requirements at the master/slave side double (see

[5]There is no buffer needed for the reverse channel, because, in the write-only case, there is no data being sent in the reverse channel. Slots must still be reserved for the transportation of credits, however, they are not buffered in the NI, but processed directly.

Equations (1-40) and (1-41). Let us consider, for example, the case in which both master and slave produce and consume data irregularly. For a write-only connection, buffer sizes are given by:

$$\beta^{F,M}_{DC,i} = 2 \times (L^{Wr}_{DATA,i} + L^{Wr}_{CMD,i}) + W^{F}_{p,i} \quad (1\text{-}54)$$

$$\beta^{F,S}_{DC,i} = W^{F}_{p,i} + 2 \times (L^{Wr}_{DATA,i} + L^{Wr}_{CMD,i}) \quad (1\text{-}55)$$

$$\beta^{R,S}_{DC,i} = 0 \quad (1\text{-}56)$$

$$\beta^{R,M}_{DC,i} = 0 \quad (1\text{-}57)$$

For a read-only connection, buffer sizes are given by:

$$\beta^{F,M}_{DC,i} = 2 \times L^{Rd}_{CMD,i} + W^{F}_{p,i} \quad (1\text{-}58)$$

$$\beta^{F,S}_{DC,i} = W^{F}_{p,i} + 2 \times L^{Rd}_{CMD,i} \quad (1\text{-}59)$$

$$\beta^{R,S}_{DC,i} = 2 \times L^{Rd}_{DATA,i} + W^{R}_{p,i} \quad (1\text{-}60)$$

$$\beta^{R,M}_{DC,i} = W^{R}_{p,i} + 2 \times L^{Rd}_{DATA,i} \quad (1\text{-}61)$$

For a read-write connection, buffer sizes are again a sum up of the buffers for the write-only and read-only cases:

$$\beta^{F,M}_{DC,i} = 2 \times (L^{Wr}_{DATA,i} + L^{Wr}_{DATA,i} + L^{Rd}_{CMD,i}) + W^{F}_{p,i} \quad (1\text{-}62)$$

$$\beta^{F,S}_{DC,i} = W^{F}_{p,i} + 2 \times (L^{Wr}_{DATA,i} + L^{Wr}_{CMD,i} + L^{Rd}_{CMD,i}) \quad (1\text{-}63)$$

$$\beta^{R,S}_{DC,i} = 2 \times L^{Rd}_{DATA,i} + W^{R}_{p,i} \quad (1\text{-}64)$$

$$\beta^{R,M}_{DC,i} = W^{R}_{p,i} + 2 \times L^{Rd}_{IP,i} \quad (1\text{-}65)$$

4.2 Round-Trip Latency-Hiding Buffers

The round-trip latency-hiding buffer is located in the NI at the consumer side (see Figure 1-7). It is needed to compensate for the time from which a producer NI reduces its credits when sending packets until it receives back the credits from the consumer NI. If this buffer is too small, the producer NI can run out of credits and temporarily stall its transmission of packets, and, therefore, does not meet its bandwidth requirements.

To compute the minimum size of round-trip latency-hiding buffer, we need to compute the worst-case latency necessary for the credits to be reported back to the NI from where the data is sent. The buffer size must be larger than or equal to the maximum amount of credits that can be consumed when sending data without being reported back to the sender NI.

For a connection c_i, the round-trip latencies $T^{M}_{RL,i}$ and $T^{S}_{RL,i}$ from when the data is sent by the producer NI (at master and slave, respectively) until the first

credits reach back the producing NI is given by:

$$T^{M}_{RL,i} = T^{F,T}_{L,i} + T^{R,T}_{L,i} + \max_{\langle s,\lambda\rangle \in \mathcal{E}^{R}_{i}} \lambda \tag{1-66}$$

$$T^{S}_{RL,i} = T^{R,T}_{L,i} + T^{F,T}_{L,i} + \max_{\langle s,\lambda\rangle \in \mathcal{E}^{F}_{i}} \lambda \tag{1-67}$$

where $T^{F,T}_{L,i}$ and $T^{R,T}_{L,i}$ refer to the amount of time to transport data (given by the number of hops, i.e., the number of links traversed by the packets of a channel from the producer to the consumer) on the forward and reverse channels, respectively, and $\mathcal{E}^{F}_{i}$ and $\mathcal{E}^{R}_{i}$ refer to the slots not reserved (to send flow control) for the forward and reverse channels, respectively. The first two terms represent the network latency in both directions, and the third term represents the time flow control has to wait in the NI until it can be sent. This formula represents the latency until the first credits are returned. Recall that the maximum amount of credits that can be transported in a packet is bounded to M_{FC}.

To address the case in which the amount of data transferred in the $T^{F}_{RL,i}$ and $T^{R}_{RL,i}$ intervals is larger than M_{FC}, we compute the amount of flow control that can accumulate in any time interval spanning over $0 < \delta \leq |\mathcal{S}|$ slots[6] at the destination NI without being reported back with credits (in case it is consumed)[7]:

$$\begin{aligned} \phi^{F,acc}_{i}(\delta) \;=\; & L_s \times \max_{\langle s_0,\lambda\rangle \in \mathcal{F}^{F}_{i}} \left\{ \begin{array}{l} |\,\{s \in \mathcal{S}^{F}_{i} \mid 0 < s \ominus s_0 \leq \delta\}\,| - \\ |\,\{s \in \mathcal{H}^{R}_{i} \mid 0 < s_0 \ominus s \leq \delta\}\,| \end{array} \right\} - \\ & M_{FC} \times \min_{s_0 \in \mathcal{H}^{R}_{i}} |\,\{s \in \mathcal{H}^{R}_{i} \mid 0 < s_0 \ominus s \leq \delta\}\,| \end{aligned} \tag{1-68}$$

$$\begin{aligned} \phi^{R,acc}_{i}(\delta) \;=\; & L_s \times \max_{\langle s_0,\lambda\rangle \in \mathcal{F}^{R}_{i}} \left\{ \begin{array}{l} |\,\{s \in \mathcal{S}^{R}_{i} \mid 0 < s \ominus s_0 \leq \delta\}\,| - \\ |\,\{s \in \mathcal{H}^{F}_{i} \mid 0 < s_0 \ominus s \leq \delta\}\,| \end{array} \right\} - \\ & M_{FC} \times \min_{s_0 \in \mathcal{H}^{F}_{i}} |\,\{s \in \mathcal{H}^{F}_{i} \mid 0 < s_0 \ominus s \leq \delta\}\,| \end{aligned} \tag{1-69}$$

where the first terms represent the maximum amount of data that can be transferred in the interval δ (excluding headers), and the second terms represent the minimum amount of credits that can be transported back to the sender in the same interval δ.

[6] This interval is practically a sliding window of δ slots: $[t, t + \delta \times L_s/B_L]$, where t is any time aligned to the slot boundary.

[7] For a correct slot allocation, for each channel, the maximum amount of credits that can be sent by the consumer in a slot table rotation ($M_{FC} \times |\mathcal{H}^{R}_{i}|$ and $M_{FC} \times |\mathcal{H}^{F}_{i}|$) must be larger than or equal to the amount of data produced by the producer NI in a slot table rotation ($W^{F}_{p,i}$ and $W^{R}_{p,i}$, respectively).

Let $\delta_i^{F,acc}$ and $\delta_i^{R,acc}$ be the largest interval less than a slot table rotation (i.e., $0 \leq \delta_i^{F,acc} \leq |\mathcal{S}_i^F|$ and $0 \leq \delta_i^{R,acc} \leq |\mathcal{S}_i^R|$) for which $\beta_i^{F,acc}$ and $\beta_i^{R,acc}$ are maximized, respectively. Then

$$T_{RL,i}^{M} = T_{L,i}^{F,T} + T_{L,i}^{R,T} + \delta_i^{F,acc} \tag{1-70}$$

$$T_{RL,i}^{S} = T_{L,i}^{R,T} + T_{L,i}^{F,T} + \delta_i^{R,acc} \tag{1-71}$$

represent the minimum time intervals in which the worst-case amount of data (the maximum amount of data) matches with the worst-case amount of credits (the minimum amount of credits).

The buffering to hide the round-trip latency is then:

$$\beta_{RL,i}^{F,S} = \left\lfloor \frac{T_{RL,i}^{M}}{|\mathcal{S}_i^F|} \right\rfloor \times W_{p,i}^{F} + \beta_i^{F,acc}(\delta_i^{F,acc}) \tag{1-72}$$

$$\beta_{RL,i}^{R,M} = \left\lfloor \frac{T_{RL,i}^{S}}{|\mathcal{S}_i^R|} \right\rfloor \times W_{p,i}^{R} + \beta_i^{R,acc}(\delta_i^{R,acc}) \tag{1-73}$$

The first terms represent the buffering needed to accommodate the data sent in complete table rotations (when latency is larger then a complete slot rotation), and the second terms represent the buffering needed to accommodate the maximum amount of data that can be sent in the fraction of the latency overlapping with a partial slot table rotation.

4.3 Total Buffer Sizes

As mentioned earlier, the buffer sizes in the network interfaces are given by several components: decoupling buffers and credit round-trip latency-hiding buffers. At the producer sides ($\beta_i^{F,M}$ and $\beta_i^{R,S}$), the buffers are for decoupling buffer only, while at the consumer sides ($\beta_i^{F,S}$ and $\beta_i^{R,M}$), the buffers are used for both decoupling and credit round-trip latency-hiding. As a result, the buffer sizes are given by:

$$\beta_i^{F,M} = \beta_{DC,i}^{F,M} \tag{1-74}$$

$$\beta_i^{F,S} = \beta_{DC,i}^{F,S} + \beta_{RL,i}^{F,S} \tag{1-75}$$

$$\beta_i^{R,S} = \beta_{DC,i}^{R,S} \tag{1-76}$$

$$\beta_i^{R,M} = \beta_{DC,i}^{R,M} + \beta_{RL,i}^{R,M} \tag{1-77}$$

5. LATENCY ANALYSIS

The latency of a connection c_i, $T_{L,i}$ is composed of the latency of forward channel $T_{L,i}^{F}$, reverse channel $T_{L,i}^{R}$ and IP latency $T_{L,i}^{IP}$ (see Figure 1-12). For the

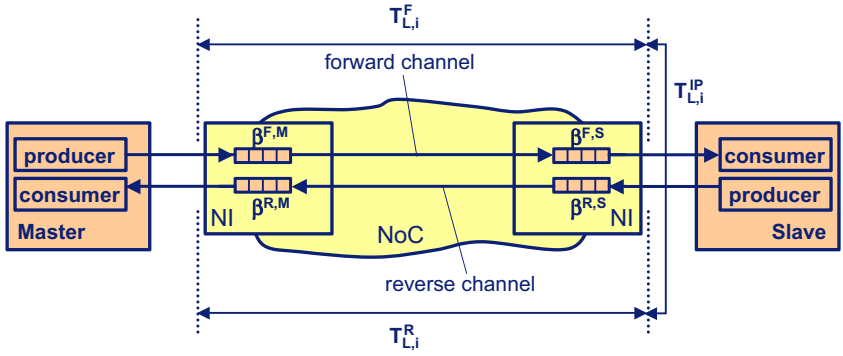


Figure 1-12. Latency of a connection.

sake of simplicity, we calculate latency in terms of slot cycles, so, in all the formulas that we derive for latency a multiplication factor of T_s must be used when converting them in terms of seconds.

We first address the general case with one producer and one consumer connected through a channel (see Figure 1-13). The latency for a channel, $T^{ch}_{L,i}$, is measured from the time a word data is accepted by the NI at producer side until the same word is accepted by the consumer. Note that the latency of a channel depends on the behavior of the consumer. We consider two cases for consumer:

Unoccupied consumer, when a consumer is ready to consume data as soon as data are offered by the NI at the consumer side.

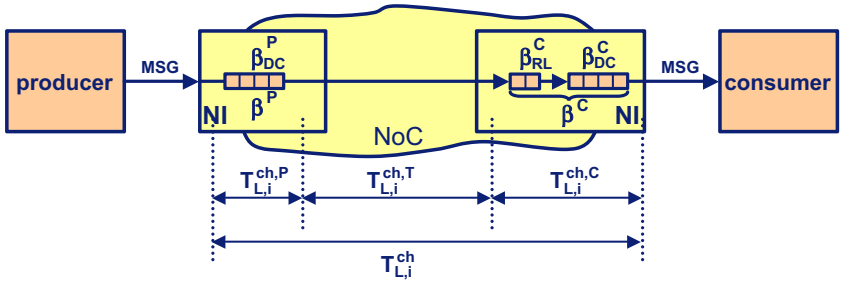


Figure 1-13. Latency of a channel.

Occupied consumer, when a consumer delays consumption of data due to, for example, sharing of the same port by many connections or when the consumer cannot, for some reason, accept the data.

In the previous section buffering values for a channel between a producer and a consumer are derived such that full utilization of bandwidth can be achieved, i.e., the producer never stalls because of lack of credits. Buffers are introduced at the producer NI (called producer buffer) as well as at consumer NI (called consumer buffer), these buffers contribute to the latency (see Figure 1-13). We split total latency for a channel $T^{ch}_{L,i}$ in three components, $T^{ch,P}_{L,i}$ the time required for the data in front of the current word to leave the producer NI, $T^{ch,T}_{L,i}$ the latency to transport a word from producer NI to the consumer NI (given by the number of hops), and $T^{ch,C}_{L,i}$ the time required for the data in front of the current word to leave the consumer NI.

$$T^{ch}_{L,i} = T^{ch,P}_{L,i} + T^{ch,T}_{L,i} + T^{ch,C}_{L,i} \tag{1-78}$$

The latency $T^{ch,P}_{L,i}$ depends on the slot allocation. For a given slot allocation the amount of data that can be removed from the producer buffer during one iteration of slot table is denoted by $W^{ch}_{p,i}$ (see Equation (1-10) on page 10). In the worst-case, the buffer is full, meaning that as many words as the buffer size $\beta^{ch,P}_{i}$ must be sent. We divide the data of the buffer into two parts, first part is an integer multiple n of $W^{ch}_{p,i}$ and the second is remainder r.

$$\beta^{ch,P}_{i} = n \times W^{ch}_{p,i} + r \tag{1-79}$$

The time $T^{ch,P}_{L_I,i}$ to send the first part of the buffer is n iterations of the slot table. However, to derive time $T^{ch,P}_{L_R,i}$ to send the remainder data r, first a function $W^{ch}_{p_{min},i}(d)$ is defined to calculate the minimum number of payload words that can be sent for a given window size d. The payload words are calculated by subtracting the number of words used for sending headers from the total number of words that can be sent for the allocated slots for the channel in the given window.

$$W^{ch}_{p_{min},i}(\delta) = \min_{s \in \mathcal{S}} \left\{ \begin{array}{l} L_s \times |\{s' \in \mathcal{S}^{ch}_{i} \mid s' \ominus s < \delta\}| - \\ |\{s' \in \mathcal{H}^{ch}_{i} \mid s' \ominus s < \delta\}| \end{array} \right\} \tag{1-80}$$

Figure 1-14 shows the number of payload words sent for the given values of δ in an example slot allocation. For a window size of 4 (i.e., $\delta = 4$) two extreme possibilities for payload p=2 and p=7 and the minimum value for payload is 2.

The set $\{\delta \in \mathbb{N} \mid W^{ch}_{p_{min},i}(\delta) \geq r\}$ defines the values of window in which at least the remaining data r can be sent. The minimum value in this set is not the

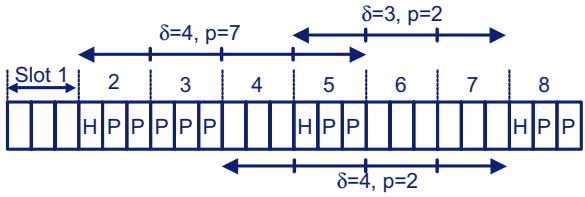


Figure 1-14. The amount of payload data p sent for a given window d for an example slot table size of 8.

worst-case delay value as it does not include all empty slots around allocated slots. For example, to send a payload of 2 (i.e., $p = 2$) the set of window is $\{3, 4, 5, 6, 7, 8\}$ (derived from the condition above) and the minimum of the set is 3 that is not the worst-case value rather the worst-case value is 4 (see Figure 1-14). So, we define an upper bound ($W^{ch}_{p_{min},i}(\delta) < (r + L_s)$) for the set such that it only allows sending the minimum payload data of remainder plus the number of words in a slot. As the minimum payload words are discrete values with the maximum step size of L_s, this upper bound also ensures that the set of window is not empty. The maximum value of the bounded set gives the worst-case delay to transfer the remaining data.

$$T^{ch,P}_{L_R,i} = \max\left\{\delta \in \mathbb{N} \mid r \leq W^{ch}_{p_{min},i}(\delta) < (r + L_s)\right\} \tag{1-81}$$

The total latency $T^{ch,P}_{L,i}$ is the sum of the integer part $T^{ch,P}_{L_I,i}$ and the remainder part $T^{ch,P}_{L_R,i}$.

$$\begin{aligned} T^{ch,P}_{L,i} &= T^{ch,P}_{L_I,i} + T^{ch,P}_{L_R,i} && (1\text{-}82)\\ &= n \times |\mathcal{S}| + \max\left\{\delta \in \mathbb{N} \mid r \leq W^{ch}_{p_{min},i}(\delta) < (r + L_s)\right\} && (1\text{-}83) \end{aligned}$$

where

$$\begin{aligned} n &= \left\lfloor \frac{\beta^{ch,P}_i}{W^{ch}_{p,i}} \right\rfloor \\ r &= \beta^{ch,P}_i \,\%\, W^{ch}_{p,i} \end{aligned}$$

$$T^{ch,T}_{L,i} = \text{Number of hops between producer NI and consumer NI.} \tag{1-84}$$

The latency for the consumer buffer, $T^{ch,C}_{L,i}$, depends on the consumption pattern of the consumer. We consider the case of an unoccupied consumer which is ready to consume data as soon as it is offered by the consumer NI. In this case, the consumer buffer would always remain empty as consumer is aggressively removing data. Hence, latency caused by the consumer buffer does not have any contributions to the total latency, (i.e., $T^{ch,C}_{L,i} = 0$).

In the case of an occupied consumer, we assume that in the worst-case a consumer consumes ($R_{IP,i} \times T^C_{IP,i}$) data words in a period $T^C_{IP,i}$ (see Section 2). $T^{ch,C}_{L,i}$ is given by the number of consumer periods needed to empty a full buffer. We convert it into the number of slot periods by dividing it by the time required to traverse a slot T_s.

$$T^{ch,C}_{L,i} = \left\lceil \frac{\beta^{ch,C}_i}{R^C_{IP,i} \times T^C_{IP,i}} \right\rceil \times \frac{T^C_{IP,i}}{T_s} \tag{1-85}$$

We derive the latencies for a given connection (read and/or write) from the latencies for a given channel in the following sections.

5.1 Latency for write-only connections

The latency $T^{Wr}_{L,i}$ for a write-only connection c_i depends only on the forward channel as data are only sent in forward direction. The latency for the *occupied consumer* case is the same as the latency of the forward channel (see Equation (1-78)).

$$T^{Wr}_{L,i} = T^F_{L,i} = T^{F,M}_{L,i} + T^{F,T}_{L,i} + T^{F,S}_{L,i} \tag{1-86}$$

For the latency of the *unoccupied consumer* case the latency introduced due to the buffer in the forward channel at the slave side $T^{F,S}_{L,i}$ is always zero as consumer keeps the buffer empty.

The specialized formulas are derived by substituting the values of channel, component type and and data rates in Equations (1-83), (1-84) and (1-85).

$$\begin{aligned}
T^{F,M}_{L,i} &= \left\lceil \frac{\beta^{F,M}_i}{W^F_{p,i}} \right\rceil \times |\mathcal{S}| + \\
&\quad \max\left\{ \delta \in \mathbb{N} \mid (\beta^{F,M}_i \% W^F_{p,i}) \leq W^F_{p_{min},i}(\delta) < ((\beta^{F,M}_i \% W^F_{p,i}) + L_s) \right\} \\
T^{F,T}_{L,i} &= \text{Number of hops in forward direction.} \\
T^{F,S}_{L,i} &= \left\lceil \frac{\beta^{F,S}_i}{(1+\gamma^{Wr}_i) \times R^{Wr}_{IP,i} \times T^{Wr}_{IP,i}} \right\rceil \times \frac{T^{Wr}_{IP,i}}{T_s}
\end{aligned}$$

5.2 Latency for read-only connections

The latency $T^{Rd}_{L,i}$ for a read-only connection c_i, depends on the both forward and reverse channel as read commands are sent in the forward direction and read data are sent in the reverse direction. We further add the latency of IP to provide responses T^{IP}_L.

$$T^{Rd}_{L,i} = T^F_{L,i} + T^{IP}_L + T^R_{L,i} \tag{1-87}$$

By substitution Equation (1-78) for each channel, $T^{Rd}_{L,i}$ is given as:

$$T^{Rd}_{L,i} = T^{F,M}_{L,i} + T^{F,T}_{L,i} + T^{F,S}_{L,i} + T^{IP}_{L} + T^{R,S}_{L,i} + T^{R,T}_{L,i} + T^{R,M}_{L,i} \tag{1-88}$$

For the *unoccupied consumer* case, there are no latency contributions from slave buffer in the forward channel (i.e., $T^{F,S}_{L,i} = 0$) and master buffer in the reverse channel (i.e., $T^{R,M}_{L,i} = 0$) as both sides can accept data as soon as it is offered by the respective NIs.

The specialized formulas are derived by substituting the values of channel, component type and data rates in Equations (1-83), (1-84) and (1-85).

$$\begin{aligned}
T^{F,M}_{L,i} &= \left\lfloor \frac{\beta^{F,M}_i}{W^F_{p,i}} \right\rfloor \times |\mathcal{S}| + \\
&\quad \max\left\{ \delta \in \mathbb{N} \mid (\beta^{F,M}_i \% W^F_{p,i}) \leq W^F_{p_{min},i}(\delta) < ((\beta^{F,M}_i \% W^F_{p,i}) + L_s) \right\} \\
T^{F,T}_{L,i} &= \text{Number of hops in forward direction.} \\
T^{F,S}_{L,i} &= \left\lfloor \frac{\beta^{F,S}_i}{\gamma^{Rd}_i \times R^{Rd}_{IP,i} \times T^{Rd}_{IP,i}} \right\rfloor \times \frac{T^{Rd}_{IP,i}}{T_s} \\
T^{IP}_{L,i} &= \text{Latency of IP to provide responses after receiving requests.} \\
T^{R,S}_{L,i} &= \left\lfloor \frac{\beta^{R,S}_i}{W^R_{p,i}} \right\rfloor \times |\mathcal{S}| + \\
&\quad \max\left\{ \delta \in \mathbb{N} \mid (\beta^{R,S}_i \% W^R_{p,i}) \leq W^R_{p_{min},i}(\delta) < ((\beta^{R,S}_i \% W^R_{p,i}) + L_s) \right\} \\
T^{F,T}_{L,i} &= \text{Number of hops in reverse direction.} \\
T^{R,M}_{L,i} &= \left\lfloor \frac{\beta^{R,M}_i}{R^{Rd}_{IP,i} \times T^{Rd}_{IP,i}} \right\rfloor \times \frac{T^{Rd}_{IP,i}}{T_s}
\end{aligned}$$

5.3 Latency for read-write connections

By substituting the buffer sizes derived for read-write case from Section 4.3 in equations defined in Section 5.1 and Section 5.2, latency for write and read transactions can be calculated.

6. VERIFICATION TOOL AND RESULTS

We have developed a tool to verify the performance of a SoC against its specifications using the analytical method described in the previous sections. This verification tool takes as input NoC attributes (e.g., slot table size, slot size, word width, and frequency), a NoC configuration per connection (e.g.,

connection id, slot table allocation for both channels, number of hops for each channel, transaction type), and the specified values for throughput, latency and buffer sizes per connection. The tool derives the required buffering, the worst-case latency, and the minimum throughput for each connection. We have developed this tool in Matlab™ and the tool produces results in XML (Extensible Markup Language) format which can be converted in HTML (Hypertext Markup Language) format.

The tool provides bounds for buffering for each connection and available slack (e.g., additional amount of buffer) from the specified values. This information allows to build correct NoCs with properly dimensioned buffers. The available slack information can be used to reduce the cost of a NoC by stripping additional buffers because the cost of a NoC is dominated by the cost of buffering (Rădulescu et al., 2005). In our tool flow (Goossens, Dielissen, Gangwal, González Pestana, Rădulescu and Rijpkema, 2005), for GT connections, we can automatically adjust buffer size to the derived bounds for each connection.

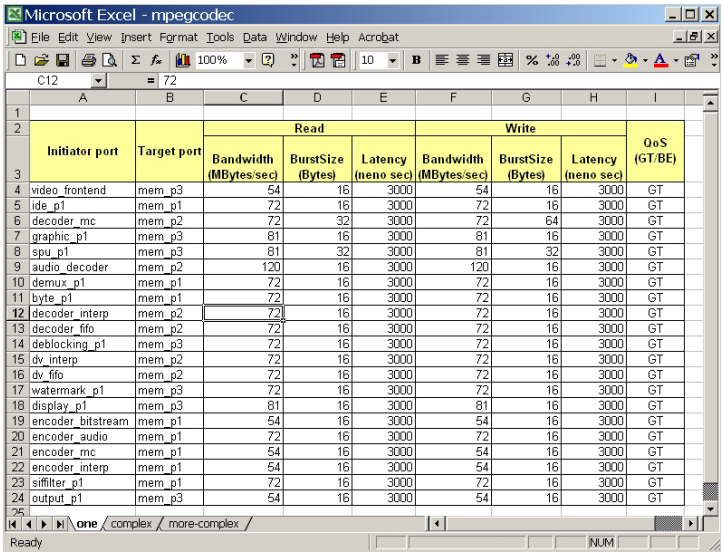
This tool derives exact values for throughput, assuming buffers are dimensioned using the derived bounds, for a given slot allocation. Furthermore, it checks whether the given slot allocation meets the specified data rates and the required flow control rates in the forward and the reverse direction. When these requirements are not met, the tool provides detailed feedback about what requirements are not met with exact numbers. Note that these verification equations have been incorporated in our slot allocation tool to build our NoCs in a correct-by-construction manner.

The tool also calculates the worst-case latency for both unoccupied consumer case and occupied consumer case, per connection basis. It also provides feedback whether we meet the specified latency requirements or not.

The tool processes one connection at a time, so, execution time of the tool is linear in number of connections in a SoC. The execution time was approximately a minute for a complex SoC with as many as 200 connections. We demonstrate the usefulness of the analysis method and the tool through an MPEG-2 codec example.

6.1 Example

The example MPEG-2 codec SoC has 16 IPs and 3 memories and 21 guaranteed throughput read-write connections (see Figure 1-15). A connection is specified between an initiator port and target port with read and/or write bandwidth requirements, burst size and latency requirements. These connections have bandwidth requirements varying from 54 to 120 Mbytes/sec and burst sizes varying from 16 to 64 bytes.



Microsoft Excel - mpegcodec

File Edit View Insert Format Tools Data Window Help Acrobat

C12 = 72

	A	B	C	D	E	F	G	H	I
1									
2			Read			Write			
3	Initiator port	Target port	Bandwidth (MBytes/sec)	BurstSize (Bytes)	Latency (neno sec)	Bandwidth (MBytes/sec)	BurstSize (Bytes)	Latency (neno sec)	QoS (GT/BE)
4	video_frontend	mem_p3	54	16	3000	54	16	3000	GT
5	ide_p1	mem_p1	72	16	3000	72	16	3000	GT
6	decoder_mc	mem_p2	72	32	3000	72	64	3000	GT
7	graphic_p1	mem_p3	81	16	3000	81	16	3000	GT
8	spu_p1	mem_p3	81	32	3000	81	32	3000	GT
9	audio_decoder	mem_p2	120	16	3000	120	16	3000	GT
10	demux_p1	mem_p1	72	16	3000	72	16	3000	GT
11	byte_p1	mem_p1	72	16	3000	72	16	3000	GT
12	decoder_interp	mem_p2	72	16	3000	72	16	3000	GT
13	decoder_fifo	mem_p2	72	16	3000	72	16	3000	GT
14	deblocking_p1	mem_p3	72	16	3000	72	16	3000	GT
15	dv_interp	mem_p2	72	16	3000	72	16	3000	GT
16	dv_fifo	mem_p2	72	16	3000	72	16	3000	GT
17	watermark_p1	mem_p3	72	16	3000	72	16	3000	GT
18	display_p1	mem_p3	81	16	3000	81	16	3000	GT
19	encoder_bitstream	mem_p1	54	16	3000	54	16	3000	GT
20	encoder_audio	mem_p1	72	16	3000	72	16	3000	GT
21	encoder_mc	mem_p1	54	16	3000	54	16	3000	GT
22	encoder_interp	mem_p1	54	16	3000	54	16	3000	GT
23	siffilter_p1	mem_p1	72	16	3000	72	16	3000	GT
24	output_p1	mem_p3	54	16	3000	54	16	3000	GT
25									

one complex more-complex

Ready NUM

Figure 1-15. Description of connections for the example MPEG-2 codec.

6.2 Analysis Results and Observations

We build two NoCs for the example MPEG-2 codec. The first is an automatically generated (using our tool flow (Goossens, Dielissen, Gangwal, González Pestana, Rădulescu and Rijpkema, 2005; Goossens, González Pestana, Dielissen, Gangwal, van Meerbergen, Rădulescu, Rijpkema and Wielage, 2005) minimum mesh topology of size 2x3 using a slot table size of 64 (called *ex64*) and the second is manually mapped and dimensioned 1x3 mesh topology using a slot table size of 8 (called *ex8*). The results for both examples are shown in Figure 1-16 and 1-17, respectively. First we describe what is shown in the result tables then we compare results for both NoCs. The key points to observe in this comparison are the effects of the size of a NoC, the size of a slot table, and the burst size on the buffer sizes, the latency, and the throughput of a connection.

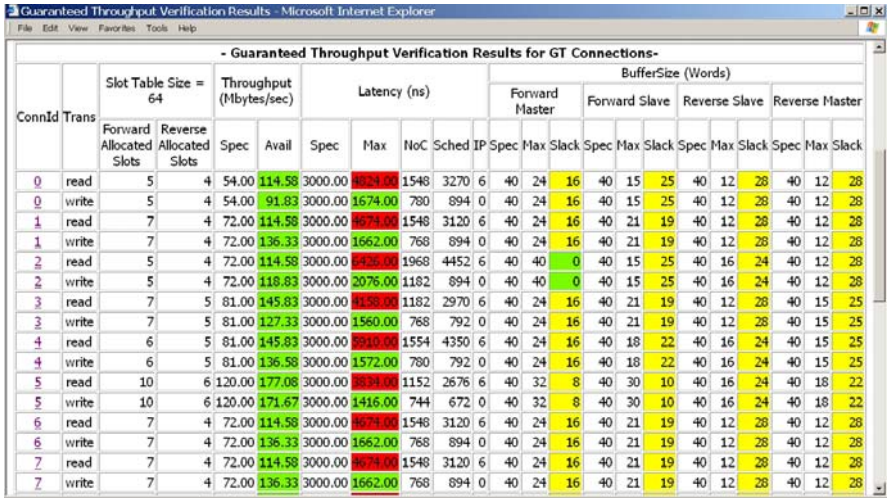
The first column of the table shows the unique connection identifiers of the connections, the second column shows the type of transactions allowed on the connection (i.e., read and/or write). The third and fourth columns provide information about slot table size and allocated number of slots for both forward and reverse channel. The fifth and sixth columns show the specified throughput values and available throughput for the given slot allocation in Mbytes per second. Notice that the available throughput is not exactly equal to the spec-

ified value rather it is usually more than the specified value because the slot allocation results in terms of integer number of slots, the available bandwidth may exceed the specified bandwidth. Note that the bandwidth per slot depends on the slot table size (i.e., the larger the slot table the smaller is the bandwidth per slot). A larger slot table provides more options to match closely with the specified throughput. For example, the available throughput for read connection 1 matches more closely to the specified value (i.e., 72 MB/s) for the ex64 architecture with 64 slots (i.e., 114.58 MB/s) than for the ex8 architecture with only 8 slots (i.e., 166.67 MB/s).

The next four columns are for the latency represented in nanoseconds. It shows the specified latency and worst-case latency including contributions from the NoC, contributions of an occupied consumer port, (i.e., *Sched* column), and the given latency of the consumer to send responses, (i.e., *IP* column) (see Figure 1-12). Note that, for low bandwidth read connections scheduling latencies dominate the unoccupied consumer latency. A connection with low bandwidth requirements gets its turn to be served by the consumer after long time (as the port of consumer is occupied with other connections of high bandwidth requirements) leading to high latency. By deriving unoccupied consumer latency and latency due to an occupied consumer port separately, one can understand which part of the latency is due to what reason. When comparing the results for the ex64 architecture with the ex8 architecture, the latency for the ex64 architecture is always larger than for the ex8 architecture due to larger buffer sizes and larger mesh for the ex64 architecture.

The rest of the columns show the specified and the computed buffer sizes, and the slack for all four buffers (forward master, forward slave, reverse slave and reverse master), respectively. The slack information tells where additional buffering is required (when it is a negative number), and where the specified buffer size is higher than needed (when it is a positive number). The buffer sizes for all buffers is larger for the ex64 architecture than the ex8 architecture due to larger NoC (mesh) and more number of allocated slots. Recalling the equations for buffering, we observe that buffer sizes are proportional to the number of allocated slots per channel. We can also back annotate these buffer sizes per connection automatically and run the analysis again. The results after running the analysis are shown in Figure 1-18.

When comparing the results for the two different topologies (ex64 and ex8) for the given example, we observe that larger slot table sizes allow a good match for the specified throughput requirements but they result in larger buffer sizes and latency. The net effect of all these is an increase in area due to the larger slot tables itself and larger buffer sizes. Figure 1-19 provides the overview of area numbers (for 0.13μ CMOS technology) for both topologies with derived buffering (i.e., buf-opt suffix) and with a fixed buffer size of 40 (i.e., buf-no-opt suffix). As expected router cost is higher for ex64 topology as



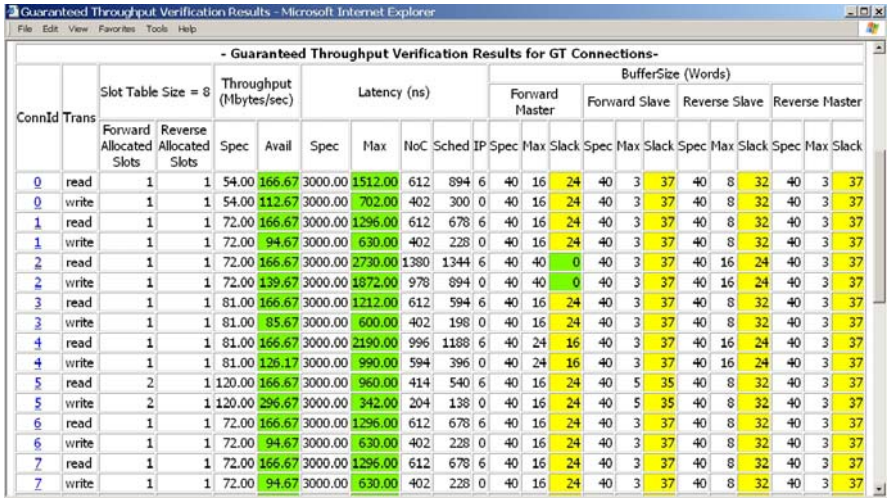
Guaranteed Throughput Verification Results - Microsoft Internet Explorer

File Edit View Favorites Tools Help

- Guaranteed Throughput Verification Results for GT Connections-

ConnId	Trans	Slot Table Size = 64		Throughput (Mbytes/sec)		Latency (ns)					BufferSize (Words)											
											Forward Master			Forward Slave			Reverse Slave			Reverse Master		
		Forward Allocated Slots	Reverse Allocated Slots	Spec	Avail	Spec	Max	NoC	Sched	IP	Spec	Max	Slack	Spec	Max	Slack	Spec	Max	Slack	Spec	Max	Slack
0	read	5	4	54.00	114.58	3000.00	4824.00	1548	3270	6	40	24	16	40	15	25	40	12	28	40	12	28
0	write	5	4	54.00	91.83	3000.00	1674.00	780	894	0	40	24	16	40	15	25	40	12	28	40	12	28
1	read	7	4	72.00	114.58	3000.00	4674.00	1548	3120	6	40	24	16	40	21	19	40	12	28	40	12	28
1	write	7	4	72.00	136.33	3000.00	1662.00	768	894	0	40	24	16	40	21	19	40	12	28	40	12	28
2	read	5	4	72.00	114.58	3000.00	6426.00	1968	4452	6	40	40	0	40	15	25	40	16	24	40	12	28
2	write	5	4	72.00	118.83	3000.00	2076.00	1182	894	0	40	40	0	40	15	25	40	16	24	40	12	28
3	read	7	5	81.00	145.83	3000.00	4158.00	1182	2970	6	40	24	16	40	21	19	40	12	28	40	15	25
3	write	7	5	81.00	127.33	3000.00	1560.00	768	792	0	40	24	16	40	21	19	40	12	28	40	15	25
4	read	6	5	81.00	145.83	3000.00	5910.00	1554	4350	6	40	24	16	40	18	22	40	16	24	40	15	25
4	write	6	5	81.00	136.58	3000.00	1572.00	780	792	0	40	24	16	40	18	22	40	16	24	40	15	25
5	read	10	6	120.00	177.08	3000.00	3834.00	1152	2676	6	40	32	8	40	30	10	40	16	24	40	18	22
5	write	10	6	120.00	171.67	3000.00	1416.00	744	672	0	40	32	8	40	30	10	40	16	24	40	18	22
6	read	7	4	72.00	114.58	3000.00	4674.00	1548	3120	6	40	24	16	40	21	19	40	12	28	40	12	28
6	write	7	4	72.00	136.33	3000.00	1662.00	768	894	0	40	24	16	40	21	19	40	12	28	40	12	28
7	read	7	4	72.00	114.58	3000.00	4674.00	1548	3120	6	40	24	16	40	21	19	40	12	28	40	12	28
7	write	7	4	72.00	136.33	3000.00	1662.00	768	894	0	40	24	16	40	21	19	40	12	28	40	12	28

Figure 1-16. Results of GT verification for the ex64 architecture.



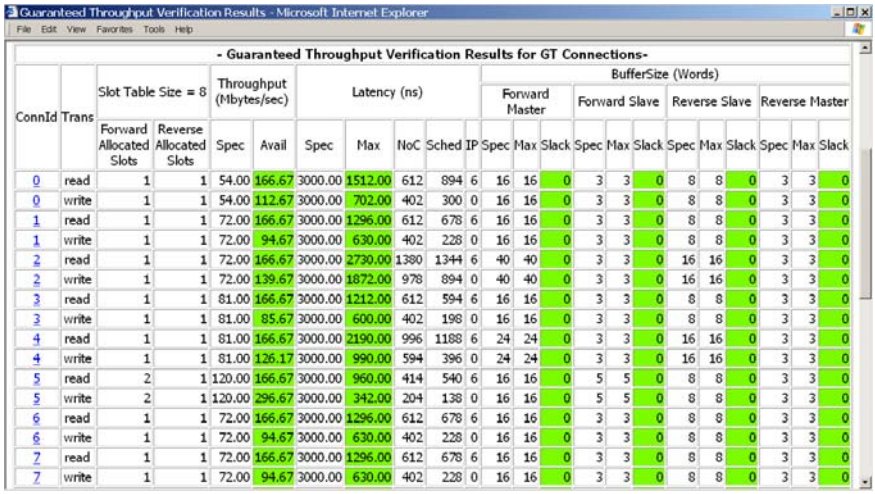
Guaranteed Throughput Verification Results - Microsoft Internet Explorer

File Edit View Favorites Tools Help

- Guaranteed Throughput Verification Results for GT Connections-

ConnId	Trans	Slot Table Size = 8		Throughput (Mbytes/sec)		Latency (ns)					BufferSize (Words)											
											Forward Master			Forward Slave			Reverse Slave			Reverse Master		
		Forward Allocated Slots	Reverse Allocated Slots	Spec	Avail	Spec	Max	NoC	Sched	IP	Spec	Max	Slack	Spec	Max	Slack	Spec	Max	Slack	Spec	Max	Slack
0	read	1	1	54.00	166.67	3000.00	1512.00	612	894	6	40	16	24	40	3	37	40	8	32	40	3	37
0	write	1	1	54.00	112.67	3000.00	702.00	402	300	0	40	16	24	40	3	37	40	8	32	40	3	37
1	read	1	1	72.00	166.67	3000.00	1296.00	612	678	6	40	16	24	40	3	37	40	8	32	40	3	37
1	write	1	1	72.00	94.67	3000.00	630.00	402	228	0	40	16	24	40	3	37	40	8	32	40	3	37
2	read	1	1	72.00	166.67	3000.00	2730.00	1380	1344	6	40	40	0	40	3	37	40	16	24	40	3	37
2	write	1	1	72.00	139.67	3000.00	1872.00	978	894	0	40	40	0	40	3	37	40	16	24	40	3	37
3	read	1	1	81.00	166.67	3000.00	1212.00	612	594	6	40	16	24	40	3	37	40	8	32	40	3	37
3	write	1	1	81.00	85.67	3000.00	600.00	402	198	0	40	16	24	40	3	37	40	8	32	40	3	37
4	read	1	1	81.00	166.67	3000.00	2190.00	996	1188	6	40	24	16	40	3	37	40	16	24	40	3	37
4	write	1	1	81.00	126.17	3000.00	990.00	594	396	0	40	24	16	40	3	37	40	16	24	40	3	37
5	read	2	1	120.00	166.67	3000.00	960.00	414	540	6	40	16	24	40	5	35	40	8	32	40	3	37
5	write	2	1	120.00	296.67	3000.00	342.00	204	138	0	40	16	24	40	5	35	40	8	32	40	3	37
6	read	1	1	72.00	166.67	3000.00	1296.00	612	678	6	40	16	24	40	3	37	40	8	32	40	3	37
6	write	1	1	72.00	94.67	3000.00	630.00	402	228	0	40	16	24	40	3	37	40	8	32	40	3	37
7	read	1	1	72.00	166.67	3000.00	1296.00	612	678	6	40	16	24	40	3	37	40	8	32	40	3	37
7	write	1	1	72.00	94.67	3000.00	630.00	402	228	0	40	16	24	40	3	37	40	8	32	40	3	37

Figure 1-17. Results of GT verification for the ex8 architecture.



Guaranteed Throughput Verification Results - Microsoft Internet Explorer

File Edit View Favorites Tools Help

- Guaranteed Throughput Verification Results for GT Connections-

ConnId	Trans	Slot Table Size = 8		Throughput (Mbytes/sec)		Latency (ns)					BufferSize (Words)											
											Forward Master			Forward Slave			Reverse Slave			Reverse Master		
		Forward Allocated Slots	Reverse Allocated Slots	Spec	Avail	Spec	Max	NoC	Sched	IP	Spec	Max	Slack	Spec	Max	Slack	Spec	Max	Slack	Spec	Max	Slack
0	read	1	1	54.00	166.67	3000.00	1512.00	612	894	6	16	16	0	3	3	0	8	8	0	3	3	0
0	write	1	1	54.00	112.67	3000.00	702.00	402	300	0	16	16	0	3	3	0	8	8	0	3	3	0
1	read	1	1	72.00	166.67	3000.00	1296.00	612	678	6	16	16	0	3	3	0	8	8	0	3	3	0
1	write	1	1	72.00	94.67	3000.00	630.00	402	228	0	16	16	0	3	3	0	8	8	0	3	3	0
2	read	1	1	72.00	166.67	3000.00	2730.00	1380	1344	6	40	40	0	3	3	0	16	16	0	3	3	0
2	write	1	1	72.00	139.67	3000.00	1872.00	978	894	0	40	40	0	3	3	0	16	16	0	3	3	0
3	read	1	1	81.00	166.67	3000.00	1212.00	612	594	6	16	16	0	3	3	0	8	8	0	3	3	0
3	write	1	1	81.00	85.67	3000.00	600.00	402	198	0	16	16	0	3	3	0	8	8	0	3	3	0
4	read	1	1	81.00	166.67	3000.00	2190.00	996	1188	6	24	24	0	3	3	0	16	16	0	3	3	0
4	write	1	1	81.00	126.17	3000.00	990.00	594	396	0	24	24	0	3	3	0	16	16	0	3	3	0
5	read	2	1	120.00	166.67	3000.00	960.00	414	540	6	16	16	0	5	5	0	8	8	0	3	3	0
5	write	2	1	120.00	296.67	3000.00	342.00	204	138	0	16	16	0	5	5	0	8	8	0	3	3	0
6	read	1	1	72.00	166.67	3000.00	1296.00	612	678	6	16	16	0	3	3	0	8	8	0	3	3	0
6	write	1	1	72.00	94.67	3000.00	630.00	402	228	0	16	16	0	3	3	0	8	8	0	3	3	0
7	read	1	1	72.00	166.67	3000.00	1296.00	612	678	6	16	16	0	3	3	0	8	8	0	3	3	0
7	write	1	1	72.00	94.67	3000.00	630.00	402	228	0	16	16	0	3	3	0	8	8	0	3	3	0

Figure 1-18. Results of GT verification with derived buffer sizes for the ex8 architecture.

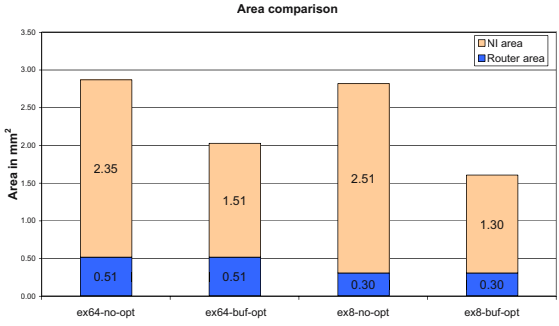


Figure 1-19. Comparison of various topologies and buffering.

compared to ex8. It is clear from Figure 1-19 that cost of NI dominates the cost of network. The cost of the network can thus be reduced by carefully choosing the network parameters (e.g., slot table size and number of routers and their connections to IPs).

Our technique allows analysis of each GT connection independently leading to a composable design. The complete design is analyzable without the use of any simulation techniques.

7. CONCLUSIONS

To build predictable systems all components of the system must be predictable. This includes computation components, memory components and communication components. Our focus is on predictable communication components (i.e., Æthereal NoC). We explain that guaranteed services are required to do worst-case analysis of a NoC without performing time consuming simulations that may not cover the worst-case, anyways. An analyzable communication infrastructure is a must to build correct-by-construction predictable systems. We derived bounds for worst-case values for buffer sizes, latency, and throughput for the Æthereal NoC. We show how these techniques can be applied using an MPEG-2 codec example. Furthermore, we show that the cost of the communication infrastructure can be reduced with derived values for buffer sizes and quick exploration of various topologies through our analysis without a need of simulation.

8. GLOSSARY

Symbol	Brief description (subscript *i* denotes a connection identifier)	Page
$\beta_{DC,i}^{F,M}$	Decoupling buffer size for forward channel at master side, in words	18
$\beta_{DC,i}^{F,S}$	Decoupling buffer size for forward channel at slave side, in words	18
$\beta_{DC,i}^{R,M}$	Decoupling buffer size for reverse channel at master side, in words	18
$\beta_{DC,i}^{R,S}$	Decoupling buffer size for reverse channel at slave side, in words	18
$\beta_{RL,i}^{F,S}$	Buffer size needed to hide the round-trip latency delay of the flow control for the forward channel at the slave side, in words	21
	Continued on next page ...	

Symbol	Brief description	Page
$\beta_{RL,i}^{R,M}$	Buffer size needed to hide the round-trip latency delay of the flow control for the reverse channel at the master side, in words	21
$\beta_i^{F,M}$	Total buffer size at the master side of the forward channel, in words	21
$\beta_i^{F,S}$	Total buffer size at the slave side of the forward channel, in words	21
$\beta_i^{R,M}$	Total buffer size at the master side of the reverse channel, in words	21
$\beta_i^{R,S}$	Total buffer size at the slave side of the reverse channel, in words	21
γ_i^{Wr}	Command to data ratio for a write connection	8
γ_i^{Rd}	Command to data ratio for a for a read connection	8
$\Theta_{FC,i}^{ch}$	Flow control symbol rate for channel *ch*, in Symbol/sec	10
B_L	Raw link bandwidth, in words/sec	6
$B_{h,i}^{ch}$	Header bandwidth for channel *ch*, in words/sec	10
$B_{p,i}^{ch}$	Payload bandwidth for channel *ch*, in words/sec	10
$B_{r,i}^{ch}$	Total raw bandwidth for channel *ch*, in words/sec	10
B_s	The bandwidth associated to a reserved slot, in words/sec	7
B_w	The bandwidth associated to a reserved word, in words/sec	7
$\mathcal{E}_i^{ch}$	Blocks of contiguous slots not allocated to channel *ch*	9
$\mathcal{F}_i^{ch}$	Blocks of contiguous slots allocated to channel *ch*	9
f_{noc}	Frequency of a NoC, in Hz	6
$\mathcal{H}_i^{ch}$	Slots containing headers for channel *ch*	9
L_{CMD}	Number of words used to encode the command and address in a message	8
L_{DATA}	The amount of words that is transferred in a single transaction	8
L_h	Packet header length, in words	5
L_s	Slot size, in words	7
L_w	Word length, in bits	5

Continued on next page ...

Symbol	Brief description	Page
M_{FC}	The maximum flow control value that can be sent in one header (or symbol)	6
$R^{Rd}_{CMD,i}$	The available data throughput for read commands, in words/sec	12
$R^{Wr}_{CMD,i}$	The available data throughput for write commands, in words/sec	11
$R^{Rd}_{DATA,i}$	The available data throughput for read data, in words/sec	12
$R^{Wr}_{DATA,i}$	The available data throughput for write data, in words/sec	11
$R^{Rd}_{IP,i}$	The specified data throughput for a read connection, in words/sec	8
$R^{Wr}_{IP,i}$	The specified data throughput for a write connection, in words/sec	8
$\mathcal{S}$	Sequence of slots in a slot table	7
$\|\mathcal{S}\|$	Slot table size	7
$\mathcal{S}^{ch}_i$	Sequence of slots allocated to channel *ch*	7
$T^{Rd}_{IP,i}$	The period with which an IP module issues/processes read commands, in seconds	8
$T^{Wr}_{IP,i}$	The period with which an IP module issues/processes write commands, in seconds	8
T^{IP}_{L}	IP latency to provide responses after a request is made, in seconds	25
$T_{L,i}$	Latency of a connection c_i, in seconds	21
$T^{F,T}_{L,i}$	Latency to transport data in the forward channel, in seconds (given by the number of hops)	20
$T^{R,T}_{L,i}$	Latency to transport data in the reverse channel, in seconds (given by the number of hops)	20
$T^{ch}_{L,i}$	Latency of a channel *ch* in seconds	22
$T^{Rd}_{L,i}$	Latency of a read-only connection, in seconds	26
$T^{Wr}_{L,i}$	Latency of a write-only connection, in seconds	25
T_{noc}	Clock period of a NoC, in seconds	6
T_s	Time required to traverse a slot, in seconds	7
$W^{ch}_{h,i}$	Number of header words sent in one iteration of	10

Continued on next page ...

Symbol	Brief description	Page
	the slot table for channel *ch*	
$W_{p,i}^{ch}$	Number of payload words sent in one iteration of the slot table for channel *ch*	10
$W_{r,i}^{ch}$	Total number of words sent in one iteration of the slot table for channel *ch*	10

References

Adriahantenaina, A., Charlery, H., Greiner, A., Mortiez, L. and Zeferino, C. A., 2003, SPIN: A scalable, packet switched, on-chip micro-network, *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*.

ARM Ltd., 2003, *AMBA AXI Protocol Specification*.

Benini, L. and De Micheli, G., 2001, Powering networks on chips, *Int'l Symposium on System Synthesis (ISSS)*, pp. 33–38.

Benini, L. and De Micheli, G., 2002, Networks on chips: A new SoC paradigm, *IEEE Computer* **35**(1), 70–80.

Bolotin, E., Cidon, I., Ginosar, R. and Kolodny, A., 2004, QNoC: QoS architecture and design process for network on chip, *Journal of Systems Architecture* **50**(2–3), 105–128. Special issue on Networks on Chip.

Dally, W. J. and Towles, B., 2001, Route packets, not wires: on-chip interconnection networks, *Proc. Design Automation Conference (DAC)*, pp. 684–689.

Goossens, K., Dielissen, J., Gangwal, O. P., González Pestana, S., Rădulescu, A. and Rijpkema, E., 2005, A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification, *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*.

Goossens, K., Gangwal, O. P., Röver, J. and Niranjan, A. P., 2004, Interconnect and memory organization in SOCs for advanced set-top boxes and TV — Evolution, analysis, and trends, *in* J. Nurmi, H. Tenhunen, J. Isoaho and A. Jantsch (eds), *Interconnect-Centric Design for Advanced SoC and NoC*, Kluwer, chapter 15, pp. 399–423.

Goossens, K., González Pestana, S., Dielissen, J., Gangwal, O. P., van Meerbergen, J., Rădulescu, A., Rijpkema, E. and Wielage, P., 2005, Service-based design of systems on chip and networks on chip, *in* P. van der

Stok (ed.), *Dynamic and Robust Streaming in and Between Connected Consumer-Electronic Devices*, Kluwer.

Goossens, K., van Meerbergen, J., Peeters, A. and Wielage, P., 2002, Networks on silicon: Combining best-effort and guaranteed services, *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 423–425.

Guerrier, P. and Greiner, A., 2000, A generic architecture for on-chip packet-switched interconnections, *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 250–256.

Karim, F., Nguyen, A. and Dey, S., 2002, An interconnect architecture for networking systems on chips, *IEEE Micro* **22**(5), 36–45.

Liang, J., Swaminathan, S. and Tessier, R., 2000, aSOC: A scalable, single-chip communications architecture, *Proc. Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*.

Millberg, M., Nilsson, E., Thid, R. and Jantsch, A., 2004, Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip, *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*.

OCP International Partnership, 2003, *Open Core Protocol Specification. 2.0 Release Candidate*.

Rădulescu, A., Dielissen, J., González Pestana, S., Gangwal, O. P., Rijpkema, E., Wielage, P. and Goossens, K., 2005, An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming, *IEEE Transactions on CAD of Integrated Circuits and Systems* **24**(1).

Rădulescu, A. and Goossens, K., 2004, Communication services for networks on chip, *in* S. S. Bhattacharyya, E. F. Deprettere and J. Teich (eds), *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, Marcel Dekker, pp. 193–213.

Rijpkema, E., Goossens, K. G. W., Rădulescu, A., Dielissen, J., van Meerbergen, J., Wielage, P. and Waterlander, E., 2003, Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip, *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 350–355.

Philips Semiconductors, 2002, *Device Transaction Level (DTL) Protocol Specification. Version 2.2*.

Tanenbaum, A. S., 1996, *Computer Networks*, Prentice-Hall.

Wiklund, D. and Liu, D., 2003, Socbus: switched network on chip for hard real time embedded systems, *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*.

Chapter 2

SERVICE-BASED DESIGN OF SYSTEMS ON CHIP AND NETWORKS ON CHIP

Kees Goossens, Santiago González Pestana,
John Dielissen, Om Prakash Gangwal, Jef van Meerbergen,
Andrei Rădulescu, Edwin Rijpkema, and Paul Wielage
Philips Research Laboratories, Eindhoven, The Netherlands
{ Kees.Goossens,Santiago.Gonzalez.Pestana } @Philips.com

Abstract: We discuss why performance verification of systems on chip (SOC) is difficult, by means of an example. We identify four reasons why building SOCs with predictable performance is difficult: unpredictable resource usage, variable resource performance, resource sharing, and interdependent resources. We then introduce the concept of a service, aiming to address these problems, and describe its advantages over "ad-hoc" approaches. Finally, we introduce the ÆTHEREAL network on chip (NOC) as a concrete example of a communication resource that implements multiple service levels.

Keywords: System design, embedded system, system architecture, real time, network on chip, quality of service, performance analysis, best effort.

1. INTRODUCTION

Moore's Law results in increasing computational power, which enables sophisticated functions to be incorporated in ever-smaller devices. Consumer electronics is shifting from discrete tethered devices to pervasive systems embedded in every-day objects. The increased interaction with the real world (e.g. managing the intelligent home, as opposed to e.g. a stand-alone personal computer) requires real-time reactions, a high degree of reliability, and, for user comfort, predictable behaviour.

Moreover, as the computational power of these systems grows, more advanced algorithms are introduced, such as MPEG4 (moving-picture experts group) and 3D graphics. These algorithms make use of the increased flexibility (software-programmability) of embedded systems. Combining variable

P. van der Stok (ed.),
Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, 37-60.

resource requirements (computation, storage, and communication) with the robust and predictable behaviour required by embedded consumer-electronics devices is the challenge that we address in this chapter.

The embedded systems just described are implemented using one or more chips, which together contain one or more *systems on a chip* (SoC). SoCs are composed of hardware components (*intellectual property*, or IP), which are interconnected by a communication infrastructure, here assumed to be a *network on a chip* (NoC).

Designing a SoC is an expensive undertaking, requiring large hardware and software design teams. The bulk of the effort of SoC design resides not in the design of the IP, but in their composition or integration into a larger, working whole. Verifying that the ensemble of IP behaves correctly with the required functionality and real-time performance is the bottle neck in SoC design.

In this chapter we advocate that the notion of *services* can ease system design. Computation, communication, and storage services enable the construction of modular SoCs, allowing compositional verification.

Overview In the following section we describe and analyse the problem of performance verification of SoCs. Building on the notions of resources, their usage and performance, we show that unpredictable resource usage, variable resource performance, and resource sharing, complicate the construction of predictable systems. In Section 3 we define the service concept, describe its advantages over "ad-hoc" approaches, and show how it addresses the performance verification problems identified earlier. In Section 4 we describe a concrete application of the concepts. The ÆTHEREAL NoC implements two communication service levels (Goossens, Dielissen, van Meerbergen, Poplavko, Rădulescu, Rijpkema, Waterlander and Wielage, 2003; Rădulescu and Goossens, 2004). We describe their implementation, intended usage, and how they tackle the problems listed above. In Section 5 we reflect and conclude.

2. RESOURCES: THEIR PERFORMANCE AND USAGE

IP *re-use* addresses the so-called design-productivity gap by using IPs in derivative and multiple designs. This approach works well for components, such as peripherals, memories, programmable processors, communication infrastructure, and real-time operating systems (RTOS). *Platforms* (Keutzer, Malik, Newton, Rabaey and Sangiovanni-Vincentelli, 2000), such as Philips's Nexperia (de Oliveira and van Antwerpen, 2003), provide the next level of reuse, by defining interfaces and protocols to connect the re-usable components (both hardware and software).

As an example, consider one of Philips's largest SoCs to date, PNX8550, shown in Figure 2-1 (Goossens, Gangwal, Röver and Niranjan, 2004), which exemplifies the Philips Nexperia platform. Many parts of its design are re-

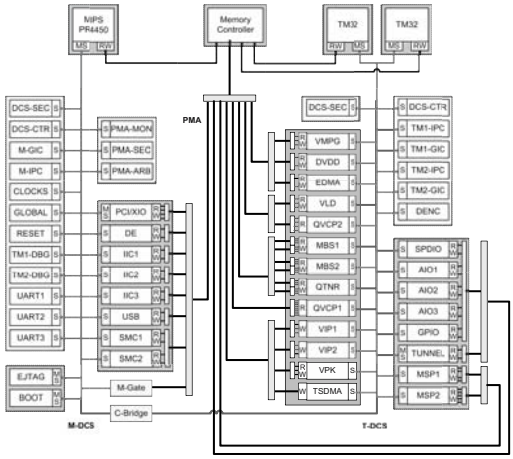


Figure 2-1. Simplified block diagram of PNX8550.

used from earlier designs, or are standard components that can be found in the library of Nexperia-compliant IP. They are combined using the device-transaction-level (DTL, 2002) hardware and TriMedia streaming software architecture (TSSA) software protocols, and use the PSOS operating system, as prescribed by the platform.

PNX8550 implements many set-top-box and video-enhancement functions, which require predictable real-time audio and video streaming. *Performance verification* is the difficult task of ensuring that the assembly of the large number of computation, storage, and communication IP meets all real-time constraints under all circumstances.

Although platforms have made assembling a SoC much easier, insufficient steps have been taken in applying the same ideas to performance verification of the resulting SoC. We tackle this issue by enriching the platform concept with services, to allow more explicit descriptions of, and reasoning about performance.

2.1 Why is Performance Verification Difficult?

Consider PNX8550, and assume that both TriMedia processors (TM32 in Figure 2-1) have a cache, and run multiple real-time tasks scheduled by a RTOS. If we want to know the time it takes a task on one of the TriMedia processors to communicate with a task on the other TriMedia using shared external memory,

there are several issues to consider. First, each task shares its TriMedia with other tasks. pSOS supports task preemption, and its arbitration mechanism therefore determines the delay before a tasks is active. Second, the caches may or may not contain the instructions and/or data of the task in question, resulting in a varying delay before the requested data is produced. Moreover, the cache is shared with other tasks on the same TriMedia. For example, an interrupt prior to swapping in the task could have flushed the cache, delaying the task's start. Third, the communication between the tasks uses the external memory. The memory is attached to the memory controller, which is again a shared resource with a sophisticated arbiter. Depending on the arbitration scheme employed, the write and read latencies and bandwidths may be influenced by other traffic contending for the external memory, such as the MIPS and the streaming traffic from the pipelined memory-access (PMA) interconnect (Goossens et al., 2004). As a result of these phenomena, computing the communication performance (latency and bandwidth) between the two tasks is non-trivial.

Resources and Users In the above example, we can identify several kinds of *resources*: *computation* (TriMedia, MIPS, IP such as the quality temporal noise reduction or QTNR), *communication* (device-control-and-status (DCS) and PMA interconnects), and *storage* (caches, off-chip and on-chip memories). These are used by several types of *users*: tasks (of computation), communication connections (of communication), and buffers for intermediate results or for communication (of storage).

In the example there are four independent factors complicating the performance analysis, as shown in Table 2-1: unpredictable resource usage, variable resource performance, users sharing resources, and (inter)dependence of multiple resources. We discuss each in turn below.

Table 2-1. Independent factors complicating performance analysis.

	user	resource
uncertainty	unpredictable usage	variable performance
multiple	shared resource	resource dependence

2.1.1 Unpredictable resource usage.

Algorithms defined by newer data compression standards, such as MPEG, are increasingly dynamic. For example, MPEG2's data compression allows variable bit rates, and MPEG4 uses dynamic object creation. As a result, the usage of resources (computation, communication, and storage) to encode or decode is variable. Figure 2-2 shows an abstract example of time-varying usage of a resource ("instantaneous usage").

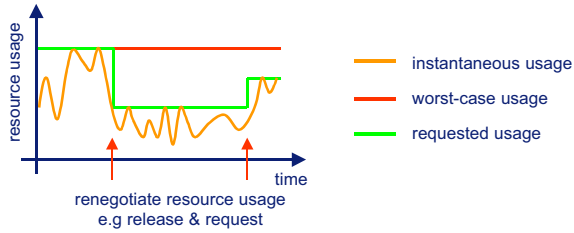


Figure 2-2. Variable resource usage and requested performance.

As an example, compressed MPEG2 data streams usually contain so-called I frames followed by several B and P frames. I frames are larger than B and P frames, and require more computation to decode them. However, I frames are required to decode B and P frames, and hence the memory requirements (size and bandwidth) to decode B and P frame are larger than those for I frames. However, a set-top box, for which PNX8550 is designed, must display a constant number of pictures per second on the TV screen, regardless of the content MPEG stream. Thus, even in a SoC with predictable resources, if their usage is variable, care has to be taken to ensure results with constant quality.

2.1.2 Variable resource performance.

As shown abstractly in Figure 2-3, resources themselves can have varying performance. The "instantaneous performance" of a resource, such as instructions per second, can vary over time for architectural reasons, as we describe below.

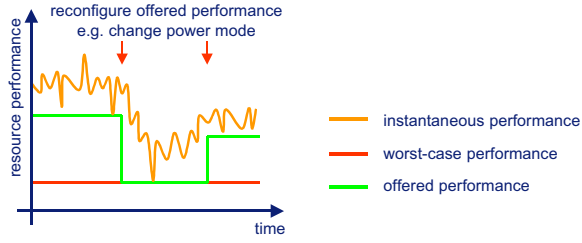


Figure 2-3. Variable resource performance and offered performance.

Computation resources, such as a MIPS, can have variable performance due to low-power (sleep) modes, which introduce a wake-up delay. In addition, techniques to reduce power consumption, such as voltage and/or frequency

scaling, give rise to multiple steady-state performance levels. As a result, the time it takes to perform a given fixed computation can vary.

Storage resources provide two examples of architectural variability: caches and volatile memories. A cache returns the requested data after a variable time. If the requested data is in the cache, it is returned quickly, but in the case of a cache miss, the data is returned after a much longer time. A cache therefore improves the *average* performance, but not the *guaranteed* (worst-case) performance. Although cache has a known deterministic algorithm, it is difficult to characterise its performance.

Volatile memories, such as dynamic random-access memories (DRAM), lose their data after some time, unless it is refreshed. The periodic refresh takes a long time, compared to a single memory access. This can result in unpredictable access times to the memory, depending on whether a read access is delayed by a refresh or not. Moreover, the order of read and write transactions, as well as the order in which transactions access the memory banks has a large impact on the memory's nett bandwidth. The *nett bandwidth* is the number of user data words per second as opposed to the number of cycles per second that the memory is occupied (*gross bandwidth*). Memory controllers therefore often reorder transactions to maximise the (average) nett memory bandwidth. As a result, the memory bandwidth and latency that a user experiences is dependent on his transactions (e.g. mix of reads and writes). Although the memory controller uses a known deterministic algorithm, the resulting average performance is difficult to characterise, like for caches, described above.

It is important to note that the variation in resource performance is *not intrinsic*, but is a consequence of the resource architecture. (Single-event upsets, such as alpha particles are an exception. They must be dealt with using error-correcting techniques.) The variable resource performance can be due to the resource's internal behaviour that, however, affects the user (e.g. processor sleep modes and SDRAM refresh), or it can be user dependent and difficult to capture (e.g. caches, or memory transaction reordering). In Section 3.2.2 we show examples of how architectures can be made predictable, and how their behaviour can be made to depend more clearly on the behaviour of users, using services.

2.1.3 Shared resources.

Consumer-electronics SoCs such as PNX8550 must deliver huge computational performance at low cost to the end-user. The number of computation, communication, and storage resources must therefore be minimised, and be used efficiently. Often this entails *sharing* a resource between multiple users.

For example, the programmable processors (TriMedia, MIPS) are shared between multiple tasks, and often include a real-time operating system (RTOS)

with some arbitration policy, which makes a task's execution time dependent on other tasks. Moreover, modern processors pipeline instructions, perform speculative execution, and so on. As a result, executing a given number of instructions may take different lengths of time, depending on e.g. the interleaving with other instruction streams. As a result, computing the number of (millions of) instructions per second (MIPS) as opposed to processor clock speed, may be difficult.

External memories are expensive because they raise the cost of the chip package, by introducing extra pins. In PNX8550, therefore, a single large external memory is used for the communication between streaming IPs. The scarce memory bandwidth is shared to near its capacity by the programmable processors and streaming IP. As we saw in the previous section, the memory's nett bandwidth is strongly impacted by the order of read and write transactions, as well as the order in which transactions access the memory banks. The memory controller therefore reorders transactions of the multiple users to maximise the nett memory bandwidth. As a result, the memory bandwidth and latency that a single user experiences is dependent on not only his, but also on other users's transactions.

Finally, shared "single-hop" communication infrastructures between multiple IP, such as busses and switches, contain a single arbiter, and face similar issues. For example, the DCS busses in PNX8550 use round-robin arbitration, and the PMA communication infrastructure uses a multi-level arbitration scheme (described in Goossens et al., 2004).

In all cases, many different arbitration policies are possible, when sharing a resource, e.g. first-come first-serve, round-robin, time-division-multiple-access (TDMA), and rate-monotonic scheduling. From the perspective of a single resource user, they introduce uncertainty regarding the resource performance. For example, the latency or bandwidth of a memory or bus may vary, depending on the behaviour of other users. This complicates the construction of a predictable SoC as a whole. Therefore, to design a predictable SoC it is helpful if shared resources use arbitration mechanisms amenable to analysis, e.g. TDMA (Rijpkema, Goossens, A. Rădulescu, van Meerbergen, Wielage and Waterlander, 2003), deadline-monotonic scheduling (Audsley, Burns, Richardson and Wellings, 1991). Next to this, to facilitate reasoning about performance, each user of a resource is preferably presented with a view on the resource that is independent of other users.

2.1.4 Dependence of multiple resources.

In the example at the start of this section, the two communicating tasks each used two shared resources: the programmable processor (computation) and the external memory (storage). (The communication infrastructure is not

shared for the programmable cores.) To reason about task-to-task communication performance, we must reason about all arbiters that are involved: it is the composition of the arbiters that determines the *end-to-end* performance. For example, a bandwidth guarantee of x bytes/sec on both processors and the external memory (ignoring caches), does *not* guarantee that the end-to-end (i.e. task-to-task) bandwidth is x. In the worst case, mismatched arbitration can cause starvation, resulting in zero bandwidth. The presence of "gates" or "bridges" in an architecture (see, e.g, Figure 2-1) couple arbiters of different resources, and are an indication that these issues could arise. (In fact, the gate and bridge in PNX8550 are well-behaved (Goossens et al., 2004).)

We use the term *(inter)dependence* for the effect that arbiters of different resources interact in an unforeseen or unintended manner, possibly degrading end-to-end performance. All shared resources that are used by a single user must be taken into account in an end-to-end performance analysis. Several approaches tackling this analysis are being investigated (Sha and Sathaye, 1993; Richter, Jersak and Ernst, 2003; Bekooij, Moreira, Poplavko, Mesman, Pastrnak and van Meerbergen, 2004), and they rely on expressing the user behaviour (e.g. worst-case execution time) and local arbitration policies in a single formalism for end-to-end reasoning.

As a special case, when the resources are of the same type, more specialised approaches exist. In particular, NoCs are "multi-hop" communication infrastructures, meaning that they are composed of multiple routers (or switches), each with their local arbiter. Fundamentally, this leads to the problem of interfering arbiters identified above. There is a great deal of research on providing end-to-end service guarantees in computer networks (Zhang, 1995; Rexford, 1999), and NoCs (Rijpkema et al., 2003; Goossens et al., 2003; Millberg, Nilsson, Thid and Jantsch, 2004; Liang, Swaminathan and Tessier, 2000) to which we return in Section 4.

For the construction of predictable SoCs, it must be possible to clearly describe and manage the interrelations and interdependencies between the behaviours of multiple resources. We believe that services, defined in the next section, provide a first step towards this goal.

2.2 Conclusions

We identified four reasons why performance verification is difficult: unpredictable resource usage, resources with variable performance, sharing of resources, and dependencies between multiple resources. These causes are independent and several of them usually act simultaneously, as we saw in the example of Section 2.1.

The first reason, unpredictable resource usage, is often externally imposed (external standards). Many algorithms, however, are, or can be made pre-

dictable, perhaps at some cost. We believe that service (levels), introduced below, can be used to characterising resource usage, in order to limit the effects of unpredictable resource requirements. The remaining reasons are due to architectural choices, which are under our own control. Service-based design, introduced in the next section, can help in making the right choices.

3. OFFERING AND USING SERVICES

In the previous section we identified four reasons why performance verification of SoCs is difficult, based on resources and resource users. In this section we describe and contrast two approaches to build SoCs: ad hoc and based on services. We motivate why we believe the latter has many advantages.

Ad-Hoc Systems The ad-hoc approach basically consists of instantiating a number of resources, adding arbiters to those that are shared. Performance verification is then difficult for the following reasons.

- To verify the performance of a SoC it must be considered in its entirety. It is not possible to consider the constituent resources in isolation because their behaviours are (inter)dependent and can interfere with one another, as we saw in Section 2.1.4.
- To accurately understand the complete SoC behaviour, current practice uses simulation of all (interdependent) resources in full detail. However, accurate simulation of the complete SoC is slow, which limits the number and length of simulations that can be performed. Moreover, simulation can only cover a small part of all possible SoC states and inputs (traces). It may be difficult to force a SoC to be in its worst state (e.g. longest latency) with simulation, especially if the worst state is unknown in advance. As a result, the observed worst case of the simulated traces can be much smaller than the real worst case. This could lead to underdimensioned resources (such as communication buffers), and a SoC that will not function correctly under all circumstances.
- If, during the performance verification process, a SoC is found to not meet its specification, a simulation trace does not necessarily give insight in how to remedy the problem. The most obvious cure, increasing the number of (shared) resources, may actually decrease the performance.
- It is not easy to make ad-hoc SoCs robust. Activation of a new user (e.g. a picture-in-picture in a set-top box) may cause a working SoC to fail completely, instead of affecting only the new user. We shall return to this issue below, in Section 3.2.1.

Below, we propose a compositional solution that is based the concept of services, to characterise and decouple the behaviour of both resources and users.

3.1 Services

We compose a SoC of resources such as processors, memories, and NoCs. These resources offer *services*,[1] which are requested and used by users. A user service request includes *attributes* to specify the desired service *level*. Examples of computation, communication, or storage service attributes are (Rădulescu and Goossens, 2004):

- Uncorrupted completion, e.g. of a write transaction to a memory, or its transport by the communication resource. If an action completes, then it is guaranteed to be correct.
- Guaranteed completion. This is not automatic; e.g. a task may be blocked until a minimum amount of memory is available, and in a NoC data may be dropped in case of congestion.
- (Minimum) capacity, e.g. amount of buffering, the number of simultaneous users of a resource.
- Ordering: is there any ordering between subsequent actions? Examples are read transactions from one master IP to multiple slave IPs, which in a NoC can come back out of order, and also multiple computations which can finish out of order on a processor.
- (Minimum) average throughput, measured in instructions per second for computation resources, and bytes per second for memory and communication resources.
- (Maximum) bound on the completion time. Guaranteed completion is defined as an unspecified completion time less than infinity; here the maximum is finite and known in advance. Examples are the latency of a read transaction on the memory, or its transport by the communication resource.
- (Maximum) variation in completion time (jitter), which is important for real-time audio and video.

These service attributes can be combined to specify a particular service level, e.g. a communication connection between two IPs could be lossless, ordered, with 100 Mbyte/sec average throughput, and with a maximum latency of 0.8 microseconds.

A resource can offer different services levels (or differentiated services, Kumar, Lashman and Stiliadis, 1998) to different users at the same time. For example, a NoC may offer communication services with different latency, throughput and jitter levels, e.g. for control traffic (low latency, low throughput) and

[1] Or: resources are used to offer services. In this chapter a narrow view on services is taken, by restricting them to a single kind of resource (computation, storage, or communication). It is possible and useful to generalise services to use multiple resources, as well as lower-level services. Examples are database, printing, or secure-storage services.

streaming traffic (high throughput, low jitter). It is fruitful to offer different services levels simultaneously to increase the resource utilisation, as argued in (Goossens et al., 2003; Rijpkema et al., 2003).

Most services must be *negotiated* (Figure 2-4): a user must specify and request his desired service level from the resource. A service level describes both

Figure 2-4. Users (request service level) negotiate with resources (offer service level).

the performance offered by a resource to a user (e.g. "the NoC has only lossless, ordered connections available with at most 10 Mbyte/sec average throughput"), as well as the (potentially different) performance requested by a user ("the current task graph requires three connections with 5 Mbyte/sec average throughput but without loss or ordering constraints"). If the resource commits to the request, then the service is then guaranteed to be available until the user releases the service, when he no longer needs it. Otherwise the resource rejects the request, and the user must give up or retry with different (lower) service requirements. Note that a service is either committed to (i.e. guaranteed) or not. A resource cannot renegade on its commitment.

Services must be negotiated because the resource must ensure that its capacity (storage size, instructions per second, bytes per second, etc.) is not oversubscribed, to avoid invalidating the services it has already committed to. Resources manage their number of users by performing admission control, which is why users must specify their required services in advance. Moreover, after admission, users must be prevented from using more than their allocated share of the resource (Otero Pérez, Rutten, van Eijndhoven, Steffens and Stravers, 2005).

The service concept is well established: it originated in protocol communication stacks, e.g OSI (Rose, 1990) and has been extended to cover resource discovery, leases, etc. in approaches such as Sun's Java Jini (Jin, 2001), and HAVi (HAV, 2000; Lea, Gibbs, Dara-Abrams and Eytchison, 2000). A *lease* is a service that is valid for a certain amount of time (we have assumed it will remain valid until the user releases it). This is more robust, in case the resource user does not correctly release resources (e.g. in the case of unreliable com-

munication between user and resource, or malicious or fault users), or when resources are inherently unreliable. Although currently not required for SoCs, we anticipate that these techniques will be applicable in the long term.

3.2 The Advantages of Using Services

We will now describe how services are used to ease each of the four obstacles to building predictable SoCs, identified in Section 2.1: unpredictable resource usage, variable resource behaviour, shared resources, usage of multiple dependent resources (Table 2-1). Table 2-2 outlines how services address each case; a fuller description is given below.

Table 2-2. Services simplify performance analysis.

	user	resource
uncertainty	*characterise* unpredictable usage	*abstract* variable performance
multiple	*virtualise* shared resources	*decouple* resource behaviours

3.2.1 Services characterise unpredictable resource usage.

Quality of service is the process whereby a trade off is made between the available resources and the requests to implement the functionality (quality) required by the user (Figure 2-4). For example, suppose that a set-top box displays a high-definition film, when the user requests a picture in picture (PIP) (Otero Pérez, Steffens, van der Stok, van Loo, Alonso, Ruíz, Bril and Valls, 2003). With the resources available in the SoC it may not be possible to honour this request. The first possible course of action ("ad-hoc," common in personal computers) is to activate the PIP anyway. This will result in a mode where neither the high-definition film nor the PIP are displayed correctly, and the result can be anything from a "blue screen" (crashed system) to a garbled screen. Alternatively, in a service-based SoC, the quality-of-service manager requests the additional services required by the PIP (e.g. additional memory bandwidth) from the appropriate resources. If not all resources commit to the requested services, then the high-definition film and PIP can not be activated simultaneously. The SoC could inform the user that this is the case. (Note that the high-definition film has been running undisturbed throughout this process.) Another option would be to change the high-definition film to a standard-definition film (requiring fewer resources), freeing enough resources to also support the PIP.

Here we are concerned not so much with quality of service, but rather how to enable it. Services form the basis, by abstracting variable resource usage to requested services for users (see Figure 2-2), and by abstracting variable

resource performance to offered services (see Figure 2-3) for resources. We first discuss discuss the former, the next section describes the latter.

Different service levels abstract instantaneous resource requirements of the user. This allows less frequent negotiation ("negotiated usage" versus "instantaneous usage" in Figure 2-2), at the cost of claiming too many resources. The limits of this trade off are continuous negotiation (returning to "instantaneous usage") and worst-case design with no negotiation ("worst-case usage"). Two renegotiations are shown in Figure 2-2: the first reduces the negotiated usage, and the second increases it.

Services simplify the interface and corresponding interaction between user and resource because the requirements of the user are requested using abstract service levels, instead of detailed descriptions of actual instantaneous usage. This simplifies the implementation of resources.

Moreover, as the PIP example demonstrates, SoCs are more robust when using services because it is possible to verify in advance that a mode change will succeed, without disturbing active functions.

3.2.2 Services abstract variable resource performance.

Different service levels offered by a resource can also abstract its variable resource performance. As an example, in Figure 2-3 the instantaneous actual performance of a resource (e.g. voltage-controlled processor) may be variable and difficult to capture exactly. The offered performance therefore offers simplified view on the resource (e.g. piece-wise constant). Two reconfiguration points are shown, which could correspond to an adaptation of voltage to change processor speed.

Services offer an abstract view on resource performance, to make it simpler for users to claim the performance they desire. For example, a NoC user could ask for a connection with 100Mbyte/sec average throughput and a maximum latency of 2 microseconds. The NoC translates this abstract request for nett bandwidth (user data per second) to its internal representation of gross bandwidth (which takes into account, e.g. packetisation and flow-control overhead, and the number and spacing of TDMA slots). The underlying NoC arbitration policy (TDMA or otherwise) and architecture (flow control or not), etc. that implement the services are hidden from the user because they are irrelevant to him. The translation from gross resource performance to what is offered nett to the user may not be easy, as we have seen in Section 2.1.2. In Section 4 and elsewhere in this volume (Gangwal, Rădulescu, Goossens, González Pestana and Rijpkema, 2005) we describe in more detail the ÆTHEREAL NoC where this translation has been implemented successfully.

Abstract services also make QoS independent of particular resource implementations. QoS managers match the requested user services with the offered

resource services. After resources have committed to providing services they must not renegade on its commitment, because this makes the notion of negotiation superfluous, and makes it hard for the QoS manager to offer a reliable service to end users. Taking processor power management as an example, Simunic, Boyd and Glynn, 2004 describes how resources can autonomously change their performance (e.g. frequency) to optimise a power budget. This impacts the service levels users receive. Instead resources should regulate their performance in concordance with its users. A good example is the autonomous islands of performance of Meijer, Pessolano and Pineda de Gyvez, 2004, where a resource's performance (operating frequency) is specified by the user, and the resource internally finds an optimal operating point (using adaptive voltage scaling and adaptive body bias) that guarantees the requested performance, even under (varying) environmental conditions (such as silicon processing variations, and voltage drops). Predictable system-level power and performance management can be built on top of these islands of performance (Hu and Marculescu, 2004).

3.2.3 Services virtualise shared resources.

In Section 2.1.3 we discussed how sharing a (constant-performance) resource can result in a variable performance for a single user. However, when, in a service-based SoC, a resource commits a particular service level to a user, it guarantees that the service is available to the user independent of other users of the resource. Thus, every user has his own *virtual* resource, with a performance that has been agreed upon during negotiation.

As a result, the users can be simpler because they have fewer failure modes. A user can be affected by the other users only during negotiation for a service (when the resource rejects the request), instead of any point in time (as happened in the ad-hoc implementation of the PIP example of Section 3.2.1). Services can thus isolate users from one another. This avoids the need for cooperation between users (such as required by e.g. the internet's transmission control protocol), and can make the SoC more robust against erroneous or misbehaving users (Kumar et al., 1998).

3.2.4 Services decouple usage of multiple resources.

In Section 2.1.4 we observed that when a user uses multiple shared resources, unforeseen interactions (dependencies) between these resources can affect the end-to-end performance the user obtains. The previous section showed that services decouple (or isolate) users of a single shared resource, and that each user can reason about his services independent of other users. As a result,

when a user uses multiple shared resources, he can reason about all resource reservations independently (they are decoupled).

However, as discussed in Section 2.1.4, and as is shown elsewhere (Bekooij et al., 2004), resource requirements are interdependent when end-to-end performance guarantee must be given that involve multiple resources. For example, when two tasks on different processors communicate via shared memory, the processor bandwidth, memory bandwidth, and memory buffer size are interdependent.

Although services do not remove this interdependence, there are several advantages when they are used. First, resource performance is reasoned about in terms of abstract nett service levels rather than the actual detailed resource implementation. Second, users of shared resources can be considered independently because they each have their own virtual resource. Both cases reduce the complexity of the QoS manager, which can use data-flow (Bekooij et al., 2004), and other (Richter et al., 2003) techniques to compute the resource reservations that ensure end-to-end (e.g. task-to-task) performance guarantees.

3.3 Conclusions

In this section we introduced the notions of services and service levels. A requested service level serves to abstract or simplify the description variable resource requirements of a user (Figure 2-2) by hiding internal details and dynamism. Similarly, an offered service level serves to abstract or simplify the description of the variable offered performance of a resource (Figure 2-3). Figure 2-4 then shows how a QoS manager matches the requested and offered services, using negotiation. Abstract, implementation-independent services are an important enabler for effective QoS.

Services *decouple* the multiple users of a single resource (Section 3.2.3), as well as the multiple resources used by a single user (Section 3.2.4). As a result, resource users can be simpler, and SoCs can be made more robust. Although resource interdependencies are not eliminated by services, they become explicit and more abstract.

Service-based design can reduce functional and performance verification of the complete SoC in several ways. First, users and resources are specified in terms of their services. For example, a communication or storage resource can be specified to support a certain number of users with particular service levels (e.g. with nett bandwidths). Following this, they can be independently designed, implemented, and their function and performance verified, because their specifications and implementations do not depend on other users or resources. Users implement their functionality making use of (building on top of) services provided by the resources. It is easier to reason about abstract services provided by resources than about their combined implementations. After

integrating the verified user and resource implementations, the SoC as a whole must be verified. Because resources are known to be correct, system verification can take place at the level of services offered by the resources, and not performed on the ensemble of all user and resource implementations (the ad-hoc approach). This compositional method is also known as *assume-guarantee reasoning* (Henzinger, Qadeer and Rajamani, 2000), because by guaranteeing the performance or behaviour of components (service providers), this guarantee can be used as a safe assumption in the performance analysis in the larger SoC using it (service users). Services naturally provide the abstraction for the guarantee step.

In the next section we will show how the ÆTHEREAL NoC can be automatically generated, programmed, and verified because it has been designed with these concepts in mind.

4. CASE STUDY: THE ÆTHEREAL NETWORK ON CHIP

The communication infrastructure is key in any platform (Sgroi, Sheets, Mihal, Keutzer, Malik, Rabaey and Sangiovanni-Vincentelli, 2001) because it integrates all IP into a larger SoC, and because it is the locus of the platform communication protocols. The communication infrastructure is therefore a natural place to initiate a service-based design method. In this section we discuss how Philips's ÆTHEREAL NoC (Goossens et al., 2003) attempts to solve the issues raised in Section 2 by introducing communication services, as described in Section 3.

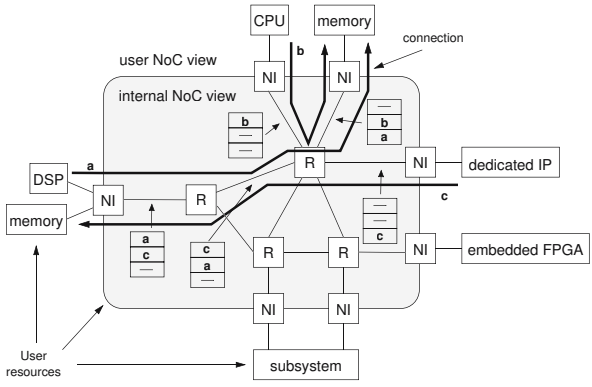


Figure 2-5. SoC composed of heterogeneous IP interconnected by a NoC.

Figure 2-5, shows the basic architecture of a NoC. There are two different points of view: (a) that of the NoC user, where the whole NoC can be seen as

a single resource providing communication services to different users, and (b) the internal NoC view consisting of multiple interacting resources.

A NoC is composed of two components: routers (R) and network interfaces (NI), see Figure 2-5. Routers transport data within the NoC. NIs convert the IP view on communication (e.g. read and write transactions) to the NoC's internal view (e.g. packets, flow control). Importantly, the NIs also implement the service abstraction, reducing the NoC's internal multiple-resource view to a NoC user's single-resource view.

In the remainder of this section we describe ÆTHEREAL's service-based communication model, which comprises *best-effort* (BE) and *guaranteed-throughput* (GT) service levels. We explain their characteristics and intended uses, and how they aim to enable service-based SoC design.

4.1 The Æthereal Communication Model

As discussed above, in the NoC internal view, ÆTHEREAL is a multi-hop interconnect, i.e. it contains multiple components (routers and NIs). Each of these components has a constant performance (e.g. every NoC link has 2Gbyte/sec bandwidth, Rijpkema et al., 2003). The NoC is shared by multiple users, who may have variable resource requirements.

ÆTHEREAL offers communication services, and comprises the *best-effort* (BE, Section 4.1.1) and *guaranteed-throughput* (GT, Section 4.1.2) service levels. These service levels have different characteristics and intended uses. The BE service level exhibits several of the problems listed in Table 2-1, whereas the GT service level does not. However, as argued in Goossens et al., 2003, it is advantageous to offer both service levels to increase resource utilisation and hence reduce cost.

Communication services are provided on connections (Rădulescu and Goossens, 2004). A connection specifies the communication between one master (e.g. the digital-signal processor DSP of Figure 2-5) and one or more slaves (e.g. distributed shared memories). Figure 2-5 shows three example connections. The user indicates the required service level per connection by specifying communication attributes, as described in Section 3.1. A BE connection offers uncorrupted, lossless, ordered communication, to which GT connections add minimum throughput, maximum latency, and maximum jitter.

As discussed in Section 2.1, the translation from the user view on performance to the NoC view on performance may be far from trivial. We illustrate this for NoCs in Figure 2-6. A user of a NoC most often reasons in terms of application data, such as bits per second of an MPEG stream ("nett bandwidth"). Assuming this data is memory-mapped, the IP uses read and write transactions to access the data, and a command and address are added to the application data. The NI convert these transactions into packets, by chopping it into pieces

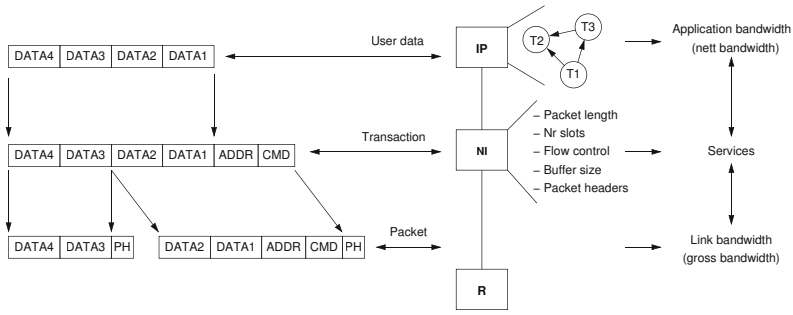


Figure 2-6. From user view to NoC view on communication.

and adding a header. Packets may be of different lengths, and the NoC may also internally generate packets that are not visible to the user, e.g. for flow control. As a result, the gross bandwidth to be claimed inside the NoC will be more than the requested bandwidth for the application data. The strength of ÆTHEREAL NoC is that the communication services include this nett to gross bandwidth translation (described in detail in Gangwal et al., 2005).

The following two subsections describe the BE and GT service levels, and how they enable the move from the internal NoC view on communication to the user's view on communication, by solving the problems described in Table 2-1 (resource sharing, interdependent resources).

4.1.1 The best-effort service level.

We first describe what the BE service level consists of, and how it is implemented. Then we list which of the problems of Table 2-1 are present, and finally motivate the reasons for offering BE service level.

BE connections implement uncorrupted, lossless, ordered data transport (transaction completion is a result of absence of data loss). This is implemented by a common NoC architecture (Rijpkema et al., 2003): a packet-switched NoC, with input-queued routers using worm-hole routing and round-robin arbitration. Packets are never dropped, and credit-based end-to-end flow control is used to avoid congestion. Packet ordering is ensured by deterministic source routing.

A NoC will be shared by multiple users, and their packets may clash inside the NoC. To solve this contention, routers and NIs use local round-robin arbitration. However, when a connection uses multiple routers or NIs, the combined effect of multiple interdependent arbiters becomes difficult to characterise. Input queuing causes interdependencies between different connections (called head-of-line blocking), and worm-hole routing causes interdependencies between arbiters of different routers. As listed in Table 2-3, these are examples

Table 2-3. The best-effort service level.

problem (cf. Table 2-1)	BE service level
unpredictable resource usage	not addressed
variable resource performance	not applicable
resource shared by multiple users	local round-robin arbitration
multiple interdependent resources	local round-robin arbitration

of resource sharing and interdependent resources. Note that unpredictable resource usage is not addressed by the BE service level, and that each of the routers and NIs has a constant performance.

As a result, end-to-end (IP-to-IP) service guarantees such as throughput, latency, and jitter can not be given. Thus, only simulation can be used to correctly dimension a NoC (including its topology, buffer sizes, etc.) for given application requirements (throughput, latency, etc.).

Nonetheless, ÆTHEREAL offers the BE service level for a number of reasons. First, it enables a NoC to be used where user resource requirements can not be characterised well, or are highly variable. Moreover, not all applications require real-time guarantees, such as web browsing or graphics. By using the BE service level the NoC resources can be dimensioned for the average instead of worst-case communication requirements. This allows a higher resource utilisation, potentially using fewer resources, i.e. a smaller NoC. The BE service level therefore trades real-time performance for higher resource utilisation.

4.1.2 The guaranteed-throughput service level.

We first describe what the GT service level consists of, and how it is implemented. Then we list how the problems of Table 2-1 are addressed.

The GT service level adds minimum throughput, and maximum latency and jitter bounds to the BE service level. This is implemented by a NoC architecture first introduced by ÆTHEREAL (Rijpkema et al., 2003): a global distributed TDMA arbitration scheme that emulates pipelined time-division-multiplexed circuit-switched connections. ÆTHEREAL implements the *global* TDMA arbitration in a *distributed* manner (using only local synchronisation). This scheme eliminates contention, and hence ensures minimal buffering in routers (one-flit input queues for worm-hole routing). Figure 2-5 shows an example NoC with three GT connections, labelled a, b, and c, with the corresponding TDMA tables and slot reservations.

A NoC will be shared by multiple users, as is the case for the BE service level. However, the GT service level avoids contention in the NoC by means of global TDMA arbitration. The same scheme also eliminates resource interdependencies due to head-of-line blocking and worm-hole routing. As a result,

Table 2-4. The guaranteed-throughput service level.

problem (cf. Table 2-1)	GT service level
unpredictable resource usage	must be characterised
variable resource performance	not applicable
resource shared by multiple users	analysable global TDMA arbitration
multiple interdependent resources	analysable global TDMA arbitration

throughput, latency, and jitter guarantees can be derived as described elsewhere in this volume (Gangwal et al., 2005). As listed in Table 2-4, this solves resource sharing and interdependent resources that plagued the BE service level. When a GT connection is requested, the required throughput, latency, and jitter must be specified, to reserve communication resources (essentially, buffers and slots in the TDMA tables), in contrast to a BE connection. Thus, the resource usage must be characterised by the user, as discussed in Section 3.2.1. Finally, note that each of the routers and NIs has a constant performance.

```
# User view: GT service-level request:
open_connection("decoder.mc","mem.p2","GT",
                72 Mbyte/sec,2.5 ms,16 byte,72 Mbyte/sec,1.7 ms,16 byte)
                # throughput, latency, burst size for read & write
```

```
# Internal NoC view: GT reservation:
open_connection("decoder.mc","mem.p2",
                "GT","22-32","3 1 0",33, "GT","7-13","1",60)
                # type, slots, path, credits for request & response
```

Figure 2-7. Service level versus GT connection reservation.

As a result, end-to-end (IP-to-IP) service guarantees such as throughput, latency, and jitter can be given on GT connections. The GT service level first enables the transition from an internal NoC view to a service-based user view (Figure 2-6). That is, the internal structure of the NoC is hidden for the user, and the collection of resources (routers and NIs) behaves as a single resource. A single global arbitration scheme (TDMA) implements resource sharing, and resource interdependencies are eliminated. Second, building on top of this, the user's (nett) requirements are translated to internal NoC (gross) resource reservations by the NoC, as advocated in earlier sections. A simplified example for a single connection is shown in Figure 2-7. A user specifies a connection from a master to a slave with required nett bandwidth and latency constraints, for given burst sizes, as shown in the top half of the figure. This is translated to the internal resource reservation view, consisting of slots, path, credits, etc., shown in the lower half of the figure.

Disadvantages of the GT service level include the need to characterise user communication requirements in advance. Between negotiation points (Fig-

ure 2-2), resources are reserved for the worst-case, potentially increasing the NoC size. The GT service level therefore trades real-time performance for possibly higher resource requirements.

4.1.3 Combining BE and GT services levels.

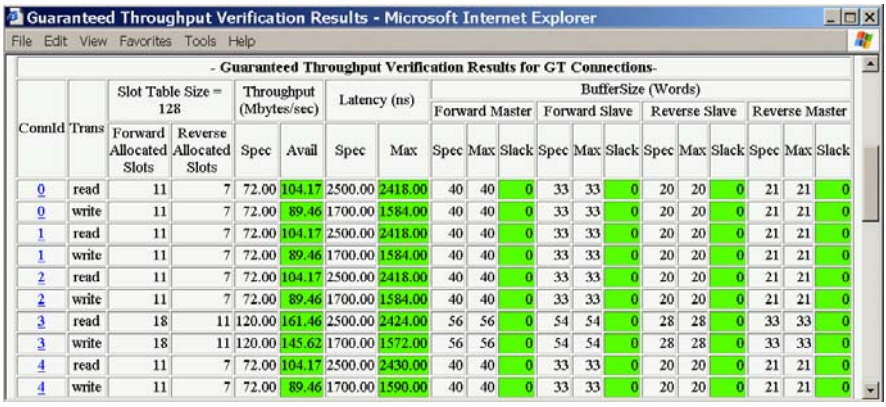
In the two preceding sections we have introduced the BE and GT service levels. The former aims for high resource utilisation for which it sacrifices throughput and latency guarantees. The latter aims for real-time performance guarantees, potentially at the cost of more resources (a larger NoC). By offering both service levels, ÆTHEREAL resources are reserved as required for GT connections, but unclaimed or unused GT bandwidth is used by BE connections. As a result, real-time (GT) services and good resource utilisation (low cost) are combined (Rijpkema et al., 2003; Goossens et al., 2003).

4.2 The Æthereal Design Flow

The ÆTHEREAL NoC design flow (Goossens, Dielissen, Gangwal, González Pestana, Rădulescu and Rijpkema, 2005) comprises design-time NoC generation (i.e. dimension and generate the NoC hardware based on user requirements), NoC configuration (compute the resource reservations from the user requirements, as shown in Figure 2-7), NoC simulation, and NoC performance verification (for GT connections). User requirements are usually stated as a collection of modes (or use cases) that the SoC must support, and NoC configuration therefore usually proceeds at the granularity of modes rather than connections. Figure 2-8 shows an example of performance verification. Connection 2 corresponds to the decoder.mc to mem.p2 connection of Figure 2-7. For each connection the computed resource reservations (number of TDMA slots), specified and available (minimum) bandwidth and (maximum) latency are shown, as well as the specified and required buffer sizes in the NoC. NoC generation, configuration, and verification for GT connections is performed on the basis of analytical models. Hence, simulation is only required if BE connections are used.

5. CONCLUSIONS

In this chapter we described and analysed the problem of performance verification of SoCs. We identified four reasons why building SoCs with predictable performance is difficult (Table 2-1): unpredictable resource usage, variable resource performance, resource sharing, and interdependent resources. We introduced the concept of a service, aiming to address these problems, and described its advantages over "ad-hoc" approaches. Finally, we introduced the ÆTHEREAL NoC as a concrete example of a communication resource that implements multiple service levels.



Guaranteed Throughput Verification Results - Microsoft Internet Explorer

File Edit View Favorites Tools Help

- Guaranteed Throughput Verification Results for GT Connections-

ConnId	Trans	Slot Table Size = 128		Throughput (Mbytes/sec)		Latency (ns)		BufferSize (Words)											
								Forward Master			Forward Slave			Reverse Slave			Reverse Master		
		Forward Allocated Slots	Reverse Allocated Slots	Spec	Avail	Spec	Max	Spec	Max	Slack	Spec	Max	Slack	Spec	Max	Slack	Spec	Max	Slack
0	read	11	7	72.00	104.17	2500.00	2418.00	40	40	0	33	33	0	20	20	0	21	21	0
0	write	11	7	72.00	89.46	1700.00	1584.00	40	40	0	33	33	0	20	20	0	21	21	0
1	read	11	7	72.00	104.17	2500.00	2418.00	40	40	0	33	33	0	20	20	0	21	21	0
1	write	11	7	72.00	89.46	1700.00	1584.00	40	40	0	33	33	0	20	20	0	21	21	0
2	read	11	7	72.00	104.17	2500.00	2418.00	40	40	0	33	33	0	20	20	0	21	21	0
2	write	11	7	72.00	89.46	1700.00	1584.00	40	40	0	33	33	0	20	20	0	21	21	0
3	read	18	11	120.00	161.46	2500.00	2424.00	56	56	0	54	54	0	28	28	0	33	33	0
3	write	18	11	120.00	145.62	1700.00	1572.00	56	56	0	54	54	0	28	28	0	33	33	0
4	read	11	7	72.00	104.17	2500.00	2430.00	40	40	0	33	33	0	20	20	0	21	21	0
4	write	11	7	72.00	89.46	1700.00	1590.00	40	40	0	33	33	0	20	20	0	21	21	0

Figure 2-8. Performance verification output example.

ACKNOWLEDGEMENTS

The authors thank Liesbeth Steffens and Clara Otero Pérez for their extensive and constructive feedback.

References

Audsley, N. C., Burns, A., Richardson, M. F. and Wellings, A. J., 1991, Hard real-time scheduling: The deadline monotonic approach, *in* W. A. Halang and K. Ramamritham (eds), *Real-Time Programming*, pp. 127–132.

Bekooij, M., Moreira, O., Poplavko, P., Mesman, B., Pastrnak, M. and van Meerbergen, J., 2004, Predictable embedded multiprocessor system design, *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, LNCS 3199, Springer.

de Oliveira, J. A. and van Antwerpen, H., 2003, The Philips Nexperia digital video platform, *in* G. Martin and H. Chang (eds), *Winning the SoC Revolution*, Kluwer Academic.

DTL, 2002, *Device Transaction Level (DTL) Protocol Specification. Version 2.2*.

Gangwal, O. P., Rădulescu, A., Goossens, K., González Pestana, S. and Rijpkema, E., 2005, Building predictable systems on chip: An analysis of guaranteed communication in the æthereal network on chip, In this volume.

Goossens, K., Dielissen, J., Gangwal, O. P., González Pestana, S., Rădulescu, A. and Rijpkema, E., 2005, A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design

and verification, *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*.

Goossens, K., Dielissen, J., van Meerbergen, J., Poplavko, P., Rădulescu, A., Rijpkema, E., Waterlander, E. and Wielage, P., 2003, Guaranteeing the quality of services in networks on chip, *in* A. Jantsch and H. Tenhunen (eds), *Networks on Chip*, Kluwer, chapter 4, pp. 61–82.

Goossens, K., Gangwal, O. P., Röver, J. and Niranjan, A. P., 2004, Interconnect and memory organization in SOCs for advanced set-top boxes and TV — Evolution, analysis, and trends, *in* J. Nurmi, H. Tenhunen, J. Isoaho and A. Jantsch (eds), *Interconnect-Centric Design for Advanced SoC and NoC*, Kluwer, chapter 15, pp. 399–423.

HAV, 2000, *The HAVi Specification. Version 1.0*.

Henzinger, T. A., Qadeer, S. and Rajamani, S. K., 2000, Decomposing refinement proofs using assume-guarantee reasoning, *Proc. of Int'l Conference on Computer Aided Design (ICCAD)*, pp. 245–252.

Hu, J. and Marculescu, R., 2004, Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints, *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*.

Jin, 2001, *Jini Architecture Specification, Version 1.2*.

Keutzer, K., Malik, S., Newton, A. R., Rabaey, J. M. and Sangiovanni-Vincentelli, A., 2000, System-level design: Orthogonalization of concerns and platform-based design, *IEEE Trans. on CAD of Integrated Circuits and Systems* **19**(12), 1523–1543.

Kumar, V. P., Lashman, T. V. and Stiliadis, D., 1998, Beyond best effort: Router architectures for the differentiated services of tomorrow's internet, *IEEE Communications Magazine* pp. 152–164.

Lea, R., Gibbs, S., Dara-Abrams, A. and Eytchison, E., 2000, Networking home entertainment devices with HAVi, *IEEE Computer* **33**(9), 35–43.

Liang, J., Swaminathan, S. and Tessier, R., 2000, aSOC: A scalable, single-chip communications architecture, *Proc. Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*.

Meijer, M., Pessolano, F. and Pineda de Gyvez, J., 2004, Technology exploration for adaptive power and frequency scaling in 90nm CMOS, *Proc. Int'l Symposium on Low Power Electronics and Design (ISPLED)*, pp. 14–19.

Millberg, M., Nilsson, E., Thid, R. and Jantsch, A., 2004, Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip, *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*.

Otero Pérez, C. M., Steffens, L., van der Stok, P., van Loo, S., Alonso, A., Ruíz, J. F., Bril, R. J. and Valls, G., 2003, QoS-based resource management for ambient intelligence, *in* T. Basten, M. Geilen and H. de Groot (eds), *Ambient Intelligence: Impact on Embedded System Design*, Kluwer, pp. 159–182.

Otero Pérez, C., Rutten, M., van Eijndhoven, J., Steffens, L. and Stravers, P., 2005, Resource reservations in shared-memory multiprocessor SOCs, In this volume.

Rexford, J., 1999, *Tailoring Router Architectures to Performance Requirements in Cut-Through Networks*, PhD thesis, University of Michigan, department of Computer Science and Engineering.

Richter, K., Jersak, M. and Ernst, R., 2003, A formal approach to MpSoC performance verification, *IEEE Computer* **36**(4), 60–67.

Rijpkema, E., Goossens, K., A. Rădulescu, J. D., van Meerbergen, J., Wielage, P. and Waterlander, E., 2003, Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip, *IEE Proceedings: Computers and Digital Technique* **150**(5), 294–302.

Rose, M. T., 1990, *The Open Book: A Practical Perspective on OSI*, Prentice Hall.

Rădulescu, A. and Goossens, K., 2004, Communication services for networks on chip, *in* S. S. Bhattacharyya, E. F. Deprettere and J. Teich (eds), *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, Marcel Dekker, pp. 193–213.

Sgroi, M., Sheets, M., Mihal, A., Keutzer, K., Malik, S., Rabaey, J. and Sangiovanni-Vincentelli, A., 2001, Addressing the system-on-a-chip interconnect woes through communication-based design, *Proc. Design Automation Conference (DAC)*, pp. 667–672.

Sha, L. and Sathaye, S. S., 1993, A systematic approach to designing distributed real-time systems, *Computer* **26**(9), 68–78.

Simunic, T., Boyd, S. P. and Glynn, P., 2004, Managing power consumption in networks on chips, *IEEE Transactions on VLSI Systems* **12**(1), 96–107.

Zhang, H., 1995, Service disciplines for guaranteed performance service in packet-switching networks, *Proceedings of the IEEE* **83**(10), 1374–96.

Chapter 3

CACHE-COHERENT HETEROGENEOUS MULTIPROCESSING AS BASIS FOR STREAMING APPLICATIONS

Jos van Eijndhoven, Jan Hoogerbrugge, Jayram M.N., Paul Stravers, and Andrei Terechko
Philips Research Laboratories, Eindhoven, The Netherlands

Abstract: Systems-on-Chip (SoC) of the new generation will be extremely complex devices, composed from complex subsystems, relying on abstraction from implementation details. These chips will support the execution of a mix of concurrent applications that are not known in detail at chip design time. These SoCs require a significant degree of programmability to configure both the set of functions that must execute as well as the structure of the dataflow between these functions. To ease the programming effort multiprocessor computers have employed cache coherent share memory for decades, abstracting the average programmer from system complexity issues such as multiple processors and memory hierarchies. Memory coherency in multiprocessor computers has a history of decades, and has proven to be an indispensable abstraction from system complexity towards the application programmer. This chapter describes a next generation SoC for the consumer electronics domain (e.g. audio/video, vision, robotics). It features heterogeneous multiprocessor subsystems with a snooping cache coherence protocol, combined in a system with distributed memory employing a directory coherency protocol. It is explained why and how the coherent memory model is indispensable for implementing both data transport and synchronization for multi-tasking streaming applications in distributed memory systems.

Key words: System-on-Chip, multi-processor, cache coherency, streaming, memory hierarchy

P. van der Stok (ed.),
Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, 61-80.

1. INTRODUCTION

The semiconductor industry is facing enormous challenges with the creation and marketing of every new generation of System-on-Chip (SoC) in the consumer electronics market segment. These consumers want trendy products loaded with modern features, matched to their personal taste. However the creation of such systems today with hundreds of millions of transistors in hardware and tens of megabytes of embedded software is a task of daunting complexity. The design process goes through stages of specification, implementation and verification both for the hardware itself as for the embedded software, with contributions of multiple design groups spread over the world and over multiple companies. These processes take several tens to a few hundred man-years of effort, stretched over a few years of elapsed time for major new products. This trend is shown for instance in a 2003 analysis of IBS (Int. Business Strategies inc.), see Figure 3-1.

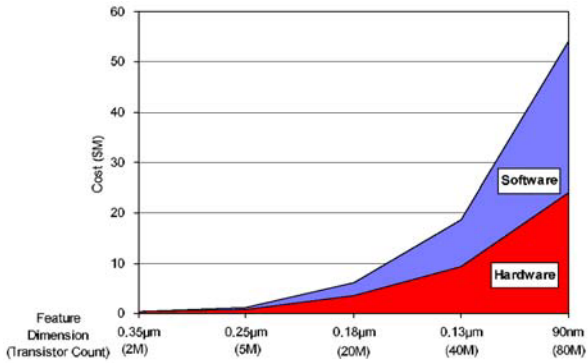


Figure 3-1. Soc development cost

In view of the fast changing and cost sensitive consumer market, such SoC products require a huge investment and carry high risk. This chapter describes the SoC architecture being created in the Philips Research 'Computer Architecture for Killing Experience (CAKE)' project (Stravers, 2001). The project targets the consumer electronics media processing domain, aiming to address the hardware and software SoC challenges mentioned above. The first implementation of this architecture is now in the design phase. The following paragraphs describe a set of top-level design considerations that together form the key aspects of the CAKE architecture.

Programmability: Programmability is a key property that allows late changes in the product functionality, as well as adaptations to local market needs. Similarly, programmability can be regarded as an insurance cost to reduce the investment risk of creating a silicon product with features that mismatch with new demands at the time the product reaches the market. Programmability also allows the re-use of the same silicon across different products, maybe created by different companies. This is an important aspect, as for domain-specific SoCs the silicon design cost will not be negligible in comparison with the production cost. If the chip supports an industry-standard programming model, functions can be created by re-use of standard software, which will give a tremendous reduction on system development cost and time. The main factor that opposes programmability is power consumption, leading to more specialized engines (or coprocessors) for some functions.

Parallel processing: The employment of multiple small processors to match the large computation workload as represented by today's streaming media processing, will be beneficial both for the silicon area as well as for the power consumption. The micro-architecture of current high-end microprocessors shows a clear problem of diminishing returns (Hennesy and Patterson, 2003; Diefendorff and Duquesne, 2002). Our targeted products will typically have a system load consisting of a mixture of concurrently active tasks to occupy multiple CPU's in parallel. Furthermore, according to our experience, the audio and video processing algorithms easily allow an additional (lower) layer of thread-level parallelism by operating on multiple blocks of data in parallel, if this were needed to obtain sufficient spreading of CPU load. The use of multi-threading applications is becoming more-and-more accepted as generic programming model, as the trend towards chip-multiprocessing and CPU's with multi-thread facilities is picked up by all major processor developers (Halfhill, 2004). To support re-use of industry-standard software, the CAKE architecture creates a global uniform and coherent memory view towards its processors, in accordance with the general-purpose computing world.

Tiling: Tiling creates an extra hierarchy layer in the system hardware architecture. The top-level architecture view shows a regular structure of homogeneous tiles (subsystems) connected through (for instance) a two-dimensional torus network (see Figure 3-2). For background information see the section on 'Static Networks' in Flynn (1995), or see Dally and Towless (2001). A tiled architecture allows a trivial instantiation of both large (expensive) and small (cheap) products with scalable compute performance for little silicon design effort. Tiles can simply be replicated in layout, without the need for extensive verification per product instantiation.

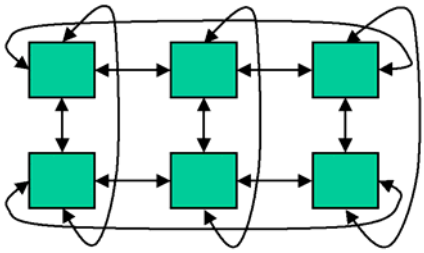


Figure 3-2. Tiling with a torus network

Torus structures are trivially mapped on silicon without long wires, creating predictable and scalable performance. If torus links are made available off-chip, then current chips can serve to create a prototype for next-generation silicon, to perform prototype software mapping and performance analysis. Inside a tile, a subsystem is created with a heterogeneous set of processors and an embedded memory, in complexity comparable with today's large SoCs. Heterogeneous processors allow a choice of processors for different types of tasks, thereby improving the computational efficiency (energy required to execute the task). The embedded memory is needed to ensure that most memory accesses of the processors can be served by tile-local memory, which avoids the access latency induced by the inter-tile network or off-chip access. The network latency would otherwise kill processor performance. Furthermore, having most data locally in the tile reduces power consumption for data transport, and saves precious bandwidth to off-chip bulk memory. With the growth of on-chip aggregate processor performance, it is almost inevitable that total on-chip memory needs to grow in capacity, taking an increasing percentage of chip area (Nair, 2002). Proper use of streaming programming models for our media processing target domain, and support for tile-local stream-buffer allocation and stream-management over the inter-tile network, will help to fight-back the growing on-chip memory needs and thereby have more effective compute resources on given silicon area. Automatic prefetching of streaming data into the cache will help to hide the latency of the memory hierarchy with little effort for the application programmer (van de Waerdt et al., 2005).

Redundancy and self-test: With SoC complexities now in few hundred millions of transistors, breaking the one billion transistor mark soon, it becomes unrealistic to expect that after fabrication every individual transistor will indeed work correctly as expected. Aiming for perfect products would result in an uneconomical loss after fabrication and test. The CAKE architecture advocates multiple identical processors and memory blocks in every tile, and multiple identical tiles on chip. Clearly, if one

processor out of ten (or a hundred) would be defect, the chip can still serve a quite useful computational load. Also if a one-megabit memory instance is faulty (after trying to exploit its built-in redundancy), the remaining memory blocks could still support useful program execution. Similarly, at the top-level architecture, an entire faulty tile might not block overall use. Only a very small percentage of the chip area is really unique and indispensable, so there is only an extremely small chance that the chip will be really dead. A chip that is loaded with identical programmable compute facilities will have the intelligence and processing power to perform self-test at every cold boot-up of the system, and can configure itself to avoid the use of its faulty parts. This allows the semiconductor vendor to sell perfect products for a premium price, and sell the other products for price-sensitive applications with hardly any production loss. The consumer could observe a small degradation of its appliance over time, and buy new equipment before a total break down occurs. In that sense, consumer electronic devices would 'wear out', and could be treated with an attitude similar as clothes, furniture, and cars.

Creating chip multi-processors (CMPs) is a rapidly growing trend in industry: the trends as sketched above have broad applicability. Both old and modern examples are too numerous to list here (Halfill, 2004). A few nice examples that target the embedded market are the 4-core ARM module with L1-cache snooping (Krewell, 2004), a 4-core PowerPC embedded in an FPGA (Kowalczyk, 2003), a 4-core MIPS with shared L2 cache and off-chip links for PCB-level tiling (Wong, 2002), and of course the Sony/IBM Cell architecture that also advocates tiling for scalability (Pham et al., 2005; Suzuoki and Yamazaki, 2002).

The first test-chip that is currently designed according to the CAKE architecture, will contain a single tile only. Off-chip links allow the creation of a multi-tile system at PCB (printed circuit board) level. The on-chip (tile) memory is currently designed as a shared level-2 cache, which allows flexible use through software configuration. For processors, a recent version of Philips' TriMedia processor is used (van de Waerdt, 2005). The TriMedia processor is optimized for audio and video media processing, and features a high computational density and power efficiency in its application domain.

Section 2 will provide more information on the CAKE architecture, in particular regarding the tile-local network and its cache coherency. Section 3 will describe approaches towards parallel programming, and shows some benchmarking results. Section 4 summarizes the current state of the project.

2. CAKE TILE ARCHITECTURE

The CAKE tile architecture comprises of multiple CPUs, various IP blocks, a shared L2 cache, and the interconnect network, see Figure 3-3 below. IP blocks may be simple coprocessors (e.g. image enhancements, video scalars, MPEG decoders, etc.), programmable processors, or complete subsystems with multiple processing units and local memory.

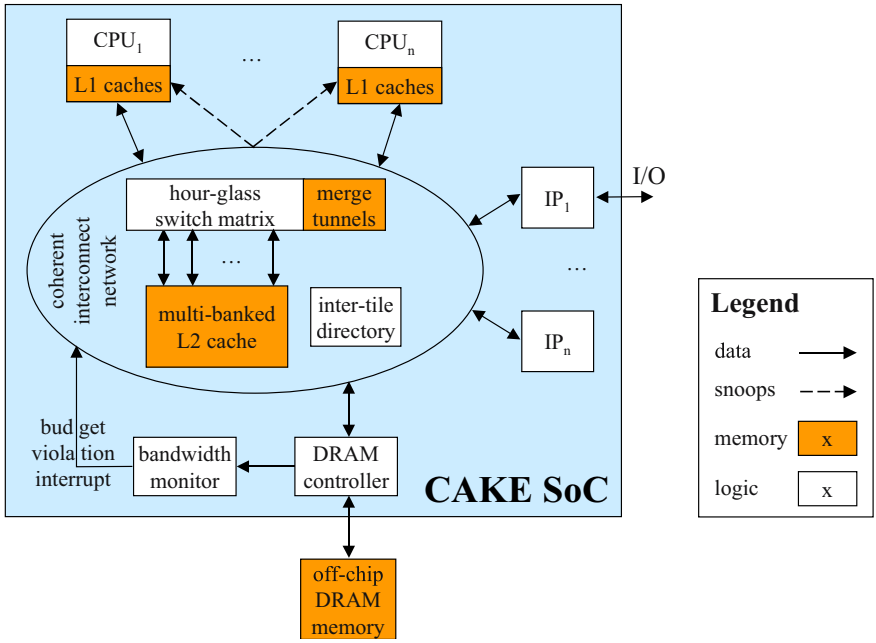


Figure 3-3. CAKE Tile architecture

CAKE's tile-local *coherent interconnect network* is capable of serving multiple concurrent transactions from the attached L1 caches and coprocessors. These transactions follow Philip's 'MTL' or ARM's 'AXI' protocols for memory requests, typically transferring a sequence of data-words per request. The tile network supports multiple outstanding requests per port, with out-of-order completion. The block-transfers are supported with 'critical word first' delivery to reduce the CPU stall time (cache miss

penalty). The interconnect network is created by a set of 'hour glass'[1] structures between the various groups of components. The 'hour glass' structures provide sufficient parallel data transfers to sustain processor performance, without unnecessary overhead of a fully-connected switch matrix. These 'groups of components' are (a) components that connect to the network:

– The various CPUs and Intellectual Property (IP) modules (co-processors).
– DMA engines that create data transport through inter-tile links.
– Different ports of the DRAM controller.

and (b) components internal in the network:

– Multiple L2 cache banks, selected through address interleaving.
– Merge tunnels, used for direct (demand driven) inter-processor communication.

The main objective of this network is to provide a low-latency access from processors to their requested data. The Register Transfer Level (RTL) code for this interconnect network is generated from a set of parameters, to create different CAKE instances from the same code base. The chip that is currently in design will probably connect 9 TriMedias, one ARM or MIPS processor, 4 function-specific (video-) coprocessors, 4 PCI-Express interfaces, and 8 L2 cache banks, through 64-bit data paths. All processors and the tile-network are targeted to run at a 350MHz to 450MHz clock rate in the CMOS 65-nm process chosen for this test-chip. (The achievable clock rate is strongly influenced by the SRAM performance, which is -at the time of writing- not accurately known for the targeted 65-nm process.)

To maintain sustained throughput, the interconnect and L2 control implement *hit-under-miss*. According to this policy the system does not block on an L2 miss. In fact, the interconnect keeps serving subsequent transaction(s) from other CPUs, in parallel with handling L2 refills. Furthermore, if the L2 miss is not caused by a demand load (but, for instance, a prefetch), the corresponding CPU does not get blocked either. To enable seamless integration of diverse CPUs running at various clock frequencies in respective islands of synchronicity, the interconnect network talks to the CPUs and IPs via *asynchronous clock domain bridges.* To avoid clock skew problems during back-end design, source-synchronous communication channels are employed to connect CPUs and IPs that are located further away from the central coherency controller and L2 banks.

CAKE SoC interconnect network scales well to about 20 or 30 CPUs/IPs: beyond that the routing area and cache snooping traffic becomes harder to

[1] An 'hour glass' network provides connections between two sets of ports, through two switch matrices and intermediate set of '*n*' ports. This allows to dynamically configure paths between any of the two sets of ports, up to a maximum of '*n*' simultaneous paths.

manage. To enable further scalability we employ *tiling* with distributed shared memory (DSM) across multiple tiles (See for instance the section on "Scalable Multiprocessors" in Flynn, 1995). Inter-tile communication is controlled by a directory-based coherence protocol with some support from firmware to handle remote misses. Each tile maintains a directory table. Each entry of the table specifies where the corresponding memory block resides (locally or remotely). By employing a cache snooping protocol inside a tile, and supporting a directory-based protocol between tiles, we believe the system is scalable to support beyond a hundred processors for the applications in our targeted audio and video-processing domain (which in general have little data dependency). In case of an L2 miss on a memory block from another tile, the hardware calls a firmware routine that orchestrates the inter-tile transfer. Such a transfer is typically carried out as message passing via DMA links and requires a few hundred cycles in our simulation model. The first CAKE chip will implement the inter-tile links as off-chip point-to-point PCI-Express channels. Using the global inter-tile coherent memory view is functionally transparent for the application programmer.

Compared to many prior-art embedded SoC designs (de Oliveira and van Antwerpen, 2003; Paver et al., 2004) the CAKE SoC relies on its shared *L2 cache*, to achieve several advantages. First, it saves on off-chip DRAM bandwidth and associated power dissipation by serving many data accesses from L1 caches and coprocessors and keeping communication on-chip. Second, the L2 cache decreases the penalty of L1 misses to a few dozens of cycles, thus increasing processor performance. Third, the L2 cache efficiently transfers data on and off the chip automatically in chunk sizes appropriate for the off-chip DDR, independent of smaller request sizes of the individual CPU's and coprocessors. This allows efficient use of DDR bandwidth, while re-using older (co-)processor designs that still employ smaller block sizes.

A SoC has multiple on-chip memories and CPUs with caches. Hence, the inevitable *cache coherence* problem (Hennessy and Patterson, 2003). For example:

a) If a process in CPU 'A' stores a value to a memory address, this updated value might remain inside A's cache for a while. If later a process on CPU 'B' wants to read the value in this address, it might miss in its cache and fetch an out-dated value from main memory.
b) A process in CPU 'A' has stored a new value to a memory address, and also (after a while) copied this data back to main memory. If a process on CPU 'B' wants to read this value, it might find a hit in its local cache and still use an out-dated value.

Note that this cache coherency problem even persists with a single process (or thread) without any inter-process communication: An SMP process

scheduler might stop a process and later re-schedule it on a different CPU causing similar cache coherency problems.

The CAKE SoC architecture provides hardware cache coherence, exposing a simple shared memory model to the programmer, functionally equivalent to a single-CPU design. Our cache coherency is implemented using *snooping* in the tile interconnect, according to the standard MSI protocol outlined in Figure 3-4 below. Each L1 cache line can be found in one of the three states: Invalid, Shared or Modified. The state changes are caused by activities of the attached CPU (shown in solid lines) and by snoop requests that arrive from the other CPUs (shown in dashed lines). The cache coherence protocol ensures only a single exclusive copy of the cache line being modified. There can, however, be multiple copies of a cache line in the Shared state. (For more information, see the section "Memory Coherence in Shared Memory Multiprocessors" in Flynn, 1995.)

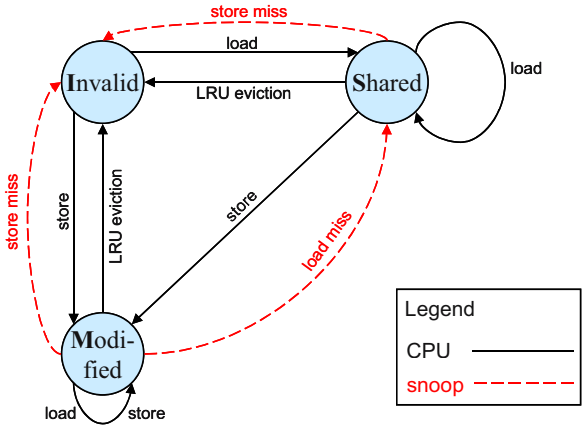


Figure 3-4. MSI protocol state transitions

The tile network broadcasts a snoop request upon receiving a transaction request from a CPU or an IP. Then the coherent CPUs reply to the requests with the status of the cache line in question. Our design can sustain a new snoop request every clock cycle. To have a snoop-answer within a few clock cycles, without disturbing the normal processing of the snooped CPU's, the L1 cache tags are duplicated into 'shadow tag' structures that are stored in the clock domain of the network (as opposed to the clock domains of the respective CPUs). Adding the copied shadow tags and snoop logic creates

about 3% area overhead relative to a TriMedia CPU. For this small overhead the SoC obtains an industry-standard programming model, and time-consuming programmer effort is saved by avoiding complex SW porting issues and intriguing (non-reproducible) software bugs.

Snooped CPUs might need to start a *snoop action*, according to the MSI protocol, shown as dashed arrows in Figure 3-4. For example, if the snoop request asks the exclusive rights for a cache line and the line is found dirty in another CPU, then that other CPU performs the snoop action of write-back and invalidate of that line. Note that non-coherent CPUs and IPs also benefit from the coherent network, when they stream in data produced by the coherent CPU cluster. The *inter-CPU communication* (such as streaming data) is naturally realized by accessing the shared memory. First, the producer CPU creates data in its local L1 cache. Then, the consumer CPU asks for this data from the interconnect, which finds it in the L1 of the producer CPU through snooping. Finally, the interconnect network streams the data to the consumer CPU via a 'merge tunnel' (see Figure 3-3). The merge tunnel can combine several cache lines from different sources (L1, L2, DRAM), depending on the requested block size. The programmer does not have to explicitly program the data transport.

The L2 cache is currently targeted to have a capacity of 2 MByte, requiring about 16mm^2 of high-density SRAM. For high-definition video applications, embedded DRAM is an attractive option, allowing significantly larger capacity in the same silicon area. Unfortunately, availability of embedded DRAM is -at the moment of this writing- uncertain at our targeted tape-out date, but can be reconsidered for a later product. The L2 implementation furthermore requires tag storage of 0.25mm^2 and negligible cache control logic. These area figures are to be considered relative to the CPU area, which is roughly 3mm^2 for each TriMedia or ARM processor, depending on instantiated CPU variation and its L1 cache parameters.

Task *synchronization* heavily relies on coherency too. The coherent CPUs may rely on two well-known operations LL (Load Linked) and SC (Store Conditional) to perform synchronization without stalling the memory subsystem (Hennesy and Patterson, 2003). In particular, these operations allow to easily create memory-mapped semaphores. The implementation of the LL/SC operations relies on cache coherency, which provides the LL operation with the freshest data without interfering with other CPUs. SC in turn consults the L1 cache line status and reuses the snooping mechanism to complete the atomic LL/SC pair. These two operations have been added to the TriMedia instruction set, to support efficient inter-thread synchronization and easy porting of external software that relies on memory-mapped semaphores.

CAKE SoC enables a predictable *compositional* system design through advanced resource management. The resource management ensures that

integration and use of many software and hardware components that share resources (caches, DRAM bandwidth, CPU cycles) does not affect the expected performance of the individual components, or at least protects the performance of critical components by proper management. The resource management in CAKE includes DRAM bandwidth and L2 *cache footprint management* by explicit partitioning of the cache space (Perez, 2005; Molnos, 2005). All data transfers from the CPUs are accompanied by a task descriptor, which selects a way-mask for the L2 cache. This way-mask protects a subset of the ways for victim assignment upon a cache miss. In other words, a L2 cache refill on behalf of one task can be prevented to evict a cache line of another critical task. Hence, task interference in the L2 cache is minimized. Note that cache hit detection uses all ways, thus enabling inter-task communication.

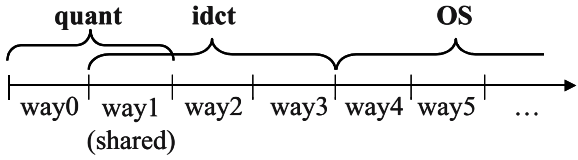


Figure 3-5. Way partitioning among the tasks

Task descriptors also identify the bandwidth domain the task belongs to. CAKE SoC provides bandwidth monitor registers keeping track of the *DRAM bandwidth* utilization per domain. When a domain has exhausted its allocated bandwidth budget, this is signaled to the (software) Quality of Service manager. The Quality of Service manager can decide to scale down the service of the violating domain if other system components are negatively affected. Task sharing of the DRAM bandwidth is limited by the respective bandwidth budgets controlled by dedicated bandwidth monitor registers. The hardware facilities to allow bandwidth and cache space control are now in detailed design, the management methods are clear (Perez, 2005), but actual verification of the overall system behavior and its tuning is yet to be performed.

3. PROGRAMMING METHODOLOGY

Application programming models are a highly evolving area of research (Lee, 2002), and each type of programming model is tailor-made to exploit

certain properties, which are essential for a particular class of applications and the targeted hardware. The multiprocessor features in CAKE can be exploited by parallellizing single application or running many applications concurrently and using the underlying cache coherent network for communication. Various types of programming models have been proposed which can be broadly classified into streaming models (detailed reference to various models are found in van der Wolf et al. (2004)) and non-streaming models like series-parallel graph, and Finite State Machine (FSM).

Kahn Process Network (KPN) and its variants are one class of programming models appropriate for modeling streaming multimedia and signal processing applications (van der Wolf et al., 2004). KPN based models have simple and well defined interfaces for high-level abstract specification of the application and thus are suitable for running on heterogeneous systems. The simple interface provided by KPN based models facilitates reuse of the streaming component for different applications. Unfortunately, KPN based models might have unpredictable execution time, deadlocks, or overuse of resources like memory. Another method of expressing concurrency is POSIX thread or Pthread (Nichols et al., 1998). Pthread provides a number of low-level, low-overhead primitives supporting multithreading and flexible synchronization between threads. Pthreads assume that all threads share a uniform address space for inter-thread communication and the underlying architecture should support this shared memory concept. Pthreads support highly dynamic thread creation by the application and is a widely used industry standard for shared memory machines. But Pthreads have a low level of abstraction, allowing programming bugs that are hard to trace. Yet another model called SDF (Synchronous Data Flow graph) [Sri2000] is popular within DSP community because of the useful property that deadlock and resource requirements are decidable or determinable. But not every class of applications can be efficiently expressed by means of SDF with reasonable effort.

Consequently we see that a platform such as CAKE to be widely usable across a range of application domains, should efficiently run a wide variety of programming models. Currently the CAKE architecture can support parallel models like Trimedia Streaming Software Architecture (TSSA) (de Oliviera and van Antwerpen, 2003), C-heap, Y-chart Application Programming Interface (YAPI), and Task Transaction Level (TTL) (van der Wolf et al., 2004) and Pthreads. Restricting an architecture or platform to run only one kind of programming model like SDF seriously hinders broad acceptance of the platform, because other models will run inefficiently. Also, complex applications need the support of mixed programming models

because different parts of the application might be expressed more naturally by different programming models (Lee, 2002). These application modeling issues prompted the CAKE architecture to support a shared-memory concept both inside and over the tile, as well as explicit streaming for more efficient inter-tile communication. Also, the cache-coherency eases the burden on programmer to bother about the location of the latest data in the system.

The application design trajectory or flow shown in Figure 3-7 starts by explicitly capturing the parallelism or expressing the parallelism possible in the application. The CAKE architecture uses the Trimedia compiler that does very well in extracting the Instruction Level Parallelism (ILP), but extracting higher levels of parallelism is either manual or semi-automatic. Once the parallelism possible in the application is analyzed it can be captured using some of the programming models like KPN, Pthreads etc. mentioned in the previous paragraph.

It is widely known that parallelism can be extracted by functional-, data- or mixed partitioning. In a functional partitioned model (such as KPN) each task (thread) performs a distinct function in a pipelined fashion and communication between tasks is made explicit (see Figure 3-7). A functional partitioned model

– Facilitates easy reuse of the functional modules for other similar applications.
– Intuitively fits with the streaming application description.
– Is an appropriate model for systems where some tasks are implemented on function-specific coprocessors.

The main disadvantages are:

– Possibility for load imbalanced partitioning where a single task limits the speedup.
– Inter-task communication might consume high bandwidth, since streaming data needs to travel between processors.
– Large human effort is required for conversion of a monolithic sequential application into a functional parallel application because all communication between the functional modules need to be made explicit. This effort in practice limits the amount of obtainable parallelism.

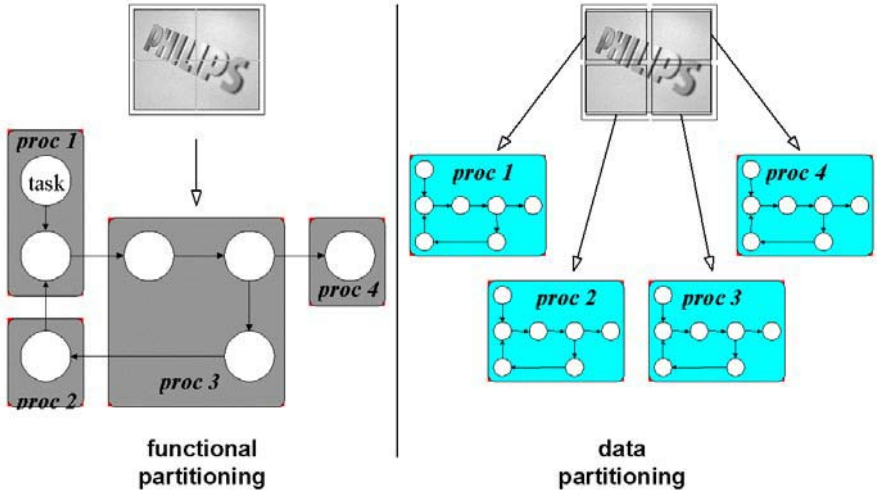


Figure 3-6. Functional versus Data Partitioning

Alternatively, data partitioned application models process different input data simultaneously by different threads executing the same function (see Figure 3-6). A data partitioned model has various advantages

– Good scalability: When the available number of processors changes then only the data distribution needs to change without changing the function implementation. Often, no code changes are needed at all, as the number of threads might be dynamically determined.
– Efficient usage of cache resources: The size of the data partition can be selected to maximize data cache hit ratio, reducing system communication.
– Allows natural load balancing depending upon the workload.

But also data parallelism extraction needs thorough study of each application and is algorithm specific. The exact scheme by which the data is partitioned and distributed to different tasks determines the performance. Still a major part of the data parallel models can be reused across various applications if within each data partitioned model some kind of modularity exists with a clean interface and encapsulation. So from our experiments we found out that data parallel models scale well for increasing or decreasing resources and also varying input streams resulting in natural load balancing and resource utilization. In practice a combination of functional and data parallelism will be applied. We foresee functional parallelism at coarse granularity, such as a video codec, an audio codec, or an image improvement

algorithm. Inside those functions, multi-threaded data-level parallelism can be exploited if needed to achieve real-time throughput.

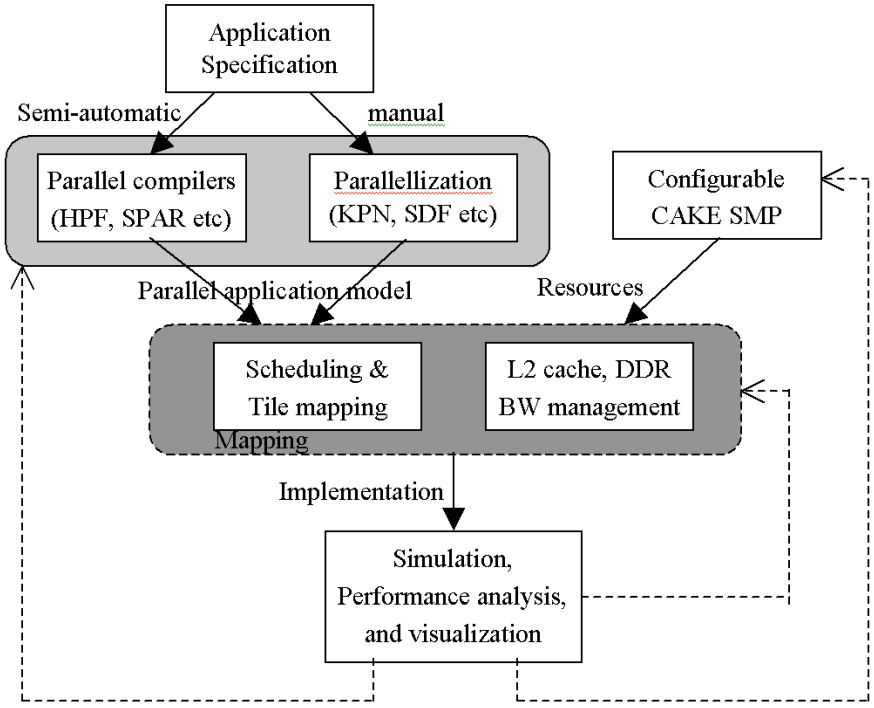


Figure 3-7. Application development trajectory

Once the parallel model of the application is developed, a mapping or binding process is carried out. Earlier the CAKE architecture was running a dedicated distributed, light-weight operating system kernel called the TRT (Tile RunTime) (Stravers and Hoogerbrugge, 2001), that supported fine-grain synchronization and fast context switching, but lacked real-time support. Recently we ported the SMP (Symmetric Multi Processing) version of the open-source, embedded real-time, operating system eCos (Massa, 2002). Currently, the CAKE software implements static task partitioning across the tiles and dynamic task scheduling inside the tiles. But other kinds of scheduling like static or quasi-static (Sriram and Bhattacharyya, 2000) can also be implemented, since the thread-scheduler is a well-separated functionality of the eCos system. To many applications where real-time constraints are not involved, dynamic scheduling offers the best usage of

available CPU resources with quickest possible design integration. Other applications, where real-time deadlines and throughput constraints are more difficult to meet, applications need to be tuned for reserving shared resources (CPU cycles, DDR memory bandwidth, L2 cache footprint), for improving the ILP (Instruction Level Parallelism) by source-code optimizations, and for modifying the data or function thread-level parallelism. Even with a fully programmable SMP, sufficient real-time guarantees could still be achieved by means of efficient resource management (see Perez et al., 2005).

We mapped various applications like an MPEG-2 decoder (Stravers and Hoogerbrugge, 2001), an MPEG-2 encoder, 3D-TV rendering algorithms, an Open-GL 3D-GFX library, an H.264 decoder (van der Tol et al., 2003) and the SPLASH-2 benchmark (Woo et al., 1995), and found good scalability and performance on the CAKE architecture. Furthermore, the 'Archtest' program (Collier, 1992) was mapped to thoroughly verify our memory consistency.

High-definition MPEG-2 decoding was easily parallellized by forking a new thread for the decoding of every slice. MPEG-2 ensures that slices can be decoded in parallel, and that slices span at most 16 video lines. As result, an HD image (1920x1080 pixels) can be processed with 68 threads in a data-parallel mode with very minor code changes.

Figure 3-8 shows the performance of the CAKE architecture for the SPLASH-2 benchmark suite. The simulations were performed with different numbers of TriMedia processors, which share an 8-bank 12MB L2 cache. The individual processors were configured with 16KB of L1 data cache with 128-byte line size. The benchmark code was run straight out-of-the-box, without any adaptation. We can see from the Figure 3-8 that even without any optimization the benchmark scales well. Further study is still needed to evaluate the details of these results.

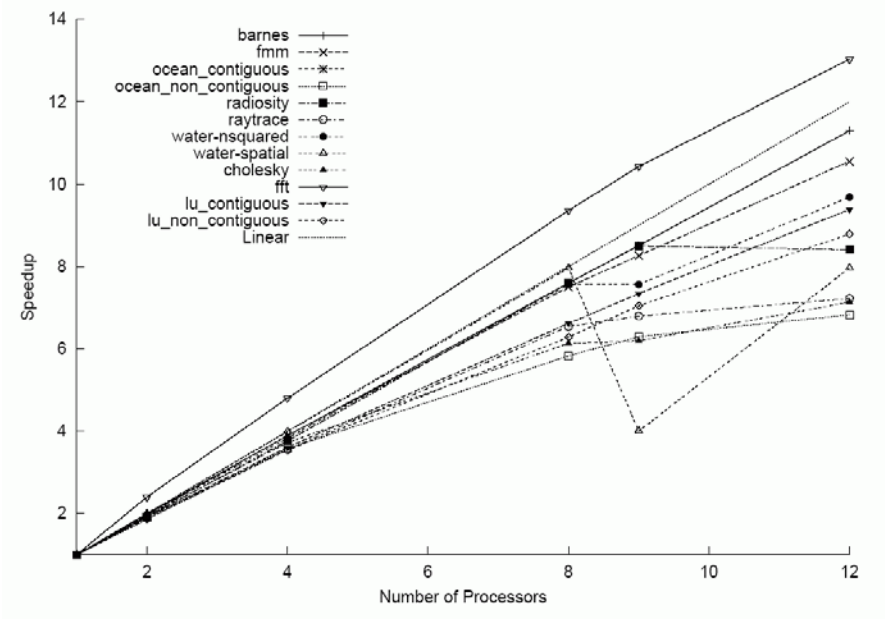


Figure 3-8. Out-of-the-box SPLASH-2 benchmark results

Clearly, many possible systems can be created by mixing HW architectural features with programming paradigms. In our view however, two particular combinations seem to fit naturally together:

a) Homogeneous multi-processors, in a shared-memory architecture, employing data-parallelism, and relying on dynamic task-to-CPU mapping.
b) Heterogeneous multi-processors, in a message-passing architecture, employing function-level parallelism, using static task mapping.

Philips' SoCs for the embedded consumer-electronics (CE) domain have traditionally used the latter (b) style, mainly because of the high efficiency of function-specific processors. The (a) style has become prominent in general-purpose computing because of its flexibility and ease of programming. With the growing transistor-count per die, and the growing amounts of embedded software, we foresee a growing trend towards (a) also for the embedded CE market.

4. STATUS AND CONLUSIONS

The CAKE project has a tool suite up and running, partially consisting of standard tools, partially newly created software. The complete suite operates in a Linux environment and contains:

- A TriMedia compiler chain, a MIPS compiler, and a shared-object linker.
- Embedded software for a low-level boot-up of the chip, and the eCos embedded OS with our own HAL (hardware abstraction layer) for support of timers, interrupt controllers, and other device drivers.
- Application software to be executed by the embedded processors for:
  a) Testing correct behavior of the hardware. Besides basic CPU tests, multi-processors data exchange test are used, such as the generic 'Archtest' program (Collier, 1992).
  b) Media processing application software that is used for performance measurements.
- The parameterized simulator with an almost cycle-accurate system model, built on top of the SystemC simulation kernel.
- Several (interactive graphical) tools for evaluating the simulated system performance based on off-line visualization of generated trace files.

Currently RTL (Verilog) code is being made for a research test-chip that implements a single tile of the CAKE architecture concepts. Individual Verilog modules are tested through co-simulation, embedded in the overall SystemC system model. Tape-out is expected in a CMOS 65-nm technology by the end of 2005. Creating this chip and its initial applications is done as a cooperative effort between Philips Research, Philips Semiconductors and Philips Consumer Electronics.

Parallel benchmarks like SPLASH-2 and Archtest did run on the CAKE platform without any modification. eCos itself was ported, including its optional libraries such as the Posix layer, with full multi-processing and real-time support, requiring only little effort for its platform-specific HAL. This proves that the project created a platform with industry-standard programmability, of which cache-coherency is an indispensable ingredient. As such, this platform paves the way for the software re-use as demanded by future embedded systems. The concepts and the architecture realized in this project will establish a strong basis for evolutionary product growth well into the next decade, reaching billions of transistors and hundreds of processors on a single die, in an economically sound way.

REFERENCES

Collier, W.W., 1992, *Reasoning about parallel architectures*, Prentice-Hall, 1992 (See also http://www.mpdiag.com/)

Dally, W.J., and Towless, B., 2001, "Route Packets, Not Wires: On-chip Interconnection Networks", *proc. DAC2001 38th Design Automation Conf.*, pp. 684-689, Las Vegas, USA, Jun. 2001

Diefendorff, K., and Duquesne, Y.,, 2002, "Complex SoCs require new architectures", *EEdesign*, Sep. 2002

Flynn, M.J., 1995, *Computer Architecture: Pipelined and Parallel processor design*, Jones and Bartlett Publ., 1995

Halfhill, T.R., 2004, "Deluge of Multicore Processors for PC's, Servers, Embedded Systems", *Microprocessor report*, Vol. 18, pp. 16-20, Sept. 2004

Hennessy, J.L., and Patterson, D.A., 2003, *Computer Architecture: A Quantitative Approach (3rd ed.)*, Morgan Kaufmann Publ., 2003

Paver, N.C., Khan, M.H., Aldrich, B.C., 2004, "Accelerating Mobile Multimedia with the Intel PXA27x Processor Family", in: *Workshop on Media and Signal Processors for Embedded Systems and SoCs (MASES),* in conj. w. CASES 2004*,* Washington DC, USA, Sept. 22-25, 2004

Keutzer, K., Malik, S., Newton, R. and Sangiovanni-Vincentelli, A., 2000, "System-level design: Orthogonalization of concerns and platform-based design," *IEEE Trans. On CAD of Integrated Circuits and Systems*, Vol. 19, No. 12, pp. 1523-1543, Dec. 2000

Kowalczyk, J., 2003, "Multiprocessor systems", *White paper: Virtex-II Series*, Xilinx WP162, Apr. 2003

Krewell, K., 2004, "ARM Opens Up to SMP", *Microprocessor Report*, Vol. 18, pp. 1 and 5-7, May 2004

Lee, E.A., 2002, "Embedded Software", *Advances in Computers,* Vol. 56, Academic Press, London, 2002

Massa, A.J., 2002, *Embedded Software Development with eCOS*, Prentice Hall, 2002 (see also: http://ecos.sourceware.org/)

Molnos, A., Heijligers, M., Cotofana, S.D., van Eijndhoven, J.T.J., 2005, "Compositional memory systems for multimedia communication tasks", accepted for publ. in: *proc. Design Automation and Test in Europe (DATE)*, Munich, Germany, Mar. 2005

Nair, R., 2002, "Effect of increasing chip density on the evolution of computer architectures", *IBM J. on Research and Develop.,* Vol. 46, No. 2/3, pp. 223-234, March/May 2002

Nichols, B., et al., 1998, *Pthreads Programming*, O'Reilly Publishers, 1998

de Oliveira, J.A., and van Antwerpen, H., 2003, "The Philips Nexperia Digital Video Platform," in *Winning the SoC Revolution,* G. Martin and H. Chang, Eds., pp. 67-96. Kluwer Academic, 2003.

Perez, C.O., et al. , 2005, "Resource reservations in shared-memory multiprocessor SoCs", *chapter 5 in this book*

Pham D. et al., 2005, "The Design and Implementation of a First-Generation CELL Processor", accepted for publ. in: *2005 IEEE Int. Solid State Circuit Conf. (ISSCC),* Feb. 6-10, San Francisco, USA

Sriram, S and Bhattacharyya, S.S., 2000, *Embedded Multiprocessors: Scheduling and synchronization*, Marcel Dekker Inc.

Stravers, P., Hoogerbrugge, J., 2001, "Homogeneous multiprocessing and the future of silicon design paradigms", *Proc. Of International symposium on VLSI Tech. Systems and applications*, 2001

Suzuoki. M., and Yamazaki, T., 2002, "Computer Architecture and Software Cells for Broadband networks", *US Pat. Appl. 2002/0138637A1*, Sep. 2002

Van der Tol, E.B.,, et. al, 2002, "Mapping of H.264 decoding on a multiprocessor architecture", *Proc. of SPIE conf. on Image and Video Communication*, Vol. 5022, Jan. 2003

van de Waerdt, J.W. et al, 2005, "Motion Estimation Performance of the TM3270", accepted for publication in: *proc. 20th ACM Symp. on Applied Computing (SAC2005)*, Santa Fe, New Mexico, Mar. 14-17, 2005

van der Wolf, P. et al., 2004, "Design and programming of embedded multiprocessors: An interface-centric approach", *proc. CODES+ISSS'04*, Stockholm, Sweden, pp. 206-217, Sept. 2004

Wong, W., 2002, "Quad 64-bit Multiprocessors targets comm. applications", *Electronic Design*, Oct. 2002. (See also Broadcom 'BCM1480' product brief, Sep. 2004)

Woo S.C., et al., 1995, "The SPLASH-2 Programs: Characterization and Methodological Considerations", In: *Proc. of the 22nd Int. Symp. on Computer Architecture*, pp. 24-36, Santa Margherita Ligure, Italy, June 1995
(see also http://www-flash.stanford.edu/apps/SPLASH/)

Yang, P., et al, 2002, "Managing Dynamic Concurrent Tasks in embedded real-time multimedia systems", *proc. 15th Int. Symp. On System Synthesis (ISSS'02),* Kyoto, Japan, pp. 112-119, ACM Press, 2002

Chapter 4

DATAFLOW ANALYSIS FOR REAL-TIME EMBEDDED MULTIPROCESSOR SYSTEM DESIGN

Marco Bekooij[1], Rob Hoes[2], Orlando Moreira[1], Peter Poplavko[2], Milan Pastrnak[2], Bart Mesman[1,2], Jan David Mol[3], Sander Stuijk[2], Valentin Gheorghita[2], and Jef van Meerbergen[1,2]

[1] *Philips Research Laboratories, Eindhoven, The Netherlands*

[2] *Eindhoven University of Technology, Eindhoven, The Netherlands*

[3] *Delft University of Technology, Delft, The Netherlands*

Marco.Bekooij@philips.com

Abstract Dataflow analysis techniques are key to reduce the number of design iterations and shorten the design time of real-time embedded network based multiprocessor systems that process data streams. With these analysis techniques the worst-case end-to-end temporal behavior of hard real-time applications can be derived from a dataflow model in which computation, communication and arbitration is modeled. For soft real-time applications these static dataflow analysis techniques are combined with simulation of the dataflow model to test statistical assertions about their temporal behavior. The simulation results in combination with properties of the dataflow model are used to derive the sensitivity of design parameters and to estimate parameters like the capacity of data buffers.

Keywords: real-time, dataflow analysis, multiprocessor system, predictable design, system-on-chip

1. INTRODUCTION

Consumers typically have high expectation about the quality delivered by multimedia devices like DVD-players, audio, and television sets. These devices process data streams and are often built using (weakly) programmable embedded multiprocessor systems for performance, cost, and power-efficiency reasons. The design and programming of these real-time multiprocessor systems should be such that the real-time constraints are met, and the desired

P. van der Stok (ed.),
Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, 81-108.

audio and video quality is delivered. These multiprocessor systems should be suitable for the simultaneous execution of audio and channel decoders as well as video decoders. The audio and channel decoders have hard real-time constraints because a miss of a deadline results in a click in the audio or loss of data which is unacceptable for the end-user. The video decoders have soft real-time constraints because if a deadline is missed then the video quality is reduced which is not appreciated by the end user but is to some extent acceptable.

The current design practice is that timing constraints of hard real-time applications are guaranteed by making use of analytical techniques while the (temporal) behavior of soft real-time applications is measured. As will be explained in the next paragraphs, these measurements do not make all the characteristics of soft real-time applications explicit which are usefull during the design process. Therefore we are concerned in this chapter with the use of dataflow models for the validation of the (temporal) behavior of applications with *soft real-time* constraints. These dataflow models are also key to derive a proper dimensioning of the multiprocessors system and to derive a proper mapping of the application onto the multiprocessors system. We claim that the use of these dataflow models reduces the number of design iterations and shortens the design time. Also our network based embedded multiprocessor system is presented. This system is suitable for the derivation of the temporal behavior of the application with dataflow models.

The applications executed on our multiprocessor system consist of jobs (see Figure 4-1). A job is an entity that processes a data stream. It is started and stopped by the user. The hard real-time jobs are indicated in this figure by dotted circles while the soft real-time jobs are indicated by dashed circles. A job is described by a dataflow graph. Such a dataflow graph contains actors that represent software tasks, or computations performed by a hardware component. Actors are started after sufficient input data and output space is available, such that they can finish their execution without having to wait for additional input data or output space. The edges denote communication of data between actors via First-In-First-Out (FIFO) buffers.

For soft real-time jobs such as video decoders, a tradeoff is typically made between the amount of resources that are made available and the deadline miss rate. Less system resources result in less hardware and a reduction of the hardware cost, but also result in a higher deadline miss rate and a reduced quality of experience for the end user. It is therefore an objective of the system designer to dimension and program the multiprocessor system in such a way that the quality is minimally compromised for a given resource budget.

The current design practice of systems that execute soft real-time jobs can be schematically depicted with a Y-chart (Kock, Essink, Smits, Wolf, Brunel, Kruijtzer, Lieverse and Vissers, 2000), as is shown in Figure 4-2. The dashed arrows in this figure denote design iterations. During an iteration a multipro-

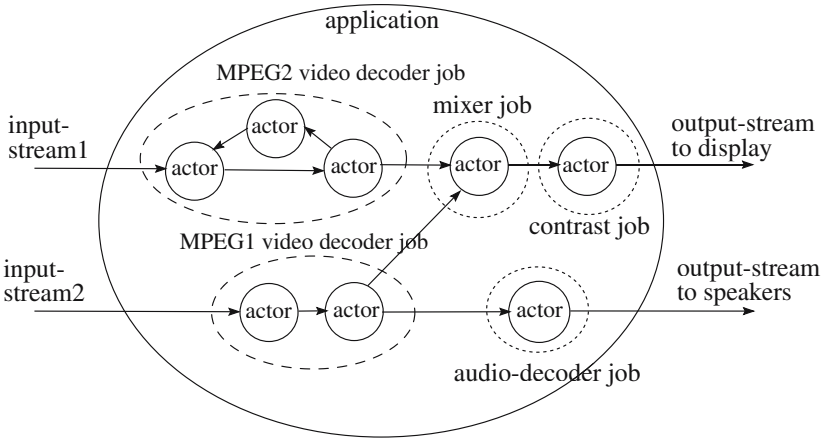


Figure 4-1. An application that consists of jobs. Jobs are started and stopped by the user. Jobs consist of actors that communicate via FIFOs. Hard real-time jobs are indicated by dotted circles while soft real-time jobs are indicated by dashed circles.

cessor instance is (re-)defined, programmed, and evaluated by means of simulating the target application in a cycle true simulator. From the simulation results, the system designer tries to derive clues on how he can improve the system or its programming such that all design constraints are satisfied, which is indicated by the light bulbs in the Y-chart figure. The current design practice is that the design constraints are verified after simulation but to a large extent ignored during mapping.

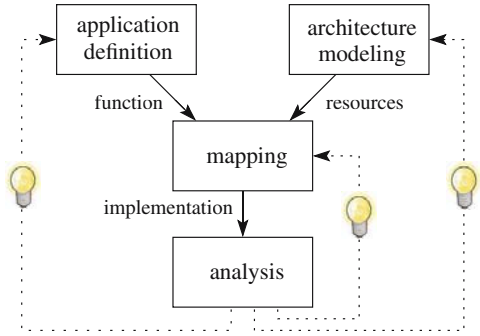


Figure 4-2. Y-chart programming paradigm.

Such a simulation based design process is cumbersome for modern applications and architectures due to the uncertainty in the amount of resources

demanded by the application at run-time and the uncertainty in the amount of resources supplied by the hardware. The resource demand fluctuates during execution because the amount of computation and communication performed by the application often depends on the content of the input stream. For example, the execution time of the actors depends usually on the values of the input data. This can also be the case for the amount of data communicated between the actors. Also, the amount of resources supplied by the hardware fluctuates due to arbitration of shared resources in the system. The term arb itration" is used in this chapter for the local scheduling of actors on processors, as well as for the policy used to resolve at run-time simultaneous requests for a shared resource, such as for example a communication bus or a memory port.

Another reason why this simulation based design process has become cumbersome is that the complexity of system-on-chip designs has grown much faster than the increase in speed of the simulators. This has resulted in slow design iterations in which usually only a small fraction of the system can be evaluated. It should also be noted that it can be very difficult to find performance critical corner cases in the design and generate the proper input stimuli to observe the system's behavior for these cases.

Another disadvantage of a simulation based design process is that it can be difficult to draw conclusions from the simulation results how to adapt the multiprocessor system's hardware or its programming. It can be difficult to draw conclusions because these multiprocessor systems can exhibit a highly non-linear behavior.

Finally, we would like to mention that it is difficult to reproduce the same temporal behavior with such a simulation based design process. The reason is that the initial state of the arbiters (e.g. Time Division Multiple Access (TDMA) arbiters) in the system is unknown at the moment that the job is started. Therefore, the order in which access to a shared resource will be granted by an arbiter is not known at compile time. A different order in which requests are granted can result in a completely different temporal behavior of a job in the case that the same job is started at a different point in time. This will make it for example impossible to reproduce the same temporal behavior with an (Field Programmable Gate Array (FPGA)) prototype of the system, which is currently often used to speed up the performance evaluation and debugging process.

In this chapter, we propose a multiprocessor system in which the uncertainty in the resource supply is bounded by enforcing resource budgets. A resource budget is for example a guaranteed amount of time to use a resource such as a bus or processor during a predefined period. These enforced resource budgets will make it possible to share resources, such as a port to background memory, between hard real-time and soft real-time jobs. These budgets also drastically reduce the effort to verify the temporal behavior of soft real-time jobs. The

reason is that given enforced resource budgets, the temporal behavior of one job cannot affect the temporal behavior of another job. This gives a job the illusion that it executes on its own private hardware, so it can be evaluated in isolation.

Given that resource budgets are enforced and guaranteed, then dataflow models and their corresponding analysis techniques can be applied to guarantee that hard real-time jobs will meet their deadlines. However these techniques are not directly applicable for soft real-time jobs because they require that a schedule can be derived offline. Such a schedule cannot be constructed for soft real-time jobs because the amount of resources that is provided for soft real-time applications is typically *less* than the worst-case amount of resources that are needed to meet all deadlines.

In this chapter we advocate the use of a mix of simulation and model based analysis techniques for the derivation of the temporal behavior of the soft real-time jobs. We show that dataflow models can be applied by demonstrating that if resource budgets are enforced that then the effect on the temporal behavior of run-time arbitration can be modeled in a dataflow model. These dataflow models can be used for soft real-time jobs to derive *conservative* arrival times of the data in the system by simulation of this dataflow model. During simulation the response times of the actors are used instead of the worst-case response times. The response time of an actor depends on the value of the input data of the actor. The arrival times of the data observed during simulation is *conservative* because data will not arrive earlier in the simulator than in the real system. There is no need to derive a schedule in advance because the execution order of actors is determined at run-time by the local schedulers/arbiters. The same dataflow models can be *analyzed* at compile-time to derive estimates of the effects on the throughput and latency of a job when a resource budget is adapted by the designer at compile time. An example of a resource budget is the capacity of a buffer.

2. RELATED WORK

In this work, dataflow models are used to derive the end-to-end temporal behavior of jobs. The focus is on synchronous dataflow (SDF) models (Lee and Messerschmitt, 1987), because it is currently the most popular and widely studied dataflow model for streaming applications with well defined semantics.

A similarity between SDF models and Kahn process networks (Kahn, 1974) is that they can be used to describe streaming applications. However SDF models are suitable for static analysis while Kahn process networks are unsuitable. Kahn process networks are unsuitable for static analysis because a Kahn process network is Turing complete. Therefore, questions of termination and bounded buffering are undecidable. That is, no finite time algorithm can

decide these questions for all Kahn process networks. This is illustrated with the Kahn process network example in Figure 4-3. In this example we assume that the behavior of process P1 depends on the values of the input data and is therefore unknown at compile time. We assumed also that the values of the input data are at run-time such that this process P1 will write one data word in FIFO1 after it has written 11 data words in FIFO2. We also assume in this example that process P2 reads first one data word from FIFO1 before it reads data from FIFO2. Deadlock of this process network occurs because process P1 cannot finish its writing of data in FIFO2 because the capacity of FIFO2 in the implementation is only 10 data words. Therefore, processes P1 will never be able to write data in FIFO1 such that process P2 can first read data farom FIFO1 and then from FIFO2. It should be noted that FIFOs with a finite capacity should be represented in a Kahn process network as two FIFOs with an infinite capacity. The data producing process stores tokens filled with data in one FIFO while the data consuming process stores tokens which indicate space in the other FIFO.

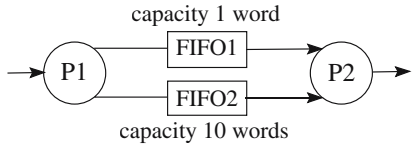


Figure 4-3. Example of Kahn process network which deadlocks due to insufficient FIFO capacity.

Another reason why Kahn process networks are unsuitable for static analysis is that a Kahn process blocks after it did a read attempt on an empty FIFO. A Kahn process that blocks must be preempted such that other processes on the same processor can continue their execution and produce the required input data. The number of times that a process blocks, depends on the run-time schedule and can strongly fluctuate. Therefore it is usually not possible to derive a tight bound on the preemption overhead at compile time. However a tight bound on the overhead due to preemption can be derived for SDF actors. The reason is that an SDF actor does not start its execution before all input data is present to finish its execution. Therefore SDF actors never block during their execution.

The SDF graphs are used in this chapter as a short hand notation of event graphs which are a special case of Petri nets (Petri, 1962). The temporal behavior of event graphs can be derived with MaxPlus Linear System Theory (Bacelli, Cohen, Olsder and Quadrat, 1992). SDF models are in (Sriram and Bhattacharyya, 2000) applied for hard real-time jobs that do not share resources with other jobs. The execution order of actors on the same processor

is derived from an offline computed schedule. Similar techniques are applied in (Poplavko, Basten, Bekooij, Meerbergen and Mesman, 2003) for soft real-time jobs. Good results could only be obtained by taking measures to limit the difference between the typical and the worst-case response times of the actors. The reason for these good results is that if the difference in response time is small, then the offline computed execution order is close to the optimal execution order. In this chapter we assume that the processors support preemption and that the execution order of the actors is determined at run-time. This makes it possible to cope with large variations in the execution time of the actors, and will allow sharing of resources by actors of different jobs.

In this chapter we advocate the use of a mix of simulation and model based analysis techniques for the derivation of the temporal behavior of the soft real-time jobs. The analysis of the temporal behavior of soft real-time jobs is different from the analysis of hard real-time jobs. The objective of the analysis for hard real-time jobs is to derive the worst-case temporal behavior of the system, while for soft real-time jobs the objective of the analysis is to derive the typical temporal behavior. The typical temporal behavior of jobs depends on the values of the input data which are unknown at compile time. Therefore, purely model based analysis techniques for hard real-time jobs, such as the techniques in (Kopetz, 1997; Pop, Eles and Peng, 2002; Richter, Jersak and Ernst, 2003), are not directly applicable for the analysis of soft real-time jobs. The reason is that the actual values of the input data can be ignored during analysis of hard real-time jobs because the objective is to derive the worst-case temporal behavior for any possible input data stream. For soft real-time jobs the values of the input data cannot be ignored during analysis because the objective is to derive the typical temporal behavior for a representative input stimuli set. The use of probabilistic models, such as Markovian and Poisson models, for the derivation of the typical temporal behavior of soft real-time jobs is either too simple to characterize the important properties of the source and the system, or too complex for tractable analysis (Zhang, 1995; Sriram and Bhattacharyya, 2000). Therefore, simulation is used by us to estimate parameters such as the execution times of the actors and to test statistical assertions about the temporal behavior of a job that is executed on the system, in a similar way as done in (Hee, 1994).

The concept of reservation based resource allocation has been introduced by the real-time community in order to eliminate interference between the software tasks of soft real-time multimedia jobs that are executed on a single processor system. The enforcement of resource budgets is a service provided by the operating system kernel (Rajkumar, Juwa, Moleno and Oikawa, 1998). The size of the resource budget is determined during a (re)negotiation phase between the job and the operating system. In this work, we address multiprocessor systems in which the resource budgets enforcement is not centralized

but distributed. Resource budgets are reserved to eliminate interference between jobs such that it is possible to share resources between hard real-time and soft real-time jobs, as well as to obtain a so-called monotonic system (see Section 4). An important property of a monotonic system is that an increase of a resource budget of a job cannot result in a reduction of the throughput of this job.

3. OUTLINE OF THIS CHAPTER

The organization of this chapter is as follows. The properties of the synchronous dataflow (SDF) model are recapitulated in Section 4. Then in Section 5 a multiprocessor architecture is presented that is suitable for the derivation of the temporal behavior of jobs with an SDF model. It is shown in Section 6 that the effects on the temporal behavior of a job, due to TDMA arbitration, can be expressed in the SDF model. By simulating this SDF model, conservative and accurate arrival times of tokens can be derived. The same SDF model is analyzed in Section 7 in order to derive at compile time the sensitivity for variations in the execution time of actors on the throughput of the system. We show that adaptation of the capacities of the FIFO buffers can reduce the sensitivity for fluctuations in the execution times of the actors on the end-to-end temporal behavior of a job. To obtain tight bounds on the arrival times of data it may be necessary to make the conditional execution of actors explicit in the dataflow model. Section 8 introduces conditional constructs in the dataflow model that guarantee mutual exclusive execution of actors. The dataflow graph that is obtained is an analyzable version of a Boolean Data Flow (BDF) graph (Buck, 1993). It is shown that these BDF graphs can be analyzed with the SDF analysis discussed in Section 4. These conditional constructs are applied in Section 9 to make explicit that different actors are executed during I-frame and P-frame decoding in an H263 video decoder. Finally, we state the conclusions in Section 10.

4. DATAFLOW ANALYSIS

In this section we define the SDF model and recapitulate its properties. This SDF model is used in successive sections for the derivation of the temporal behavior of jobs that are executed on multiprocessor systems with similar characteristics as the multiprocessor system that is presented in Section 5.

Before the properties of an SDF model are stated, we first define an SDF graph as follows:

Definition 1 (Synchronous Data Flow Graph.) *The tuple* (V, E, d, P, O, I) *defines a Synchronous Data Flow (SDF) graph, where*

- *V is the set of nodes (actors),*
- *$E \subseteq V \times V$ is the set of directed edges,*
- *$d : E \rightarrow \mathbb{N}$ is a function describing the number of initial tokens on an edge $(u, v) \in E$,*
- *$P : V \rightarrow \mathbb{R}^+$ is a function describing the worst-case response time of actor $v \in V$,*
- *$O : E \rightarrow \mathbb{N}$ is a function describing the number of tokens produced on edge $(u, v) \in E$ by actor u for each execution,*
- *$I : E \rightarrow \mathbb{N}$ is a function describing the number of tokens consumed from edge $(u, v) \in E$ by actor v for each execution.*

An arbitrary SDF graph is depicted in Figure 4-4. The nodes in an SDF graph are called actors. Actors have a well defined input/output behavior and a worst-case response time. Actors produce and consume tokens. The edges represent dependencies. A token is a container in which a fixed amount of data can be stored and is depicted in Figure 4-4 as a black dot. If more than one token is (initially) present on an edge then the number of tokens (d) is specified next to the dot. The tokens are consumed in the same order as they are produced. However random access of the data inside a token is allowed. Each actor in Figure 4-4 is annotated with its worst-case response time. An actor is enabled after a predefined number of tokens is available on every input of the actor. An actor can fire (starts its execution) after it is enabled. The number of tokens that must be available is specified next to the head of the data edges. The specified number of tokens is consumed from the input edges of the actor before the execution of an actor finishes, that is, within the response time of the actor. The number at the tail of an edge denotes the number of tokens an actor produces before the execution of the actor finishes. A self-edge of an actor is used to model that the previous execution must be finished before the next execution can start. This self edge is given one initial token such that the next execution cannot start before the previous execution of the actor is finished.

An SDF graph can be transformed into a Homogeneous Synchronous Data Flow (HSDF) graph (see Figure 4-5) on which we perform the analysis. An algorithm that transforms any SDF graph into an HSDF graph is described in (Sriram and Bhattacharyya, 2000). An HSDF graph is a special case of an SDF graph, in which the execution of an actor results in the consumption of one token from every incoming edge of the actor and the production of one token on every outgoing edge of the actor.

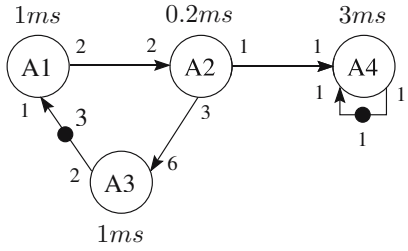


Figure 4-4. A Synchronous Data Flow (SDF) graph example.

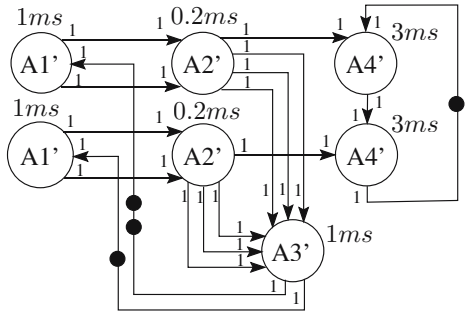


Figure 4-5. The Homogenous Synchronous Data Flow (HSDF) graph obtained after transformation of the SDF in Figure 4-4.

An HSDF graph can be executed in a self-timed manner, which is defined as a sequence of firings of HSDF actors in which the actors start immediately when there is at least one token on each input of the actor. In the case that the HSDF graph is a strongly connected graph and a FIFO ordering for the tokens is maintained between executions of the actors, then the self-timed execution of the HSDF graph has some important properties. A FIFO ordering is maintained if the completion events of firings of a specific actor occurs in the same order as the corresponding start-events. This is the case if an actor has a constant response time or belongs to a cycle in the HSDF graph with only one token. In (Bacelli et al., 1992) are the properties of the self-timed execution of such HSDF graphs derived with MaxPlus algebra.

First of all, the most important property of the self-timed execution of an HSDF graph is, that it is deadlock-free if there is on every cycle in the HSDF graph at least one initial token. Secondly, the execution of the HSDF graph (and also an SDF graph) is *monotonic*, i.e. decreasing actor response times result in non-increasing actor start times. The reason is that an earlier arrival time of a token cannot result in a later start of the actor that consumes this token.

Third, an HSDF graph will always enter a periodic regime. More precisely, a $K \in \mathbb{N}$, an $N \in \mathbb{N}$ and a $\lambda \in \mathbb{R}$, such that for all $v \in V$, $k > K$ the start time $s(v, k + N)$ of actor v in iteration $k + N$ is described by:

$$s(v, k + N) = s(v, k) + \lambda \cdot N \tag{4-1}$$

Equation 4-1 states that the execution enters a periodic regime after K executions of an actor in the HSDF graph. The time one period spans is $\lambda \cdot N$. The number of firings of an actor v in one period is denoted by N. Thus, λ is equal to the inverse of the average throughput measured over one period.

The Maximum Cycle Mean (MCM) (Sriram and Bhattacharyya, 2000) of an HSDF, which is equal to λ, is given by (4-2). The MCM of an HSDF graph is also called in literature the maximal cost to time ratio (Lawler, 1976). The Cycle Mean (CM) of a simple cycle c in the HSDF graph G is given by (4-3). In this equation denotes $d(c)$ the number of tokens on the edges in a cycle c. The Worst Case Response Time (WCRT) of actor v is denoted by WCRT(v). The MCM of an HSDF graph can be derived with a pseudopolynomial algorithm (Cochet-Terrasson, Cohen, Gaubert, McGettrick and Quadrat, 1998) with complexity $O(m|E|)$ with m the product of the out-degrees of all nodes.

$$\text{MCM}(G) = \max_{c \in C_G} \text{CM}(c) \tag{4-2}$$

$$\text{CM}(c) = \sum_{v \ on \ c} \text{WCRT}(v)/d(c) \tag{4-3}$$

The worst-case start-times of the actors during the transition state as well as the steady state can be observed during self-timed execution of an SDF graph in a simulator. During this simulation, all actors must have a response time equal to their worst-case response time. The start-times observed during this simulation are equal to the worst-case start times of the actors due to the monotonicity of the SDF graph. From (4-1) it follows that a periodic regime will be entered and therefore simulation can be stopped after the first period of the periodic regime. The SDF will enter a periodic regime because the HSDF graph that is obtained after transformation will enter a periodic regime. The SDF enters a periodic regime because the i-th start of an actor A1 in the SDF graph is as soon as all input tokens have arrived for this actor. All input tokens have arrived as soon as there is one token on each input of an actor A1' in the HSDF such that an actor A1' is started for the i-th time.

Actors in an SDF graph produce their output tokens exactly the WCRT after the actor is started. The input tokens are consumed and removed from the

input exactly the WCRT after the actor is started. Code segments in the implementation can be represented by an SDF actor in the model. Code segments produce the output tokens and consume the input tokens within the WCRT of the actor. The arrival times of tokens during selftimed execution of the SDF graph is not earlier then in the implementation due to the monotonic behavior of a selftimed executed SDF graph. Therefore an upper bound on the arrival time of tokens is observed during selftimed execution of the SDF graph.

We refer in this chapter to a code segment as an actor in the implementation. The actors in the implementation have a response time as well as an Execution Time (ET). The ET of an actor in the implementation is defined as the interval of time it takes to execute the corresponding code segment on a processor without that its execution is preempted. The execution time depends often on the values of the input data. The Worst Case Execution Time (WCET) is an upper bound on the execution time of an actor in the implementation and is derived with static program analysis techniques (Li and Malik, 1999).

It should be noted that the token arrival times during selftimed execution in the SDF simulator remain conservative if the Response Times (RTs) of the actors are used instead of the worst-case response times of the actors. The RT of an actor is an upperbound on the time interval between the point in time that the actor is enabled and that the point in time that the actor finishes its execution. The response time of an actor can depend on the values of the input data that are consumed during that execution. The token arrival times during selftimed execution in the SDF simulator are conservative because the selftimed execution of the SDF graph is monotonic. The use of the RTs of actors allows us to derive an upperbound on the token arrival times for soft real-time jobs given a specific input stimuli set for that job.

In Section 7 we will use Predicted Response Times (PRTs) of the actors to derive at compile time the resource budgets of soft real-time jobs. The PRT of an actor is the measured average response time of this actor on a processor given a specific input stimuli set for that actor. Given the PRT of an actor we will predict the resource budget for a soft real-time job.

5. MULTIPROCESSOR SYSTEM TEMPLATE

This section describes a network based multiprocessor system. This multiprocessor system is defined in such a way that a tight bound on the temporal behavior of jobs can be derived at compile time with dataflow analysis techniques. These analysis techniques are described in the previous section and are extended in Section 6.

Figure 4-6 shows the architecture template of this multiprocessor system. The processors in this template are, together with their local data memory, connected to the Network Interface (NI) of a packet switched Network on Chip

(NoC) (Rijpkema, Goossens, Rădulescu, Dielissen, Meerbergen, Wielage and Waterlander, 2003). The transfer of data between a local memory and a network interface is performed by a Communication Assist (CA). A processor together with its local instruction and data memory, communication assist, and network interfaces is grouped into a leaf. The leafs are connected to the routers of our network. Network links connect the routers in the desired network topology.

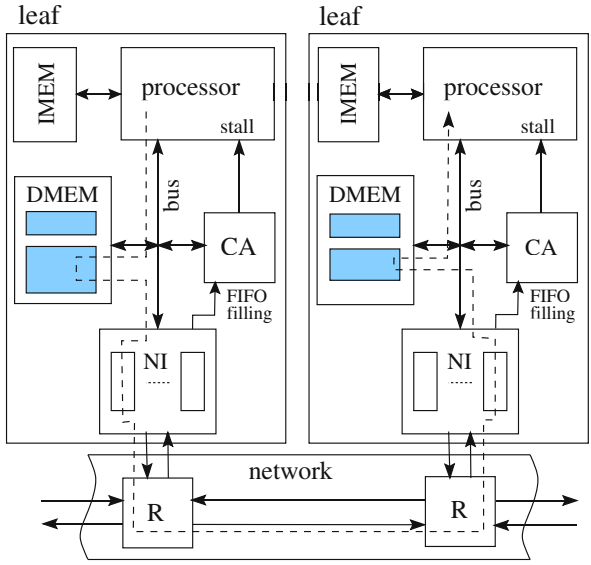


Figure 4-6. Multiprocessor template.

A processor in a leaf has a separate Instruction Memory (I-mem) and Data Memory (D-mem), such that instruction fetches and data load and store operations do not cause contention on the same memory port. An unbounded range of memory access time variations due to contention on a memory port is intolerable, as this would result in an unpredictable execution time of the instructions of the processor. This is also the reason why we consider in this paper only the case that the processors access only their local data memory. Given a 1 cycle access time of a local memory there is no reason to introduce caches.

Communication between actors on different processors takes place via a virtual point to point connection of the NoC. The result of the producing actor is written in a logical FIFO in the local memory of the processor. Such a logical FIFO can be implemented with the C-HEAP (Gangwal, Nieuwland and Lippens, 2001) communication protocol, without use of semaphores. The

communication assist polls at regular intervals whether there is data in this FIFO. As soon as the CA detects that there is data available, it copies the data into a FIFO of the NI. There is one private FIFO per connection in the NI. Subsequently the data is transported over the network to the NI of the receiving processor. As soon as the data arrives in this NI, it is copied by the CA into a logical FIFO in the memory of the processor that executes the consuming actor. The data is read from this FIFO after the consuming actor has detected that there is sufficient data in the FIFO. Flow control between the producing and consuming actor is achieved by making sure that data is not written into a FIFO before it is checked that there is space available in this FIFO.

Data is stored in the local memory of the processor before it is transferred across the network. This is done for a number of reasons. First of all, the bandwidth of a connection is set by configuring tables in the network for a longer period of time. The bandwidth reserved for a connection will typically be less than the peak data rate generated by the producing actor. Therefore a buffer is needed between the processor and the network to average out the peak data rate such that the bandwidth provided by the network is well utilized. Also the memory in the leaf which receives the data can typically not accommodate the peak bandwidth because another processor can access this memory at the same time. Another reason is that without such a buffer the execution time and the response time of the actors is dependent on the allocated bandwidth in the network. This dependency will complicate the analysis of the temporal behavior.

The size of the buffer in which data is stored before it is transferred across the network is significant, given the assumption that the actors produce large chunks of data at large intervals. On the other hand, the network will transfer very small chunks of data (3 words of 32 bits) at very small intervals ($\sim$2 ns). Given that large memories are inherently slow, it is desirable to split the large logical FIFO between the processor and the network, in a small ($\sim$32 word) dedicated FIFO per connection in the network interface, and a large logical FIFO in the local memory of the processor. The task of the CA is to copy the data between FIFOs in the NI and FIFOs in local memory of the processor.

The CA is also responsible for the arbitration of the data memory bus. The applied arbitration scheme is such that a low worst-case latency of memory store and load operations is obtained and that a minimal throughput and maximal latency per connection is guaranteed. A more detailed description can be found in (Bekooij, Moreira, Poplavko, Mesman, Pastrnak and van Meerbergen, 2004).

In the proposed architecture, the communication between actors that run on different processors has a guaranteed minimal throughput and a maximal latency. Given these characteristics, the communication can be modeled as if it takes place through completely independent virtual point-to-point connections.

These connections can be modeled together with the actors of a job in one SDF graph (Poplavko et al., 2003). Given this SDF graph, the guaranteed minimal throughput of a hard real-time job can be determined.

6. RESOURCE ARBITRATION

In this section, we show that the resource conflicts that are resolved at run-time by TDMA arbiters, can be taken into account in an SDF model. Conservative token arrival times are observed during self-timed execution of this SDF model in a simulator. The same SDF model is analyzed in Section 7 to obtain the sensitivity for fluctuations in the response times of actors on the temporal behavior of a job.

Resource conflicts can occur if multiple actors execute on one processor. These resource conflicts can be resolved at compile time or at run time. The resource conflicts can be resolved at compile time by computing offline a valid schedule for the SDF graph under consideration. In the static order scheduling approach (Sriram and Bhattacharyya, 2000), the execution order of the actors in this offline computed schedule is enforced at run time. If the static order scheduling approach is applied, then a decrease in response time of actors can only result in an earlier arrival of tokens and an increase in throughput of the system. The reason is that there is a one to one correspondence between actors in the system and the actors in the SDF model and that it is known that the self-timed execution of the SDF model is monotonic (see Section 4).

An important disadvantage of the static order scheduling approach is that it cannot be applied if the execution of actors is conditional, as is the case in the H263 video decoder example in Section 9. The execution of actors is conditional if a value of a token determines whether an actor will be executed or not. In a static order schedule it can occur that if, for example, actor A is not executed, then another actor B will wait forever for a token produced by actor A. Other actors on the same processor as actor B will not be executed as long as actor B waits for the token because the execution order of actors on the same processor is predefined and fixed.

Resource conflicts can also be resolved at run time by an arbiter (local scheduler). In the case that arbitration is performed at run time, the arrival of tokens determines whether an actor will be executed or not, and what the execution order of the actors on a processor will be. In the case that, for example, TDMA arbitration is applied, then the effects of the TDMA arbitration on the arrival time of the tokens can be taken into account in the response times of the actors in the SDF model. A proof that TDMA arbitration can be modeled implicitly in the response time of an actor is presented in the next paragraphs. This proof demonstrates with mathematical induction that tokens will not arrive later in the implementation than during selftimed execution of an

HSDF model. It is sufficient to prove for one actor executed during a TDMA time slice on a processor that the actor will not produce tokens later in the implementation than in the HSDF model because the selftimed execution of an HSDF model is monotonic.

In this proof, an abstract representation of a processor is used which executes actor A1 during interval T_1 in a period T. This representation is shown in Figure 4-9. Actor A1 starts its execution during interval T_1, as soon as the previous execution of actor A1 has finished and an input token has arrived. If actor A1 did not finish its execution at the end of the interval T_1 then this actor will be preempted and it will continue its execution in the next period. The additional time due to context switches can be included in the (worst-case) response time of an actor, because the maximum number of context switches that can happen during the execution of an actor is known at compile time. The time $p(j)$ denotes the execution time of the j-th execution of actor A1 when it executes on the processor without being preempted.

Two cases should be distinguished to determine the response time of an actor. An actor can start at the begin of the interval T_1 or during the interval T_1. If the actor starts at the begin of an interval and $p(j) = 2.5\ T_1$ then this actor will be preempted twice, as is shown in Figure 4-7. Given that the actor is preempted twice then the actor will finish its execution $p(j) + 2(T - T_1)$ after it is started. In other words the Interruption time (I1) of the actor is in this case according to (4-4).

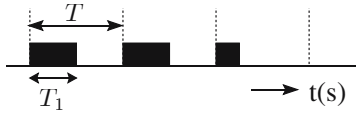


Figure 4-7. Stretch of the response time of an actor due to preemption in the case that the actor starts at the begin of interval T_1.

$$I1(j) = (T - T_1)(\left\lceil \frac{p(j)}{T_1} \right\rceil - 1) \tag{4-4}$$

On the other hand, if the execution of an actor starts during the time slice T_1, as is shown in Figure 4-8 then this actor will be preempted 3 times. Given that the actor is preempted 3 times, then the actor will finish its execution $p(j) + 3(T - T_1)$ after it is started. In other words, the interruption time (I2) of the actor is in this case according to (4-5).

$$I2(j) = (T - T_1)(\left\lceil \frac{p(j)}{T_1} \right\rceil) \tag{4-5}$$

Figure 4-8. Stretch of the response time of an actor due to preemption in the case that the actor can start at any point in time during interval T_1.

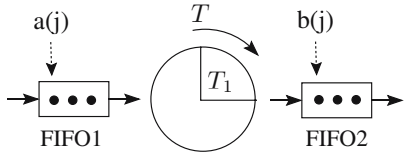


Figure 4-9. Abstract representation of the time wheel of a TDMA arbiter. The time wheel rotates every period T. Actor A1 can execute during slot 1 with duriation T_1.

The arrival time of the j-th token in FIFO1 and FIFO2 is denoted in Figure 4-9 by $a(j)$ and $b(j)$ respectively. The moment in time that the j-th execution of actor A1 finishes, is denoted by $f(j)$. During the j-th execution of actor A1, the j-th token is consumed from FIFO1 and the j-th token is produced in FIFO2. Therefore is $a(j) \leq f(j)$ and $b(j) \leq f(j)$. It will be proven for the HSDF model in Figure 4-10 that if (4-6) holds that then also (4-7) holds, where $\hat{a}(j)$ and $\hat{b}(j)$ denote the arrival time of tokens in the SDF model. The position of the initial token at time t=0 is as shown in Figure 4-10. However the position of the time-wheel in the implementation at time t=0 is unknown. Given this, it will be proven with mathematical induction for $j \geq 0$ that if (4-6) holds then also (4-7) holds.

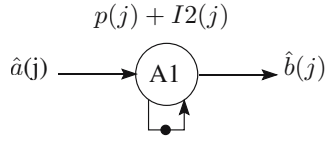


Figure 4-10. SDF model of an actor executed during a time slice on a processor.

$$a(j) \leq \hat{a}(j) \tag{4-6}$$

$$b(j) \leq \hat{b}(j) \tag{4-7}$$

Given that the position of the time-wheel of the implementation is unknown and that initially actor A1 does not execute then $f(0)$ is:

$$f(0) \leq a(0) + p(0) + \max(I1(0), I2(0)) \leq a(0) + p(0) + I2(0) \quad \text{(4-8)}$$

For the arrival time of the first output token in the HSDF model it holds that:

$$\hat{b}(0) = \hat{a}(0) + p(0) + I2 \quad \text{(4-9)}$$

From (4-6), (4-9) and (4-8) it follows that:

$$f(0) \leq \hat{b}(0) \quad \text{(4-10)}$$

Now we want to establish our inductive step by showing how the truth of our induction hypothesis in (4-11) forces us to accept that $f(j+1) \leq \hat{b}(j+1)$.

$$f(j) \leq \hat{b}(j) \quad \text{(4-11)}$$

For the implementation and $j \geq 0$ the following equations hold in which the intermediate variables tx and ty are defined:

$$tx = a(j+1) + p(j+1) + \max(T - T_1 + I1(j+1), I2(j+1)) \quad \text{(4-12)}$$

$$ty = f(j) + p(j+1) + \max(T - T_1 + I1(j+1), I2(j+1)) \quad \text{(4-13)}$$

$$f(j+1) \leq \max(tx, ty) \quad \text{(4-14)}$$

Equation 4-14 can be rewritten as:

$$f(j+1) \leq \max(f(j), a(j+1)) + p(j+1) + I2(j+1) \quad \text{(4-15)}$$

Equation 4-14 holds because: if $a(j+1) > f(j)$ then token $j+1$ has arrived after the j-th execution of actor A1 has finished (see Figure 4-11). After arrival

of token $j+1$ it will take maximally $p(j+1) + \max(T - T_1 + I1(j+1), I2(j+1))$ before the $(j+1)$-th execution of actor A1 finishes. It takes $p(j+1) + T - T_1 + I1(j+1)$ before $(j+1)$-th execution of actor A1 finishes, if the position of the time wheel is such that token $j+1$ arrives during interval $T - T1$, otherwise it takes $p(j+1) + I2(j+1)$.

If $a(j+1) \leq f(j)$ then token $j+1$ has arrived in FIFO1 before the j-th execution of actor A1 has finished (see Figure 4-12). After the j-th execution of actor A1 has finished it takes maximally $p(j+1) + \max(T - T_1 + I1(j+1), I2(j+1))$ before the $(j+1)$-th execution of actor A1 has finished. It takes $p(j+1) + T - T_1 + I1(j+1)$ before the $(j+1)$-th execution of actor A1 finishes, if the position of the time wheel is such that the j-th execution of actor A1 finishes at the end of interval T_1 otherwise it takes $p(j+1) + I2(j+1)$.

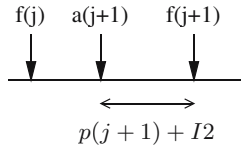


Figure 4-11. Arrival of token j+1 in FIFO1 after the j-th execution of actor A1 has finished.

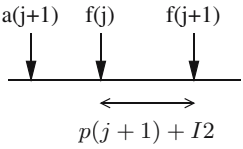


Figure 4-12. Arrival of token j+1 in FIFO1 before the j-th execution of actor A1 has finished.

For the SDF model and $j \geq 0$ the following equations hold in which the intermediate variables tp and tq are defined

$$\hat{b}(j+1) = \max(tp, tq) \tag{4-16}$$

with

$$tp = \hat{a}(j+1) + p(j+1) + I2(j+1) \tag{4-17}$$

$$tq = \hat{b}(j) + p(j+1) + I2(j+1) \tag{4-18}$$

It follows from (4-12), (4-17) and (4-6) that $tx \leq tp$. From (4-13), (4-18), and our induction hypothesis in (4-11) it follows that $ty \leq tq$. This results in the conclusion that $f(j) \leq \hat{b}(j)$ for $j \geq 0$ because:

$$tx \leq tp \wedge ty \leq tq \Rightarrow max(tx, ty) \leq max(tp, tq) \tag{4-19}$$

Given that $b(j) \leq f(j)$ we arrive at the conclusion that (4-7) holds for $j \geq 0$. □

In the proof an HSDF actor was considered with only one input and one output and FIFO buffers with an infinite capacity were assumed. However the proof also holds for an SDF actor with multiple inputs and outputs and buffers with a finite capacity. The reason is that the event $a(j)$ which denotes the arrival of a token in FIFO1 in Figure 4-9 is equivalent to the event which denotes that sufficient tokens are available on each input of an SDF actor. The proof also holds for SDF actors with multiple outputs because all output tokens are produced before the SDF actor finishes its execution. The availability of space in a finite FIFO buffer can be modeled as the presence of a space token on an input of the SDF actor.

The use of TDMA arbitration can be taken into account in the SDF model of hard real-time jobs by setting the WCRT of the actor Ax according to (4-20), in which P denotes the WCET of actor Ax if it would be executed on the processor without being preempted.

$$\text{WCRT}_{Ax}(j) = P + (T - T_1) \left\lceil \frac{P}{T_1} \right\rceil \tag{4-20}$$

Given these worst-case response times of actors, the worst-case arrival times of the tokens in the system can be derived from a self-timed execution of the SDF model. Also the minimal throughput of the system that will obtained equals 1/MCM of this SDF model. This SDF model can also be used with response times instead of worst-case response times. An upperbound on the RT of the j-th execution of actor Ax in the SDF model is equal to:

$$\text{RT}_{Ax}(j) = p(j) + (T - T_1) \left\lceil \frac{p(j)}{T_1} \right\rceil \tag{4-21}$$

Conservative token arrival times are then observed during self-timed execution of the SDF model due to monotonicity of the system. Conservative token arrival times are observed because an earlier arrival of a token can only result in an earlier start of an actor in the SDF model and an earlier production of a result. Therefore conservative arrival times are observed if the i-th response time of actor A_x in the SDF model is not shorter then the i-th response time of

actor A_x in the implementation. This is the case if response times according to (4-21) are used in the SDF model. An important advantage of the use of an SDF model instead of cycle true simulation model of the system is that the arrival time of tokens during self-timed execution is conservative while this is not the case for a cycle true simulation model. The reason is that the initial position of the time wheels in the system at time t=0 is not known. Another advantage is that execution of the SDF model will be much faster than simulation of a cycle true model because an SDF model is a more abstract model.

7. SENSITIVITY ANALYSIS & REDUCTION

This section describes dataflow analysis techniques that are used to determine the FIFOs of which the capacity should be increased, in order to reduce the sensitivity of a soft real-time job, for fluctuations in the response times of the actors. A lower sensitivity of a job will reduce the deadline miss rate which enhances the quality of experience of the user.

For soft real-time jobs a predicted MCM can be calculated with (4-2) given the resource budgets of a job and by using the predicted response times of the actors instead of the WCRT of the actors. Here it is assumed that the PRT of an actor is equal to the average response time of this actor on a processor with TDMA arbitration and representative input data.

It is obvious that this predicted MCM is not smaller than the actual MCM if the response times of the actors is smaller than the PRT of the actors. The predicted MCM is also not smaller than the actual MCM if the cycle mean of a cycle in the SDF graph, to which the actors belong that have an RT larger than their PRT, does not exceed the predicted MCM. The Cycle Mean (CM) is defined in (4-3). In other words the temporal behavior of a job is more sensitive for deviations in the response times of actors which belong to cycles of which the CM is likely to be larger than the predicted MCM. By increasing the FIFOs capacity, the CM of these cycles can be decreased such that the job becomes less sensitive.

That the sensitivity of a job can be reduced by increasing the FIFO capacities can be seen as follows. Assume that the job is described by the SDF graph in Figure 4-13. The PRT of actor A1 in this job is chosen to be equal to the average response time measured over 3 successive executions of this actor. If the desired MCM is 2T then the FIFO capacity should be at least 2 tokens given the PRT of the actors. However, it is likely that the actual MCM is larger than the desired MCM because the RT of actor A1 can be larger then its PRT which results in a CM larger than the predicted MCM.

The actual MCM would not be larger than the desired MCM if a FIFO capacity of 6 instead of 2 tokens was applied. That this is the case can be intuitively seen as follows. Assume the an actor A1' in Figure 4-14 requires 3

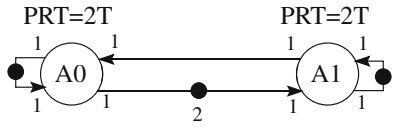


Figure 4-13. SDF with a predicted MCM of 2T.

tokens instead of 1 token on its input before it fires and that after firing it executes internally 3 times the same code segment. Actor A1' would have in this case a PRT equal to the maximum response time of 3 successive executions. In Figure 4-14 we assumed that out of the 3 successive executions of actor A1, 2 executions have a response time smaller than T and one a response time smaller than 4T. In this case is the PRT of actor A1' equal to 6T. Given the PRT there is a FIFO capacity needed of 6 tokens for a desired MCM of 2T. This MCM can be obtained with (4-2) after the SDF in Figure 4-14 is transformed in an HSDF with the algorithm described on page 40 in (Sriram and Bhattacharyya, 2000). The longest path in this HSDF contains 3 times actor A0 and once actor A1' and is 12 T long. Therefore, there must be 6 tokens on this path for an MCM of 2T. Given these 6 tokens an MCM of 2T will be obtained if actor A1 in the implementation fires as soon as there is one input token available. The reason is that starting of the actor with only 1 instead of 3 tokens can only result in an earlier production of tokens.

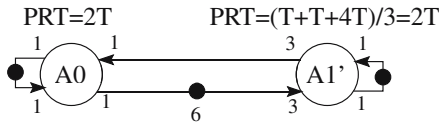


Figure 4-14. SDF with a predicted MCM of 2T.

8. PREDICTABLE DYNAMIC DATA FLOW

In this section a so-called Predictable Dynamic Data-Flow (PDDF) graph is introduced in which the conditional executions of actors can be expressed as well as a variable but bounded number of executions of actors can be expressed. An important property of a PDDF is that it can be analysed with the in Section 4 and Section 7 described analysis techniques.

An H263 video decoder is an example in which it depends on the values of the input data which actors will be executed and which not. In this decoder different actors are executed in the case that an I-frame or a P-frame is decoded. The conditional execution of actors cannot be made explicit in SDF

graphs but can be made explicit in Boolean Data-Flow (BDF) graphs (Buck, 1993). However, the use of a BDF graph is undesirable because the detection of deadlock is undecidable for an arbitrary BDF graph. By restricting, with construction rules, the type of BDF graphs that can be expressed so-called well-behaved dataflow graphs (Gao, Govindarajan and Panangaden, 1992) are obtained. These well-behaved dataflow graphs are per construction deadlock free. However, to derive a tight lower bound on the throughput of an application with MCM analysis it is also necessary that the actors that are conditionally executed do not share resources or are executed mutual exclusive. Mutual exclusive execution is typically desirable because sharing of resources reduces the resource requirements. Mutual exclusive execution can be guaranteed by extending the well-behaved dataflow graph with a so-called mode manager actor M, as is done in Figure 4-15. This mode manager actor provides N times a control token with the same boolean value for the switch and select actor and then waits till the select actor has been executed N times before it produces a control token with possibly a different boolean value. It is required that the select actor produces a token at the end of its execution. The construct in Figure 4-15 guarantees that there is no input token available for actor A0 and A1 at the same time and that therefore the execution of these actors is mutual exclusive. That N times the same control token is produced by actor M is made explicit in Figure 4-15 with the N[T/F] annotation. The name Predictable Dynamic Data-Flow (PDDF) graphs has been given to dataflow graphs in which the construct in Figure 4-15 is used to express conditional execution of actors.

The minimal throughput of a PDDF graph can be determined by calculating the MCM of the PDDF graph with (4-2) which is the same equation as is used for the calculation of the MCM of an SDF graph. The same equation can be used because the PDDF graph in Figure 4-15 has an equivalent worst-case temporal behavior as the SDF graph in Figure 4-16. This is the case because the PDDF graph in Figure 4-15 is per construction deadlock free. Also, the execution of the actors A0 and A1 is by construction mutually exclusive. Therefore, it can be assumed during MCM analysis that the actors A0 and A1 are both executed for each input token of the select actor but that each of these actors is executed on its own private processor. If actors A0 and A1 are executed for each input token then the switch and select actors should behave like ordinary actors which consume tokens from all inputs and produce tokens on all their outputs and have a zero WCRT. The value of the control token provided by the mode manager actor M to the switch (SW) actor is ignored because the data token must be duplicated by the switch actor to both outputs. The select (SE) actor should consume a token produced by actor A0 as well as A1 and copy one of these tokens to its output. Because the value of the control token is irrelevant for the worst-case temporal behavior there is no need to make explicit that the same control token is sent to the switch as well as the select actor.

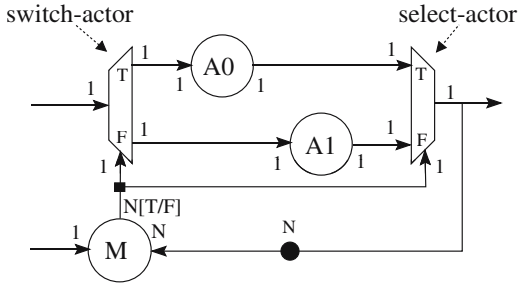


Figure 4-15. Predictable Dynamic DataFlow (PDDF) graph with mutual exclusive execution of actors in the True and False branch.

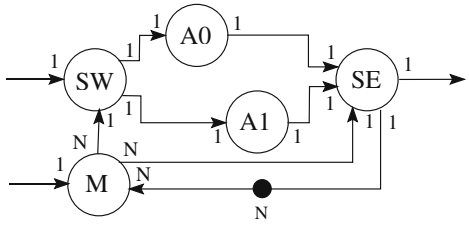


Figure 4-16. SDF graph with the same worst-case temporal behavior as the PDDF graph in Figure 4-15.

A PDDF construct in which actor A2 is executed p times is shown in Figure 4-17. This construct executes in bounded memory because actor A1 informs actor A3 about the number of tokens it must consume. Actor A1 informs actor A3 by sending one token with value p to A3. This is indicated in Figure 4-17 with the notation 1[p]. During calculation of the PDDF graph's MCM the maximum value of p must be used because a larger value p will result in more executions of actor A2 and a later start of actor A3.

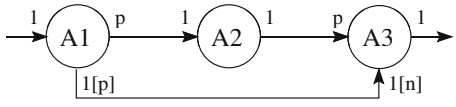


Figure 4-17. PDDF graph in which actor A2 is executed p times.

9. DATAFLOW MODEL OF AN H263 VIDEO DECODER

A dataflow model of an H263 video decoder is presented in this section which illustrates the use of the modeling techniques that were introduced in the previous sections. This H263 video decoder is a soft real-time job of which the values of the input data determine whether some of the actors will be executed or not. Also the number of tokens produced and consumed by the actors can be data dependent. Despite the dynamic behavior of this job, it remains possible to derive the minimal capacity of the FIFOs as well as conservative arrival times of tokens with a dataflow model.

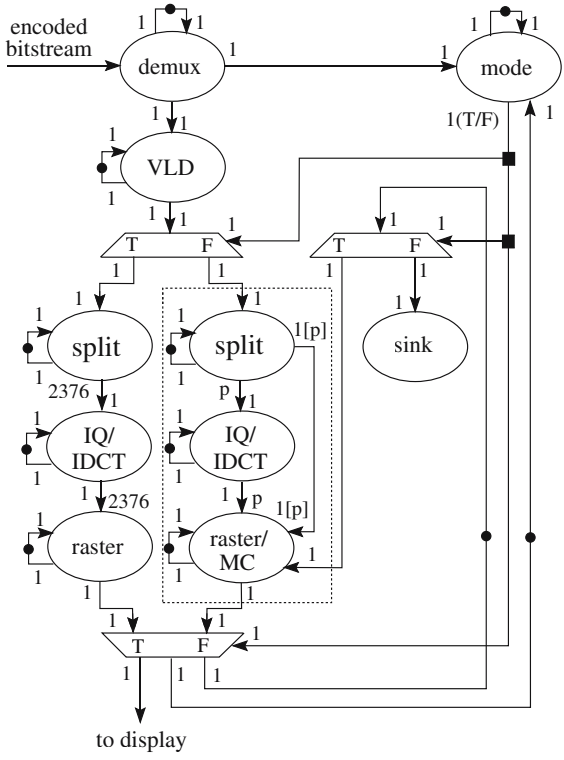


Figure 4-18. Predicatable dynamic dataflow model of an H263 video decoder.

The predictable dynamic dataflow model of an H263 decoder is shown in Figure 4-18. This decoder receives a bit stream which is split by the demultiplexer (demux) actor in a token for the mode manager (mode) actor and a token for variable length decoder (VLD) actor. The token for the mode actor

indicates whether the next frame to decode is an Intra (I) or a Predicted (P) frame. The token for the VLD actor contains one encoded frame.

The token produced by the VLD actor in case of an I-frame in CIF resolution (352×288 pixels) is split in 2376 tokens of which each token contains an encoded block. These 2376 tokens are processed by the combined Inverse Quantization (IQ) and Inverse Discrete Cosine Transform (IDCT) actor and a rasterization (raster) actor. The result of the rasterization actor is one decoded frame which can be displayed. That a complete frame has been decoded and that the next frame can be decoded is indicated by sending a token to the mode manager.

The token produced by the VLD actor in case of a P-frame is split in p tokens of which each token contains an encoded macro block. The split actor also notifies the rasterization/Motion Compensation (raster/MC) actor that it should consume p tokens. The tokens produced by the split actor are processed by an IQ/IDCT actor and the raster/MC actor. The raster/MC actor receives also a token which contains the previous frame.

The production of a variable number of tokens by the split actor is allowed in this dataflow graph because a maximum number of tokens ($p \leq 2376$) is known at compile time. Given this maximum number of tokens, the minimum FIFO capacity between the split actor and the IQ/IDCT actor can be derived, as well as the minimum FIFO capacity between the IQ/IDCT actor and the raster/MC actor. Another important property is that conceptually one actor could be introduced, which is indicated in Figure 4-18 by the dashed box, in which the production and consumption of a variable number of tokens is hidden.

Conservative arrival times of tokens can be observed during simulation of the dataflow model of the H263 decoder, given that the response times of the actors are according to (4-21).

10. CONCLUSION

Embedded multiprocessor systems in consumer products execute a combination of soft real-time and hard real-time jobs that process data streams. Dataflow models in which computation, communication and arbitration is modeled can be used to derive the minimal throughput of the hard real-time jobs, using MCM analysis. For soft real-time jobs, simulation of these dataflow models are used to test statistical assertions given representative input streams. The simulation effort is reduced and the confidence of the simulation results is improved by making only use of predictable arbitration policies (e.g. TDMA) in the proposed network based multiprocessor system. The simulation effort is reduced because the use of predictable arbitration policies eliminates the interference between jobs, and guarantees that conservative arrival times of to-

kens are observed during simulation of the dataflow model of a job. Dataflow analysis techniques are used to estimate the resource budgets of soft real-time jobs. With these analysis techniques the buffers are derived which should be increased to reduce the sensitivity for fluctuations in the response time of actors on the temporal behavior of a job. A predictable dynamic dataflow model of an H263 video decoder job is presented in which conditional construct determine which actors are executed during the decoding of I-, and P-frames. The temporal behavior of such a job can be analyzed with the presented analysis and simulation techniques.

References

Bacelli, F., Cohen, G., Olsder, G. and Quadrat, J.-P., 1992, *Synchronization and Linearity*, John Wiley & Sons, Inc.

Bekooij, M., Moreira, O., Poplavko, P., Mesman, B., Pastrnak, M. and van Meerbergen, J., 2004, Predictable embedded multiprocessor system design, *Proc. Intl Wor kshop on Software and Compilers for Embedded Systems (SCOPES)*, LNCS 3199, Springer.

Buck, J., 1993, *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*, PhD thesis, Univ. of California, Berkeley.

Cochet-Terrasson, J., Cohen, G., Gaubert, S., McGettrick, M. and Quadrat, J.-P., 1998, Numerical computation of spectral elements in max-plus algebra, *Proc. IFAC Conf. on Syst. Structure and Control*.

Gangwal, O., Nieuwland, A. and Lippens, P., 2001, A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems, *Int'l Symposium on System Synthesis (ISSS)*, ACM, pp. 1–6.

Gao, G., Govindarajan, R. and Panangaden, P., 1992, Well-behaved dataflow programs for DSP computation, *International Conference of Acoustics, Speech and Signal processing*.

Hee, K. v., 1994, *Information System Engineering*, Cambridge University Press.

Kahn, G., 1974, The semantics of a simple language for parallel programming, *Proceedings IFIP Congress*, pp. 471–475.

Kock, E., Essink, G., Smits, W., Wolf, P. v. d., Brunel, J.-Y., Kruijtzer, W., Lieverse, P. and Vissers, K., 2000, Yapi: Application modeling for signal processing systems., *In Proceedings of 37th Design Automation Conference (DAC00)*, Los Angeles, pp. 402–405.

Kopetz, 1997, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer.

Lawler, E., 1976, *Combinatorial optimization: Networks and Matroids*, Holt, Reinhart, and Winston, New York, NY, USA.

Lee, E. and Messerschmitt, D., 1987, Synchronous data flow, *Proceedings of the IEEE*.

Li, Y.-T. S. and Malik, S., 1999, *Performance analysis of real-time embedded software*, ISBN 0-7923-8382-6, Kluwer academic publishers.

Petri, C., 1962, *Kommunikation mit Automaten*, PhD thesis, Institut für Instrumentelle Mathematik, Bonn, Germany.

Pop, T., Eles, P. and Peng, Z., 2002, Holistic scheduling of mixed time/event-triggered distributed embedded systems, *Proc. Int'l Symposium on Hardware/Software Codesign (CODES)*, pp. 187–192.

Poplavko, P., Basten, T., Bekooij, M., Meerbergen, J. v. and Mesman, B., 2003, Task-level timing models for guaranteed performance in multiprocessor networks-on-chip, *Proc. Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pp. 63–72.

Rajkumar, R., Juwa, K., Moleno, A. and Oikawa, S., 1998, Resource kernels: A resource-centric approach to real-time and multimedia system, *SPIE/ACM Conference on Multimedia Computing and Networking*.

Richter, K., Jersak, M. and Ernst, R., 2003, A formal approach to MpSoC performance verification, *IEEE computer* **36**(4), 60–67.

Rijpkema, E., Goossens, K., Rădulescu, A., Dielissen, J., Meerbergen, J. v., Wielage, P. and Waterlander, E., 2003, Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip, *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 350–355.

Sriram, S. and Bhattacharyya, S., 2000, *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Dekker, Inc.

Zhang, H., 1995, Service disciplines for guaranteed performance services in packet-switching networks, *Proceedings of the IEEE* **83**(10), 1374–96.

Chapter 5

RESOURCE RESERVATIONS IN SHARED-MEMORY MULTIPROCESSOR SOCS

Clara Otero Pérez, Martijn Rutten[1], Liesbeth Steffens, Jos van Eijndhoven, and Paul Stravers
Philips Research Laboratories, Eindhoven, The Netherlands; [1] Philips Semiconductors, Eindhoven, The Netherlands

Abstract: Consumer electronics vendors increasingly deploy shared-memory multiprocessor Systems on Chip (SoC), such as Philips Nexperia, to balance flexibility (late changes, software download, reuse) and cost (silicon area, power consumption) requirements. With the convergence of storage, digital television, and connectivity, these media-processing systems must support numerous operational modes. Within a mode, the system concurrently processes many streams, each imposing a potentially dynamic workload on the scarce system resources. The dynamic sharing of scarce resources is known to jeopardize robustness and predictability. Resource reservation is an accepted approach to tackle this problem. This chapter applies the resource reservation paradigm to interrelated SoC resources: processor cycles, cache space, and memory access cycles. The presented *virtual platform* approach aims to integrate the reservation mechanisms of each shared SoC resource as the first step towards robust, yet flexible and cost-effective consumer products.

Key words: Virtual platform, multiprocessor system, shared resources, shared memory

1. INTRODUCTION

The convergence of consumer applications in the TV, PC, and storage domains introduces new combinations of features and applications that execute in parallel. In addition, consumer multimedia devices are becoming increasingly flexible. Flexibility enables accommodating late changes in standards or product scope during system design, and allows in the field

P. van der Stok (ed.),
Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, 109-137.

upgrades. To address the flexibility and concurrency requirements, consumer electronics vendors increasingly deploy heterogeneous multiprocessors systems.

The high production volume of consumer products sets severe requirements on the product cost, leading to resource-constrained devices. To achieve a cost effective solution, expensive resources, such as memory and processor time, are shared among concurrent applications.

A typical multimedia application consists of independently developed subsystems with strong internal cohesion. At the subsystem borders, the real-time requirements are decoupled from the other subsystems. However, resource sharing induces temporal interference among otherwise temporal independent subsystems given the highly dynamic workload of the targeted media applications, such as audio/video coding, image improvement, and content analysis. *Figure 5-1.* depicts the load fluctuations (in time) of two independent subsystems sharing a resource. At a given point in time, both subsystems require more resources than the total available and one (or both) of the subsystems will suffer.

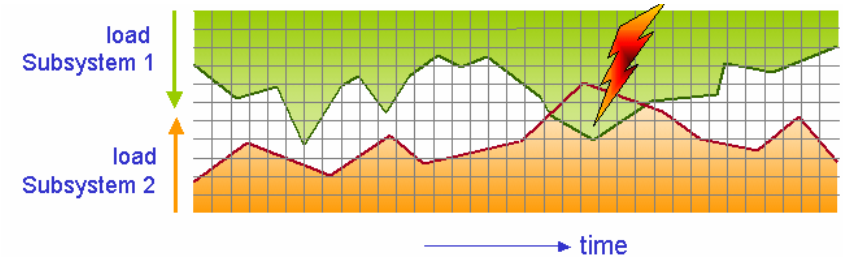


Figure 5-1. Load fluctuations of independent subsystems.

The concurrent execution of dynamic applications on shared resources is a potential source of interference, which leads to unpredictability and jeopardizes overall system robustness. We aim to bound interference by isolating and protecting independent subsystems from each other, while preserving typical qualities of multimedia devices, such as robustness and cost effectiveness.

Resource reservation is a well-known technique in operating system research to improve robustness and predictability. It is based on four components, admission control, scheduling, accounting, and enforcement. When properly combined, they provide guaranteed resource reservations. Resource reservations consist at least of two basic parameters: share and granularity. For example, a share of 5 milliseconds and a time granularity of 20 milliseconds determine a processing time reservation. Different resources

have various types and degrees of share-ability. For example, a programmable processor is shareable in time and fully preemptable. An MPEG-2 (video coding) hardware accelerator may decode one high-definition stream or decode two standard-definition streams in time-shared fashion. Memory is shared in space.

To bound interference at system level, the multimedia device must provide the subsystems with a reservation mechanism for each resource. By configuring the resource reservations, we create an execution platform that is tailored to the resource needs of the subsystem. We term this a *virtual platform*. A virtual platform provides guaranteed resource availability, while restricting resource usage to a configured maximum.

2. RELATED WORK

Multiprocessor systems on chip (SoC) are rapidly entering the high-volume electronics market. Example SoC platforms are Philips Nexperia (de Oliveira & van Antwerpen 2003), Texas Instruments OMAP(Cumming 2003), and STMicroelectronics StepNP (Paulin, Pilkington, & Bensoudane 2002). A mayor challenge for these multiprocessor systems is to effectively use the available resources and maintain a high degree of robustness. Currently, these systems do not explicitly address interference between software modules that compete for shared system resources. The robustness problems caused by interference are typically evaded by a high degree of over provisioning.

Various research efforts address interference for specific resources through processor resource reservations (Lipari & Bini 2003), (Eide et al. 2004), (Baruah & Lipari 2004), interconnection guarantees, Chapter 2 of this book (Goossens & González Pestana 2004), and cache partitioning (Liedtke, Haertig, & Hohmuth 1997),(Molnos et al. 2005). For instance, Ravi (Ravi 2004) presents various mechanisms for cache management based on priority assignment and enforcement. Recent research aims for integrated approaches that considers combination of resources such as processor and network reservations(Rajkumar et al. 2001), (Nolte & Kwei-Jay 2002). We take one step further and develop an integrated approach to bound interference for *all* shared resources in upcoming multiprocessor systems. Our multi-resource reservation is the base for defining an embedded *virtual* platform.

Numerous systems have been designed which use virtualization to subdivide the ample resources of a modern computer since IBM introduced the 360 model 67, in 1967. In a traditional virtual machine (VM), the virtual hardware exposed is functionally identical to the underlying machine (Seawright & MacKinnon 1979). However, full virtualization is not always

desired. Some of the disadvantages lead to a performance penalty that current high volume electronics vendors are not willing to pay. One of the goals of recent research on virtualization is to overcome these disadvantages. Rather than attempting to emulate some existing hardware device, the Xen VM research of Barham et al. (Barham et al. 2003) exposes specially designed block (device and network) interface abstractions to host operating systems, in what they call *paravirtualization*. Barham et al. assume full resource availability. It is not clear whether or how their approach provides guarantees and performs admission control. The severe cost requirements in the consumer electronics domain oblige us to provide resource guarantees for resource-constrained systems.

The remainder of this paper is organized as follows. Section 3 describes a embedded system consisting of a multiprocessor SoC in which concurrent media applications execute. The interference problem is explored in Section 4 and the concept of virtual platform as a solution to bound interference is introduced in Section 5. Section 6 presents the different reservation mechanism for the three main SoC resources (processor cycles, cache space and memory access cycles) used to implement the virtual platform. Finally, the conclusion is drawn in Section 7.

3. MULTIPROCESSOR SYSTEM

Multiprocessor SoCs are deployed to cope with the market demand for high performance, flexibility, and low cost. Progressive IC technology steps reduce the impact of programmable hardware on the total silicon area and power budget. This permits SoC designers to shift more and more functionality from dedicated hardware accelerators to software, in order to increase flexibility and reduce hardware development cost. However, for at least the coming decade, these multiprocessor SoCs still combine flexibility—in the form of one or more programmable central processing units (CPU) and digital signal processors (DSP)—with the performance density of application-specific hardware accelerators. *Figure 5-2* depicts such a heterogeneous SoC architecture as presented in Chapter 3 of this book (van Eijndhoven et al. 2005) and (Stravers & Hoogerbrugge 2001). In providing a virtual platform for upcoming SoCs, we have to cope with the interaction between processing in hardware and software.

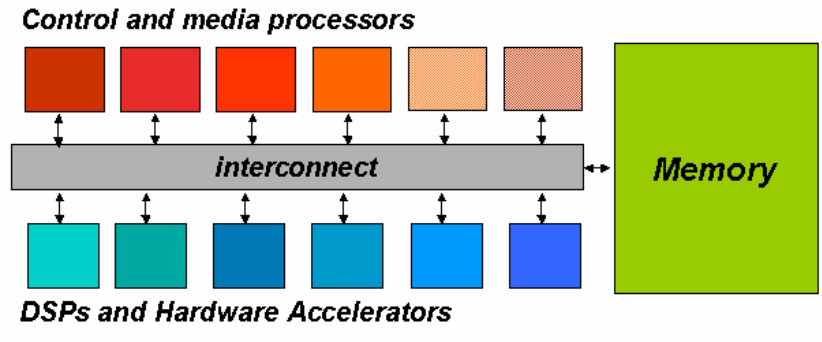


Figure 5-2. Heterogeneous SoC architecture with CPUs, DSPs, and accelerators communicating through shared memory.

With progressive technology steps, processing power and memory sizes increase, keeping the pace with the memory and processing capacity requirements imposed by media applications. In contrast, memory bandwidth scales slowly and memory latency remains almost the same. Thus, memory bandwidth and latency are becoming the dominant system bottleneck.

Figure 5-3 details the data path of a multiprocessor such as in *Figure 5-2*, in which a number of DSPs, CPUs, and accelerators communicate through shared memory. The architecture applies a two-level cache hierarchy to reduce memory bandwidth and latency requirements. The cache hierarchy is inclusive: a memory block can only be in a L1 cache if it also appears in the L2 cache. When a processing unit produces new data and stores it in its L1 cache, the L2 copy of that memory block becomes stale; in such cases a cache coherence protocol ensures that any consumer of the data always receives the updated L1 copy of the data. Furthermore, such a coherent communication network allows direct L1-to-L1 cache transfers, e.g. when a consumer task on processing unit A reads data from a producer task on processing unit B. At any moment in time, a modified data item resides only in one L1 cache. This property is intended to facilitate the partitioning of applications, consisting of multiple producer/consumer tasks, over multiple processors.

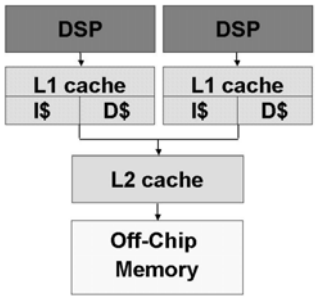


Figure 5-3. Data path for the memory hierarchy.

The applications we are dealing with are media applications (mainly audio and video). *Figure 5-4* depicts an example of a media application (Otero Pérez et al. 2003). Such applications are also known as streaming applications, because they process streams of data. A stream is a sequence of data objects of a particular type (audio samples, video pictures, video lines, or even pixels). For example, a video stream is a sequence of pictures, with a given picture rate: the number of pictures to be displayed per second. A stream is typically produced by one streaming task and consumed by some other concurrent asynchronous streaming task. The part of the stream that has been produced but not yet consumed, is temporarily stored in a buffer, or is being transferred, from producer to buffer, or from buffer to consumer.

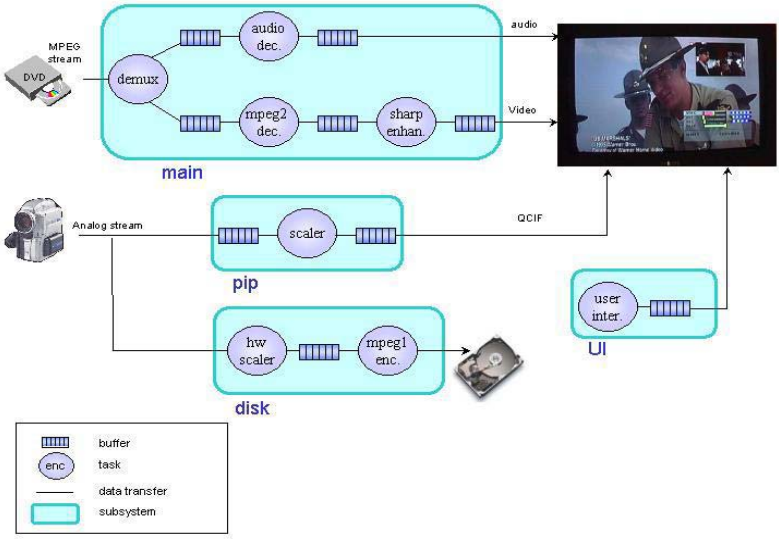


Figure 5-4. Media application example.

Our execution model for streaming applications consists of a connected graph in which the nodes represent either a task (an independent, asynchronous, active component that uses processing and memory resources) or a buffer (a passive component that uses memory resources). The interconnections represent the data transfer (memory access). The execution model is hierarchical. At higher levels of abstraction, a connected graph can again be viewed as a subsystem in a connected graph. *Figure 5-4* depicts four such subsystems: main, pip, disk, and user interface (UI). The subsystems are denoted with the rounded rectangles.

4. INTERFERENCE AMONG SUBSYSTEMS

The pressure on time-to-market, the emergence of multi-site development, and the ever-increasing size of software stacks are just some of the factors that enforce a radical change in the development of modern (multimedia) applications. In the past software systems were almost completely written from scratch as fully self-contained systems, developed under one roof. These days, systems are increasingly composed of independently developed subsystems, originating from different locations and in many cases from different companies. These subsystems are not designed as a specific part of a whole, but are intended to be deployed in many different systems, and serve different ranges of products.

Ideally, each subsystem is evaluated and tested in isolation for a specific system. The job of the system integrator is to mix and match the subsystems to compose the final system. Unfortunately, current subsystems are not compositional. The ad-hoc and implicit way in which the scarce SoC resources are managed, and the unbounded interference caused by resource sharing, introduces temporal interdependencies among these initially independent subsystems. If not properly managed, these interdependencies lead to unpredictable behavior for the integrated system.

Media-processing SoCs rely on priority scheduling in embedded real-time operating systems, such as VxWorks and pSOS, to manage real-time requirements. The current setting of priorities is an example of ad-hoc management. Traditionally, priorities were used to manage resource utilization in closed, real-time systems, where task activations and execution time are deterministic. Under these conditions, well-known priority assignment methods, such as rate monotonic assignment (Liu & Layland 1973), work fine. However, media-processing tasks tend to violate many of these assumptions, e.g., by generating idle time, dynamically fluctuating workloads, jitter, etc. The subsystem designers have to rely on trial and error to obtain a working system. Moreover, at integration time, when all tasks in all subsystems come together, the integrator has to start again from scratch. The priority assignment of the subsystem tasks cannot be reused in the integrated system, and the system integrator is faced with the difficult task of evaluating different priority settings, while other factors such as importance of the task or response time requirements influence the priority assignment.

A second example is the priority assigned to the various processors for bus access. The processor's bus priority is fixed and unrelated to the tasks executed by the processors. Oftentimes, the priority setting is based on a complex relation among the various tasks that might execute on that processor.

The use of a cache introduces a third example of unpredictability, due to the difficulty in predicting when certain data is available in the cache or still has to be fetched from off-chip memory, causing the processor to stall. Interrupts in combination with caches are a further cause of unbounded interference. A typical system relies on interrupts to activate hardware accelerator, to handle exceptions, to wake up software tasks, etc. An interrupt causes a context switch, evicting the running task from the processor and invalidating the task data present in the cache. When the running task resumes execution, potentially all its data has to be fetched again from memory. Therefore, it is very difficult to determine an upper bound for the performance impact caused by interrupts in cache-based systems.

We conclude from the previous paragraphs that resource management based on bounding interference constitutes the foundation for compositional system design. We propose an integrated approach to resource management based on guaranteed resource reservation for all shared SoC resources, tailored to the needs of the resource consumers (subsystems). The concept of a virtual platform—as outlined in the next section—summarizes our approach towards such integration.

5. VIRTUAL PLATFORM

A virtual platform provides guaranteed resource availability, while restricting resource usage to a configured maximum. Like the real platform, a virtual platform provides a wide variety of resources: programmable processors, function specific hardware, memory space, memory access bandwidth, and interconnect bandwidth. In a SoC, a virtual platform can be implemented in various ways. For example, a set of tasks can execute concurrently on multiple slow processors or sequentially on a fast processor.

The implementation of a virtual platform is based on resource reservation mechanisms that provide temporal and spatial isolation among subsystems. The resource manager is responsible for providing virtual platforms by ensuring that sufficient resources are reserved. For that, a resource reservation mechanism, for each main SoC resource, guarantees the availability of resources. As depicted in *Figure 5-5*, the resource manager translates subsystem requirements and sets the parameters for the virtual platform. This requires appropriate knowledge of the demands of the individual subsystems in terms of the specific platform resources. Characterizing performance and behavior of the subsystems is a subject of research, fundamental to the realization of a virtual platform.

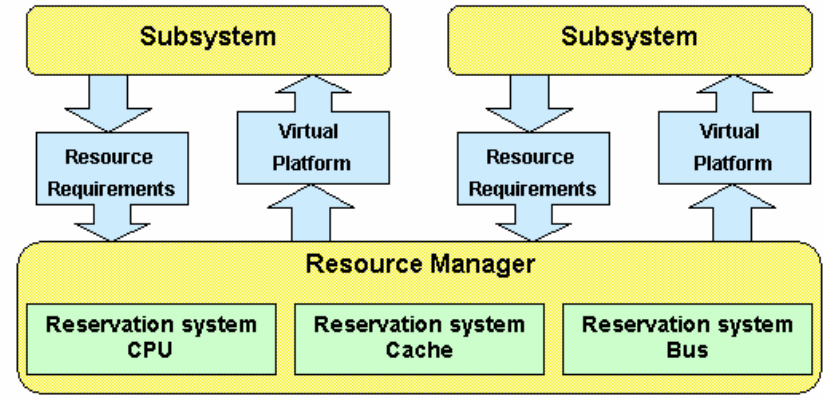


Figure 5-5. Subsystems and virtual platforms.

Furthermore, to actually deploy the virtual platform concept, the following three issues must be resolved. Firstly, to provide a virtual platform, the resource manager has to coordinate the resource reservations for each resource. For that, the interdependencies among resources must be modeled and analyzed. The effective CPU speed depends on the reservations made in the memory architecture, such as cache and bus bandwidth. For example, a memory controller that schedules processor requests to memory guarantees a given average latency for a given processor. This latency is used to calculate the execution time of a subsystem on this processor and determines its processing budget.

Secondly, the reservation of a resource in the resource hierarchy may not be based on the virtual platform using the resource, but on the actual physical components using this resource. An example is memory bandwidth. This bandwidth is allocated to the physical processors accessing the memory independently from which virtual platform this processor is allocated to. As a virtual platform is, in general, implemented by several physical processors, a complex hierarchical set of interdependencies is created. These interdependencies are very difficult to understand and to analyze. Note that this complexity is not introduced by the virtual platform concept itself, but is inherently present in current SoC architectures and must be solved independently of the virtual platform.

Finally, given the dynamic behavior of the software, absolute guarantees are only possible when the reservations are based on worst-case load. For cost effectiveness reasons, this is unfeasible even if the worst-case load would be known (which is typically not the case). Structural load fluctuations can (to a limited extent) be addressed in the virtual platforms by reallocating unused reservations or dynamically adapt the reservations to

increase/decrease the virtual platform capacity. However, high-volume electronics products stress the platform resource utilization to the limit. At a given point, the required load of the concurrently executing subsystem will exceed the resource capacity and some subsystem will experience a resource shortage. Resolving temporal overloads within a subsystem is specific to each subsystem; it is therefore the responsibility of the subsystem to resolve this.

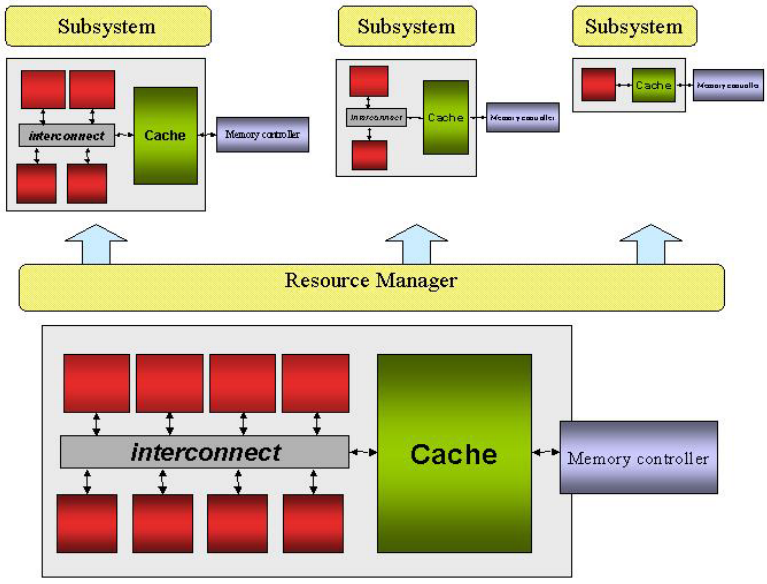


Figure 5-6. Providing virtual platforms to subsystems.

The following section presents the first step towards implementing a virtual platform: the resource reservation mechanisms for the three main SoC resources: CPU cycles, cache space and memory access cycles. *Figure 5-6* depicts the virtual platform vision, where the resource manager provides each subsystem with its own virtual platform, which are a share of the real SoC.

6. RESOURCE RESERVATION MECHANISMS

Resource reservation is a well-known technique to implement temporal and spatial isolation and to bound interference. A *resource budget* is a

guaranteed resource reservation. The resource reservation mechanism consist of the following four components, identified in (Oikawa & Rajkumar 1998).

- The scheduling/arbitration/allocation algorithm determines the run time execution. The scheduling algorithm is such that it matches the budget requirements.
- Accounting keeps track of budget usage.
- Enforcement, denying resource availability when the budget is exhausted, is required to provide guarantees.
- Admission control ensures that once a reservation has been accepted by the system, the budget will be guaranteed.

Sections 6.1 through 6.3 detail these four components of the resource reservation mechanism for the three main SoC resources: processing cycles, cache space, and memory access cycles.

6.1 Processing cycles

Multiprocessor SoCs embed various providers of processing cycles, from a dedicated, non-shareable MPEG-2 accelerator to a multitasking programmable DSP. Managing resource reservations on a multitasking resource—shared by many tasks with diverse real-time requirements—is more challenging than managing access to a hardware accelerator that typically can handle only one task. Therefore, we focus on multitasking programmable processors.

There are different implementations of processor reservation mechanisms (Lipari & Bini 2003), (Eide, Stack, Regehr, & Lepreau 2004),(Rajkumar, Juwa, Molano, & Oikawa 2001). We present our approach to processing cycles reservations in the following subsections.

Resource users

The users of processing cycles are the software subsystems, where the subsystem is temporally independent from other subsystems. Typically, subsystems consist of collections of connected tasks. We distinguish two types of subsystems.

- *Media processing*. This type of subsystem processes media streams with a highly regular pattern. A video decoder for example, produces a video frame every 20 milliseconds. The behavior of a media processing task can be described by a request period T, execution cycle requirement C, and a deadline D, where $D = T$.
- *Control*: Control subsystems have an irregular activation pattern with a minimum inter-arrival time, and their expected response time is short

(compared with the inter-arrival time). They can be described by a minimum inter-arrival time T, a processing-cycles requirement C, and a deadline D, where $D << T$.

Budget definition

CPU-cycle budgets are provided to subsystems and must match the CPU cycle requirements of the subsystems, as described in the previous paragraph. Media processing subsystems typically require periodic budgets with a budget value C (number of processing cycles), a granularity T (period of activation), and a deadline D. A periodic budget is replenished at regular intervals. Control subsystems typically require sporadic budgets which provide a limited amount of computation budget, C, during a time interval called the budget replenishment period, T. The sporadic budgets preserve and limit a certain amount of CPU cycles for the control subsystems, while guaranteeing the deadlines of all the other subsystems in the system, even under burst conditions in the activation of control subsystems (i.e., large number of requests in a short time interval). The sporadic budget is easily incorporated into rate monotonic analysis (Klein 1993), because aperiodic activations can be analyzed as if they were periodic.

Budgets can be strictly enforced, the subsystem receives only its requested C per T, or weakly enforced, a subsystem may receive more than requested if all other subsystems are out of budget. The advantage of strictly enforced budget is predictability: the subsystems always receive the same budget. The advantage of weak enforcement is high utilization.

Scheduling algorithms

The reservation algorithm for CPU cycles can be based on rate monotonic scheduling (RMS) or earlier deadline first (EDF) scheduling. These algorithms, (Liu & Layland 1973) were initially conceived for independent executing task. In our case, individual tasks are not independent whereas subsystems are. The same reasoning that used to apply to tasks applies now to budgets.

- *Rate monotonic scheduling.* Given fixed-priority scheduling, the optimal priority assignment for periodic budgets is the *rate monotonic* (RM) priority assignment. Budgets are ordered by increasing period, ties broken arbitrarily, i.e., $i < j \Leftrightarrow T_i \leq T_j$. The budget scheduling mechanism is built on top of a regular fixed priority scheduler. At the start of each period, the priority of all tasks within the subsystem is raised to the subsystem's running priority. When the subsystem budget is depleted, the subsystem's priority is lowered to background priority.

- *Earliest deadline first scheduling*. In earliest deadline first (EDF) scheduling, budgets are dynamically ordered by increasing deadlines, ties broken arbitrarily. At the start of each period, the deadline of the budget is set. The selected budget is the one with the earliest deadline among the non-zero budgets.

In the case of weak enforcement, when all budgets are exhausted, a slack allocation mechanism is used to immediately allocate the otherwise wasted, volatile, processor cycles. For example, in the case of fixed-priority scheduling, a very simple slack-allocation algorithm consists of making all budgets eligible for execution on a round-robin basis, by giving them the same background priority.

Accounting

Accounting takes place in the CPU reservation module, which maintains a subsystem descriptor per subsystem. This subsystem descriptor contains a down counter that keeps track of the processing cycles used by the subsystem. Every task context switch, the accounting system determines which task (and which subsystem) has executed and for how long. The corresponding amount is deducted from the budget counter. The processor clock is used to keep track of the time.

Enforcement

The budget is enforced by using high precision (hardware) timers that are fired when a budget is exhausted or when a budget has to be replenished.

Admission control

For a single processor system, we use an admission control algorithm that corresponds to the scheduling algorithm being used. If the admission control fails, the corresponding budgets cannot be guaranteed, therefore the subsystem corresponding to the budgets that causes the failure is not allowed to start (or to modify its resource requirements). When using RMS as scheduling algorithm a simple equation (5-1) for response time calculation from (Joseph & Pandya 1986) is used:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j \leq D_i . \qquad (1)$$

In this formula, index i identifies the budget, T_i is the period, C_i is the budget capacity, R_i is the response time, and $hp(i)$ is the set of all budgets with priority higher than i. When using EDF, an even simpler capacity check (5-2) is used:

$$\sum_{all\ i} \frac{C_i}{T_i} \leq 1 . \quad (2)$$

Note that it does not make sense to provide budgets to individual tasks in a single subsystem, because the temporal interdependencies among the tasks invalidate these acceptance tests assumptions.

6.2 Cache space

Caches are divided into cache lines, also called blocks. Cache lines are grouped into sets. A memory location is mapped to a cache set depending on its address and it can occupy any line within that set. A cache with 1 line per set is called direct-mapped, a cache with k lines per set is called k-way set-associative, and a cache with only 1 set is called fully associative. When a line is loaded into the cache, the address determines the set into which the line is loaded. In a direct mapped cache, there is only one choice for replacement, determined by the address. In a k-way set-associative cache, there are k lines that can be victimized.

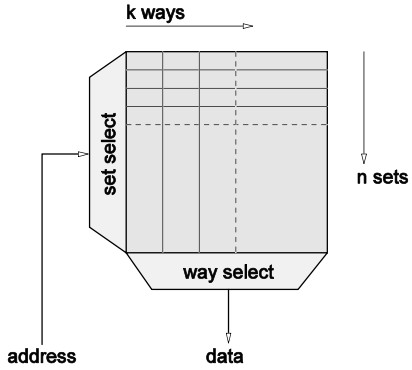


Figure 5-7. Generic cache architecture.

Figure 5-7 shows a generic k-way set-associative cache architecture. The address of a load or store operation is first translated into a *set index* that

uniquely identifies the set where the data is cached (if it is cached at all). Within each set there are *k* blocks. An associative search, *tag matching*, is required to determine which block, if any, contains the data corresponding to the specified address. If the addressed block is not found, one of the *k* blocks in the set is *victimized*: dirty data is copied from the victim block to memory, while the requested data is copied from memory to the victim block.

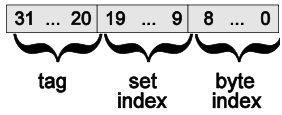


Figure 5-8. L2 cache addressing.

Figure 5-8 depicts how a cache block is addressed. Tag matching is used to locate the place within the set where the data is placed. Since tag matching is performed on the most significant bits, the data of e.g. one MPEG frame is distributed over all sets. Thus, a linear memory access pattern results in a uniform distribution of accesses over the cache.

A known issue with hardware-managed caches is the interference between multiple independent software tasks that share the same cache. Various approaches in the literature address this problem. See, for example (Liedtke, Haertig, & Hohmuth 1997;Molnos, Heijligers, Cotofana, & van Eijndhoven 2005;Ravi 2004).

In the SoC depicted in *Figure 5-2*, there are two types of caches: L1 and L2. The L1 cache is shared when the corresponding CPU supports multitasking. Upon a task context switch, new task data is loaded into the cache, evicting the exiting-task data out from the cache. When the evicted task is executed once again, its data has to be reloaded, causing an extra performance penalty compared to the case when the task runs without interruption. However, in a multiprocessor system, the CPUs are typically dedicated to a small number of tasks: for example, one CPU takes care of coprocessor management, another takes care of network traffic, etc. As a result, the remaining CPUs can concentrate on running applications without being interrupted by housekeeping jobs. Consequently, L1 cache interference appears to be a less urgent problem in a multiprocessor than it is in a single CPU system.

For the L2 cache, a different story applies. This single cache is shared concurrently by the tasks in the system. For example, a Linux operating system executing on one or more CPUs may suddenly require a lot of memory when it starts an Internet browser with Java support. We want to avoid that this action in the Linux domain evicts critical data and code

sections in another domain, for example a real-time video codec running on the DSPs sharing the same L2 cache. We define a *domain* as the collection of tasks that share a determined cache space.

In this section, we focus on a cache management mechanism for the L2 cache. The following sections describe our approach to cache management to bound interference among the various application domains that execute concurrently on the multiprocessor.

Resource users

The users of the cache are the cache domains or collection of software tasks. The software tasks request from the cache load (read) and store (write) operations. This operation can result either on a hit (the requested data is cached) or a miss (the requested data is not in cache). Upon a cache miss, data has to be brought from main memory victimizing cached data.

Budget definition

Available cache partitioning methods (Liedtke, Haertig, & Hohmuth 1997;Molnos, Heijligers, Cotofana, & van Eijndhoven 2005) allocate parts of the available cache sets exclusively to a subset of the executing tasks (*Figure 5-9*). The disadvantage of this approach is that it affects the memory model as seen by the software programmer. For example, if task A writes to memory location X, the data is cached in partition A. If task B reads from memory location X at a later moment in time, it cannot find the data in cache partition B and consequently the stale data is loaded from memory into partition B. This is probably not what the programmer expected.

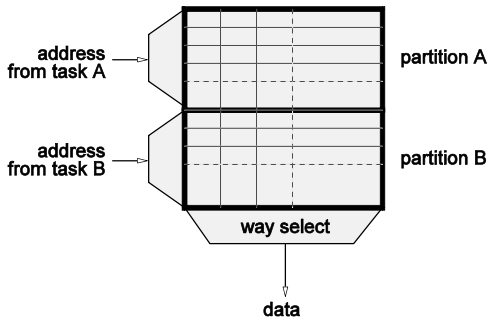


Figure 5-9. Traditional cache partitioning.

In contrast to set partitioning, we chose to partition the cache by limiting the number of *ways* a task can claim within each set in the cache. *Figure 5-10* depicts the resulting cache organization. Cache resource management is performed by allocating cache space to a domain. During cache lookup, all ways in the set are considered, including the ways associated with domains other than the one performing the lookup, i.e., any task can read or write any cache block with no restrictions. Consequently, the shared memory model remains intact. The programmer does not notice any functional difference between a traditional and a resource-managed cache.

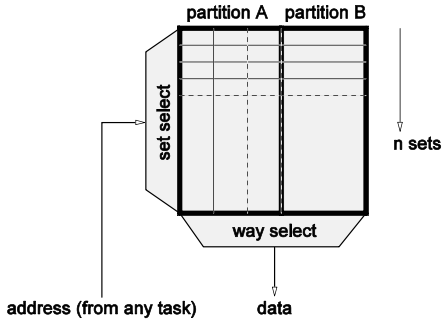


Figure 5-10. Cache way partitioning.

A cache budget determines the maximum number of ways the domain can claim, say *N*. If each way corresponds to a fixed number of bytes, say *M*, each budget corresponds to *N***M* bytes. The cache contains a bit vector for every domain, where each way in the cache is associated with one bit in these vectors. By setting the bit vector in the cache, the system integrator can choose explicitly which domains share which cache ways.

The task descriptor in the OS contains a field that identifies the domain the task belongs to. On every context switch, the OS copies this field to a hardware register. On a cache miss, the cache controller inspects this register to determine which domain is causing the miss and the victim is selected from the ways belonging to this domain.

There are two types of reservations. Cache reservations can overlap (domains can share ways) or be disjoint (no ways are shared). When two domains share a way, in the worst case, one of these domains can evict all data of the other domain from the cache. If all domains reserve disjoint ways, the reservation is not shared: tasks belonging to domain A cannot evict data from domain B. Overlapping domains are useful when the worst case cache requirement is far from the average. Each domain reserves disjoint

space to be used during normal behavior and overlapping ways for the worst case.

Replacement algorithm

The replacement algorithm, that selects which cache block is victimized, is the equivalent of the scheduling algorithm for the CPU. The cache employs a random replacement strategy. When a task belonging to domain causes a refill, a victim block is selected from the corresponding domain, such that the number of ways associated with the domain in the set does not exceed the predetermined budget for the domain. The replacement algorithm only applies during a cache refill, following a cache miss.

Accounting

Accounting has to keep track of the number of ways allocated to a particular domain in a particular set of the cache.

Enforcement

If a bit is set in the vector of a selected domain, the domain can access the cache blocks in the way corresponding to the bit position in the vector.

Admission control

The admission control for a cache reservation request is simple. The total amount of requested space should not exceed the total cache size.

6.3 Memory access cycles

As presented in Section 3, data transfer to and from memory is becoming the main system bottleneck. As an example, *Figure 5-11.* depicts the structure of the memory path of the SoC. The memory controller has three available ports. Two of these ports are used by the refill and victim engines of the L2 cache. The third port is used by the hardware accelerators.

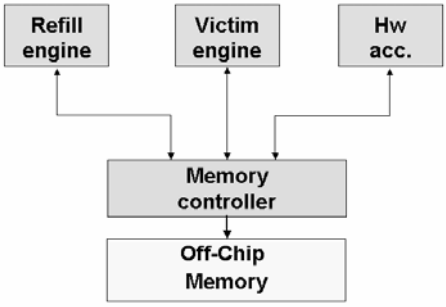


Figure 5-11. Memory controller ports.

This section focuses on the memory access cycles: the cycles available for data transfer from the memory controller ports to the off-chip memory. Similar to processing cycles, memory access cycles constitute a volatile resource: a memory access cycle that is not granted to a requester is lost forever. However, the allocation granularity for memory access cycles is a few orders of magnitude smaller than the allocation granularity for CPU cycles.

The order in which requests are presented to the memory has a large impact on the efficiency of the memory access. For example, if a write transfer follows a read transfer, the transition overhead, which consists of the cycles needed to invert the direction of the data channel, is similar to the cost of the transfers. It is very difficult, if not impossible, to guarantee net transfer cycles (excluding overhead cycles). Instead, gross transfer cycles (including overhead cycles) rather than net transfer cycles are guaranteed, and overhead cycles are attributed to the request that causes the overhead.

The reservation scheme for memory access cycles assumes a mix of low-latency traffic and high-bandwidth traffic and tries to minimize the average latency for the low-latency traffic while meeting the bandwidth requirements for the high-bandwidth traffic. Typically, cache engines generate low-latency traffic, whereas hardware accelerators generate high-bandwidth traffic.

Resource users

On behalf of the tasks they execute, hardware blocks and cache engines issue memory requests that consume memory access bandwidth. The arrival and servicing of memory requests is described by two functions of the number of memory-clock cycles (t): the request function *R* and the supply function *S*. Both functions are taken from (Feng & Mok 2002), and are depicted in *Figure 5-12*.

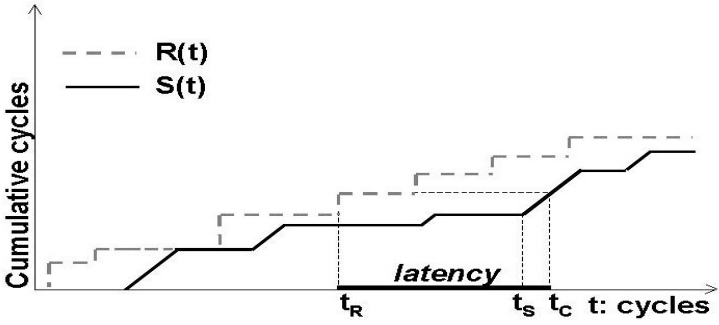


Figure 5-12. Request and supply functions.

The request function R(t) represents the total number of cycles *requested* in the interval (0, t), whereas the supply function S(t) represents the total number of cycles *supplied* in the interval (0, t). R(t) is a simple staircase function, for which every step represents the arrival of one or more multi-cycle requests. If the requests arriving at t_R have total gross size s($s \geq 0$), then

$$\lim_{t \downarrow t_R} R(t) = R(t_R) + s. \tag{5-1}$$

In the $S(t)$ function, supply intervals alternate with still intervals. Cycles are supplied in the supply intervals only:

$$S(t + \Delta t) = S(t) + \Delta t, \text{ when } (t, t + \Delta t) \text{ is a supply interval;} \tag{5-2}$$

$$S(t + \Delta t) = S(t), \quad \text{when } (t, t + \Delta t) \text{ is a still interval.} \tag{5-3}$$

The number of supplied cycles can never be larger than the number of requested cycles. A request is characterized by size s, arrival time t_R, start time t_S, completion time t_C, and latency λ. From t_R to t_S, the request is pending; from t_S to t_C, the request is being serviced. (t_S, t_C) is the service interval for the request.

$$S(t) \leq R(t). \tag{5-4}$$

$$t_S = max\{t \mid S(t) = R(t_R)\}. \tag{5-5}$$

$$t_C = min\{t \mid S(t) = R(t_R) + s\}. \tag{5-6}$$

$$\lambda = t_C - t_R. \tag{5-7}$$

Different requesters will be identified by an index to the request and supply functions. For requester i, the request and supply functions are denoted $R_i(t)$ and $S_i(t)$. Since every cycle can be supplied at most once, different requesters have disjoint supply intervals:

$$S_i(t+\Delta t) = S_i(t) + \Delta t \implies S_i(t+\Delta t) = S_j(t)\ \forall\, j \neq i. \tag{5-8}$$

Typically, there are two different types of hardware blocks, with different request characteristics and different service requirements. *High-bandwidth requests*, typically issued by hardware accelerators, tend to have a regular request pattern, and require effective use of memory bandwidth. In media systems, these requests represent the bulk of the traffic. High-bandwidth traffic generally has latency requirements that are individually, but not tightly, bounded. *Low-latency requests* have an irregular request pattern with potentially large bursts, and require low average latencies. Individual request do not have bounded latency requirements. Low-latency bursts can be accommodated because of the relatively large latency bounds of the regular traffic.

The descriptions in this section are restricted to a single low-latency requester (*LL*) and a single high bandwidth requester (*HB*). This simplification helps to focus on the quintessence of the reservation mechanism. In this area, research is still in progress and details are not yet available for publication.

Budget definition

The budget definition for the low-latency budget is given in two steps. In the first step, we make a simplifying assumption: the budget boundaries are assumed to be hard, i.e., out-of-budget cycles are not supplied, even if no other requester is contending for them. With this assumption, the reservation mechanism is non-bandwidth preserving (idle memory cycles while requests are pending), but easy to explain. In the second step, this assumption is dropped.

The low-latency budget (*LL*) can be compared to a credit card, where the customer borrows from the bank, and pays back later. *LL* goes through a sequence of active and inactive intervals. During an active interval, *LL* is either borrowing or paying its debt.

By definition, an active interval starts at $t = 0$. During each active interval, the low-latency budget is defined by two functions $UB_{LL}(t)$ and $LB_{LL}(t)$, the upper and lower bound, respectively. In the first step we assume that these functions bound the supply function $S_{LL}(t)$ directly:

$$LB_{LL}(t) \le S_{LL}(t) \le UB_{LL}(t) \quad (5\text{-}9)$$

Figure 5-13 depicts one active interval of a low-latency budget. The gray band represents the bounds that the budget imposes on $S_{LL}(t)$. At this point in time, *LL* is requesting, and $S_{LL}(t)$ starts its first supply interval after the arrival of the pending request(s). At $t = 0$, $S_{LL}(t)$ starts its first supply interval, and *LL* becomes active. At $t = a$, the upper bound is hit, no more credit is available, and the requester starts paying back. At $t = b$, there is sufficient credit again to resume supplying. At $t = c$, there are no more requests pending, and the requester starts paying back again. At $t = d$, a new burst of requests arrives. At $t = e$, supplying resumes. Finally, at $t = e$, the complete debt has been paid back. If there is no request pending and there is no remaining debt, the requester becomes inactive.

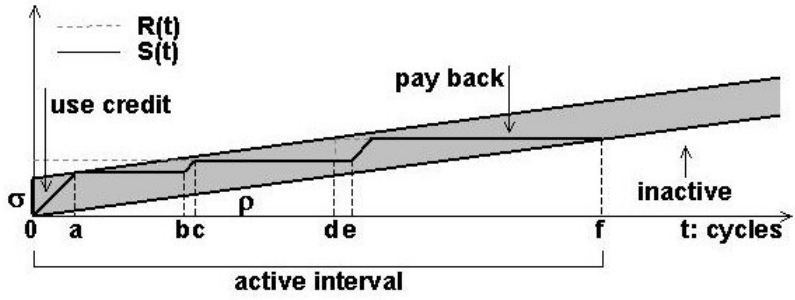


Figure 5-13. Low latency contract.

The lower bound corresponds to a function ρt, where ρ is a fraction of the available cycles, with $0 < \rho << 1$. The vertical distance between the two bounds, σ, determines the burst size accommodated by the budget. The σ and ρ parameters are taken from the sigma/rho (σ/ρ) abstraction, used for traffic characterization in network calculus (Cruz 1991). With these parameters, the low-latency budget is given by

$$UB_{LL}(t) = \rho t + \sigma, \quad (5\text{-}10)$$

$$LB_{LL}(t) = \rho t. \quad (5\text{-}11)$$

This completes the first step, in which we assumed that upper bound is *hard*. This hard upper bound implies that out-of-budget supply is not

allowed, even when *HB* is not requesting. This is a waste of bandwidth, and has a negative impact on the average *LL* latency as well.

When the upper bound is not hard, equation (5-9) does not necessarily hold. To define soft bounds, some additional terminology is needed. The functions $IBS(t)$, intra-budget supply, and $XBS(t, \Delta t)$, extra-budget supply, are defined by the following equations:

$$S(t+\Delta t) = S(t)+\Delta t \Rightarrow IBS(t+\Delta t) = \min(IBS(t)+\Delta t, UB(t+\Delta t)), \quad (5\text{-}12)$$

$$S(t+\Delta t) = S(t) \Rightarrow IBS(t+\Delta t) = \max(IBS(t), LB(t+\Delta t)), \quad (5\text{-}13)$$

$$IBS(t') = UB(t')\ \forall t' \in (t, t+\Delta t) \Rightarrow XBS(t, t+\Delta t) = (S(t+\Delta t) - S(t)) - (UB(t+\Delta t) - UB(t)), \quad (5\text{-}14)$$

$$IBS(t') < UB(t')\ \forall t' \in (t, t+\Delta t) \Rightarrow XBS(t, t+\Delta t) = 0, \quad (5\text{-}15)$$

$IBS_{LL}(t)$ can take values between 0 and σ_{LL}. $XBS_{LL}(t_S, t_C) > 0$, extra-budget supply for an *LL* request with service interval (t_S, t_C), is allowed only if *HB* is not requesting at t_S.

Arbitration algorithm

The arbitration algorithm decides on how to allocate the cycles. It is priority-based, and uses three priorities, two for *LL* (default and limit), and one for *HB*. The *LL* default priority is higher than the *HB* priority; the *LL* limit priority is lower than the *HB* priority. In the CPU domain, this dual priority scheme is known from bandwidth-limiting servers(Burns & Wellings 1993). In the following subsections it becomes clear when these priorities apply.

Arbitration is non-preemptive. Ongoing transfers are completed, even when a higher-priority request arrives. This has to be the case, because preemption is detrimental to the efficiency of the memory (causes many overhead cycles). In the discussion of the enforcement mechanism, the consequences of the choice are addressed in more detail.

The description of the implementation corresponds to a very elegant solution, conceived by Hans van Antwerpen at Philips Semiconductors, used in the arbiter of a double data rate (DDR) memory controller (de Oliveira & van Antwerpen 2003).

Accounting

The accounting mechanism is depicted in Figure 4-12. It uses a saturating counter ACCOUNT, which saturates at 0 and CLIP. ACCOUNT is initially 0, and is increased or decreased every cycle. ACCOUNT keeps track of $IBS_{LL}(t)$. It is updated every cycle. If the cycle is allocated to the requester, ACCOUNT is increased with DEN – NUM, otherwise it is decreased with NUM. NUM stands for Numerator, and DEN stands for Denominator.

$$\text{NUM/DEN} = \rho_{LL}. \tag{5-16}$$

$$\text{CLIP/NUM} = \sigma_{LL}/(1-\rho_{LL}). \tag{5-17}$$

$$\text{ACCOUNT/NUM} = IBS_{LL}(t). \tag{5-18}$$

In the budget definition, NUM, DEN and CLIP replace the original ρ and σ. One of these values can be freely chosen, the others then follow from (5-16) and (5-17). Choosing a round value for NUM, which is somewhat counter intuitive, the CLIP value becomes more intuitive.

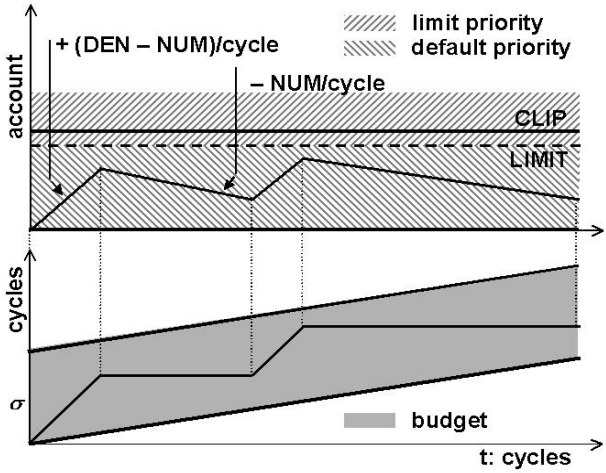


Figure 5-14. Priorities, accounting, and enforcement.

Enforcement

Enforcement makes sure that the *LL* priorities are switched at the appropriate times. The decision to raise or lower the *LL* priority is based on

comparing ACCOUNT with a threshold LIMIT. If ACCOUNT < LIMIT, then *LL* has default priority; otherwise, *LL* has limit priority. If $\max(s_{LL})$ is the maximum gross *LL* request size, then

$$\text{LIMIT} = \text{CLIP} - \max(s_{LL})*\text{NUM}. \quad (5\text{-}19)$$

The threshold value LIMIT must be such that the boundary constraint of the *LL* budget is satisfied. Extra-budget supply, $XBS_{LL}(t_S, t_C) > 0$, requires that $IBS_{LL}(t_S) > \sigma_{LL} - s$, where s is the size of the request. Because of (5-16) through (5-19), this implies ACCOUNT > LIMIT at t_S, which in turn implies that *LL* has limit priority at t_S. If *LL* has limit priority, the request can only be serviced if *HB* is not requesting. Hence, extra-budget supply is only possible if *HB* is not requesting, which was the desired effect.

For implementation simplicity, LIMIT is currently implemented as a programmable parameter. In order to minimize the number of stall cycles, the DDR controller has a small queue of *LL* requests after arbitration. Hence, in a real implementation, LIMIT/NUM has to be larger than $\max(s_{LL})$, depending on the size of this queue.

Admission control

By definition, admission control decides if a certain combination of contracts is feasible. Since there is only one contract, there is no admission control.

7. CONCLUSION

A major source of robustness problems in current generation systems is the unpredictable behavior caused by interference among concurrently executing applications that compete for access to shared system resources—such as processor cycles, cache space, and memory access cycles. Traditionally, these aversive effects of interference could be kept under control by deploying a real-time OS in combination with a sufficient degree of over provisioning.

For today's systems, this approach is no longer viable. The price erosion in the consumer electronics market forces chip vendors to integrate more and more functionality in an SoC, at the expense of system robustness. For instance, while previous generation SoCs separated real-time audio/video hardware from general-purpose hardware to handle user events, today's multiprocessor SoCs deploy generic processor and interconnect hardware that handle both.

This chapter outlines an approach to bound interference among independently developed subsystems. The system provides each subsystem with an execution environment—called a virtual platform—that emulates the environment in which the subsystem was developed and tested. A subsystem reserves a share of each required system resource. This set of reservations defines the virtual platform. All shared resources in the virtual platform must provide guaranteed reservations to subsystems, or deny a reservation request when the request exceeds the available capacity. The research challenge towards such compositional systems is threefold.

- Define hooks in hardware and software with associated strategies to provide and guarantee reservations for *every* shared system resource.
- Provide an overall resource management strategy that integrates the individual reservation strategies of each shared resource.
- Define an approach to characterize subsystems in terms of execution requirements that can be translated into the desired resource reservations.

This chapter takes on the first challenge and presents reservation mechanisms for the key resources in a multiprocessor SoC: processor cycles of a CPU, cache space in an L2 cache that is shared among multiple processors, and memory cycles arbitrated by a DDR memory controller. The described DDR controller is currently deployed in Philips Nexperia solutions, while the processor reservations are proposed for integration in embedded operating systems, such as CE Linux. The presented cache space reservations are targeted for inclusion in the next generation Philips Nexperia SoCs.

ACKNOWLEDGEMENT

The authors want to express their gratitude to Peter van der Stok, and Kees Goossens for their review comments.

REFERENCES

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., & Warfield, A. 2003, "Xen and the art of virtualization", *Proceedings of the nineteenth ACM symposium on Operating systems principles* pp. 164-177.

Baruah, S. & Lipari, G. "A multiprocessor implementation of the total bandwidth server", in *Proceedings 18th International Parallel and Distributed Processing Symposium*, pp. 40-49.

Burns, A. & Wellings, A. J. 1993, "Dual-priority Assignment: A practical method for increasing processor utilization", in *Proceedings of 5th Euromicro Workshop on Real-Time Systems*, Oulu, Finland, pp. 48-55.

Cruz, R. L. 1991, "A Calculus for network delay, part I: network elements in isolation", *IEEE Transactions an Information Theory*, vol. 37, no. 1, pp. 114-131.

Cumming, P. 2003, "The TI OMAP™ Platform Approach to SoC," in *Winning the SoC Revolution*, G. Martin & H. Chang, eds., Kluwer Academic, pp. 97-118.

de Oliveira, J. A. & van Antwerpen, H. 2003, "The Philips Nexperia™ Digital Video Platform," in *Winning the SoC Revolution*, G. Martin & H. Chang, eds., Kluwer Academic, pp. 67-96.

Eide, E., Stack, T., Regehr, J., & Lepreau, J. 2004, "Dynamic CPU management for real-time, middleware-based systems", in *Proceedings 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* , pp. 286-295.

Feng, X. & Mok, A. 2002, "A Model of Hierarchical Real-Time Virtual Resources", in *Proceedings IEEE Real Time System Symposium*, Austin, USA, pp. 26-35.

Goossens, K. & González Pestana, S. 2004, "Communication-Centric Design for Real-Time Consumer-Electronics Systems on Chip," in *Dynamic and robust streaming in and between connected consumer-electronic devices*, P. van der Stok, ed..

Joseph, M. & Pandya, P. 1986, "Finding response times in a real-time system", *British Computer Society Computer Journal*, vol. 29, no. 5, pp. 390-395.

Klein, H. 1993, *A Practitioner's Handbook for Real-Time Analysis* Kluwer Academic Publishers.

Liedtke, J., Haertig, H., & Hohmuth, M. 1997, "OS-Controlled Cache Predictability for Real-Time Systems", in *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS)*, IEEE Computer Society, pp. 213-227.

Lipari, G. & Bini, E. 2003, "Resource partitioning among real-time applications", in *Proceedings 15th Euromicro Conference on Real-Time Systems* , pp. 151-158.

Liu, C. & Layland, J. 1973, "Scheduling algorithms for multiprogramming in a hard real-time environment", *Journal of the ACM*, vol. 20, no. 1, pp. 46-61.

Molnos, A., Heijligers, M. J. M., Cotofana, S. D., & van Eijndhoven, J. 2005, "Compositional memory systems for multimedia communicating tasks", in *Proceedings of Design Automation and Test in Europe (DATE)*, Munich, Germany.

Nolte, T. & Kwei-Jay, L. 2002, "Distributed real-time system design using CBS-based end-to-end scheduling", in *Proceedings. Ninth International Conference on Parallel and Distributed Systems*, pp. 355-360.

Oikawa, S. & Rajkumar, R. 1998, "Linux/RK: A Portable Resource Kernel in Linux", in *Proceedings IEEE Real-Time Systems Symposium Work-In-Progress*.

Otero Pérez, C. M., Steffens, E., Loo, G. v., Stok, P. v. d., Bril, R., Alonso, A., Garcia Valls, M., & Ruiz, J. 2003, "QoS-based resource management for ambient intelligence," in *Ambient Intelligence: Impact on Embedded System Design*, T. Basten, M. Geilen, & H. de Groot, eds., Kluwer Academic Publishers, pp. 159-182.

Paulin, P., Pilkington, C., & Bensoudane, E. 2002, "StepNP: A System-Level Exploration Platform for Network Processors", *IEEE Design & Test of Computers*, vol. 19, no. 6, pp. 17-26.

Rajkumar, R., Juwa, K., Molano, A., & Oikawa, S. 2001, "Resource kernels: A resource-centric approach to real-time and multimedia system," in *Readings in multimedia computing and networking*, Morgan Kaufmann Publishers Inc., pp. 476-490.

Ravi, I. 2004, "CQoS: a framework for enabling QoS in shared caches of CMP platforms", in *Proceedings of the 18th annual international conference on Supercomputing*, ACM Press, pp. 257-266.

Seawright, L. & MacKinnon, R. 1979, "VM/370 -- a study of multiplicity and usefulness", *IBM Systems Journal*, vol. 18, no. 1, pp. 4-17.

Stravers, P. & Hoogerbrugge, J. 2001, "Homogeneous multiprocessing and the future of silicon design paradigms", *Proceedings of the International Symposium on VLSI Technology, Systems, and Applications(VLSI-TSA)*.

van Eijndhoven, J., Hoogerbrugge, J., Nageswaran, J., Stravers, P., & Terechko, A. 2005, "Cache-Coherent Heterogeneous Multiprocessing as Basis for Streaming Applications," in *Dynamic and robust streaming in and between connected consumer-electronic devices*, P. van der Stok, ed..

Chapter 6

STREAMING IN CONSUMER PRODUCTS

Beyond processing data

Giel van Doren, and Bas Engel
Philips Research Laboratories, Eindhoven, The Netherlands

Abstract: The signal processing for TV is changing rapidly at this moment. The classical analog broadcast is being replaced by digital broadcast, CRTs are being replaced by Matrix displays, and the convergence between PC and CE introduces PC standards and applications in the TV domain. Collectively, these changes have a big impact on the signal processing architecture of a TV. Signal processing has always been a field of competence for Philips. To keep a leading position in this area, Philips has developed the Nexperia Home Platform. This platform is a mix of both hardware (HW) and software (SW). SW gives the flexibility needed for configuring such a platform for a range of products. HW allows cost-effective implementations of compute intensive signal processing. This chapter discusses the required SW streaming infrastructure in such a platform. A SW streaming infrastructure is an enabler to fulfill the streaming requirements, and to provide the required flexibility in the platform to come to a cost-effective solution. We will explain requirements on the streaming infrastructure in a platform by looking at Hardware/Software (HW/SW) tradeoffs, real-time requirements constraints, and the complexity of controlling signal processing. This chapter will take the examples from the TV and Storage domain.

Key words: Streaming infrastructure, architecture, real-time streaming constraints, streaming design tradeoffs, streaming platform, HW/SW co-design.

1. INTRODUCTION

Consumer products in the shop today have a similar appearance to the ones from several years ago. A DVD recorder can be seen as an evolved VCR and a classical color TV even looks like today's color TV. Both still have a remote control, that allows you to change channels, adjust volume, start a recording, or simply turn the set on. Even the consumer price is comparable.

P. van der Stok (ed.),
Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, 139-165.

However, in these devices much has changed over the last few years. There is an enormous increase of processing power to, for example, improve picture quality. Where we used to have dedicated hardware to do the tuning, color decoding, de-interlacing, image enhancement, and scaling, we now have a mix of dedicated hardware and flexible software processing. The overall complexity of the processing has increased dramatically due to an increase in the number of standards and formats that have to be supported. In addition, the price erosion makes that high-end consumer products become mainstream in a short time. Therefore, solutions are required that enable a short lead-time to introduce new features. Consumer electronic vendors need to be able to spread their investment over a range of products, which can have different processing hardware depending on the market positioning. This requires a different approach than years ago. Philips' solution towards this is the use of a SW streaming platform that provides a hardware independent interface on top to allow independent evolution of the platform and the middleware on top. The first half of this chapter will focus on the trends, and the impact of the trends on the streaming platform. The second half will discuss the streaming technology that should support tradeoffs inside the platform to fulfill all imposed constraints.

1.1 Trends

There is a number of ongoing trends in the TV/Digital Versatile Disk (DVD) domain, that Philips has to follow to stay a player in this market.

The first trend is the continuous improvement of picture quality. Both for a DVD recorder compared to a VCR, and for today's TV with one of years ago, the picture quality has clearly been improved.

The second trend is digitalization, both in storage and in transmission. The analogue tapes are almost completely replaced by digital media like DVD, CD, and harddisk. At the input, consumer products are changing from classical analog reception to digital reception. Until recently digital reception was addressed solely by separate proprietary set-top boxes. The trend towards the inclusion of digital reception in standard consumer products is greatly accelerated by US legislation enforcing digital reception to be incorporated in all television sets by 2007. Consumer products in the home will deliver content from much more diverse sources than the classical broadcast only. They incorporate digital interfaces (like Ethernet, IEEE 1394, USB and wireless) to connect to those sources.

The third trend is the transition to flat digital displays like LCD and plasma.

The fourth trend is the increasing amount of software in television sets (>200 person-years). The reasons for this are the addition of large interactive

"digital" services like Multimedia Home Platform/ Open Cable Application Platform (MHP)/OCAP and or Multimedia and Hypermedia Expert Group MHEG and multimedia viewers in combination with the expanding set of digital standards (e.g. MPEG4 promoted by the Moving Picture Experts Group (MPEG), DivX) that are often realized in software (SW).

The last trend is the convergence of Consumer Electronics (CE) and PC domains, which introduces a number of formats like MP3 and DivX and applications like still picture viewing and content browsing from the PC domain into the TV domain.

These trends impact the streaming platform in consumer products as we will show in this chapter. This chapter will mainly describe the TV domain, although many of presented concepts also apply to the DVD domain.

1.2 Signal Processing

In principle, a TV is nothing more than a signal processing device that is able to capture an incoming signal, transform it into a basic stream of information, do all sorts of enhancements to improve the picture quality, and finally render it on a display as depicted in Figure 6-1. This sequence of processing steps is called signal processing.

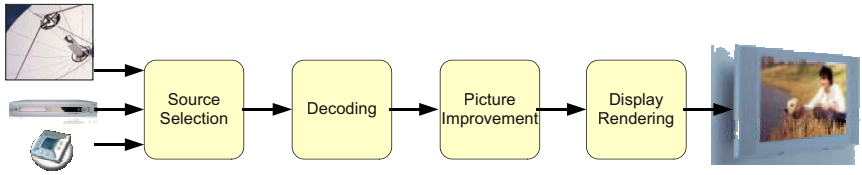


Figure 6-1. Signal processing steps in a TV

The first processing step is source selection. A TV is not a single-input device as it used to be. Nowadays, a TV can handle a diverse set of input signals, ranging from antenna and DVD to IEEE 1394, Ethernet, and memory stick. These input signals do not only differ in transportation medium, but also in size and encoding. All these inputs can require different processing steps to get a picture on the screen. Scaling, transcoding, and format conversions towards desired physical video formats (e.g. YUV 4:2:2) are part of the signal processing.

The second step is decoding. This is the process of transforming information from an encoded form to its 'natural' form. For the traditional analog TVs, decoding demodulates the chrominance signal to yield two

color (sub)signals and one luminance signal according to analog standards like NTSC or PAL. Digital TV's are based on digital broadcast standards like ATSC. These standards are based on MPEG transport streams that contain multiplexed digital audio/video information and require demultiplexing to do the actual audio/video decoding.

The most processing intensive step is the picture improvement processing. Today, it is the most important step with regard to the selling features of a TV. Picture improvement algorithms vary from relatively simple, like noise reduction, to extremely complex and processing intensive, like natural motion to smoothen frame transitions when reducing flickering and Pixel Plus to increase the resolution of the picture.

Finally, the display renderer actually puts the created frame on the screen.

1.3 Paper outline

Section 2 will explain the role of a streaming platform in a CE device. It will give an overview of the overall architecture of consumer products to position the streaming platform and to understand the influences and the requirements that are imposed on it. After this, we will zoom into the streaming platform itself. Section 3 takes a look at the concept of streaming in consumer products. Sections 4 and 5 will show that a streaming infrastructure is essential in a streaming platform, but that there is much more needed than only an infrastructure. Section 4 discusses the mixture between software and hardware processing and the advantages and disadvantages of both of them. Section 5 discusses the impact of various timing requirements on the execution behavior of the streaming platform. Finally, Section 6 will end our discussion of the streaming platform by discussing both the control of the processing algorithm and the management of the complete processing chain.

2. CONSUMER PRODUCTS

The discussed trends involve a significant increase in effort and have impact on the architecture of today's consumer products. This section will discuss how the consumer product architecture addresses these trends. We will do this mainly from a streaming platform point of view. We will also take a look at the main differentiation points in the streaming platform before starting the detailed technical discussion on streaming in the remainder of this chapter.

2.1 Consumer Product Architecture

The global architecture of a consumer product can be represented as a layered structure depicted as follows:

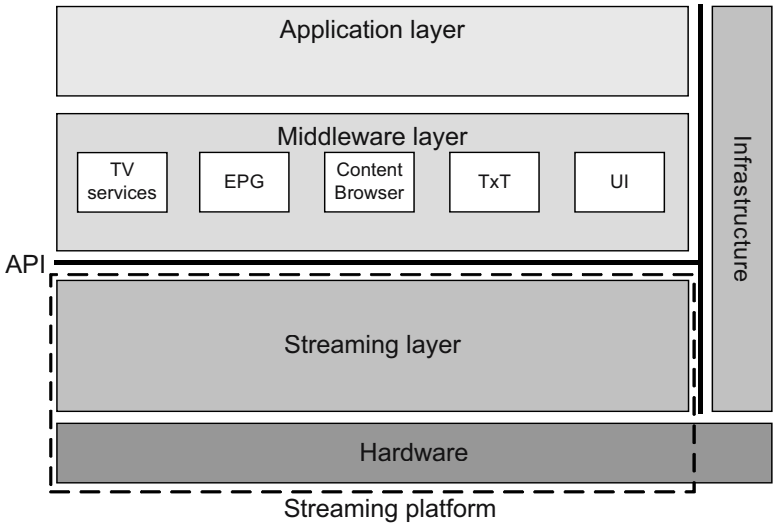


Figure 6-2. Position of the streaming platform in a consumer product architecture

The application layer combines services of the middleware layer into a coherent application together with a user interface. This forms the functionality as offered to the end user and translates user requests to the services offered by the middleware.

The middleware layer takes care of basic TV-services such as installation, program control, user interface rendering, Electronic Program Guide (EPG) and conditional access handling. Furthermore, it implements software to support the factory-required functionality.

The infrastructure layer provides a standard operating system (e.g. Linux, VxWorks), and drivers for all basic peripherals (UART, I2C, GPIO, etc).

The streaming platform layer offers basic functionality to abstract and control the TV's hardware (tuners, decoders, inputs, outputs). The platform provides interface methods that allow the service layer to set-up audio and video processing path's from a source (cable, terrestrial, IEEE 1394 etc) to one of the destinations (display, headphones, Video Cassette Recorder (VCR), IEEE 1394 etc), and offers control interfaces to manipulate the processing in between.

2.2 Industry Dilemma

Until recently most of the software in consumer products was built by the CE manufactures themselves. Nowadays, to build such a consumer product requires millions of lines of code and hundreds of man-years to build the product. Moreover, with the trends we discussed a significant increase is inevitable for tomorrow's products.

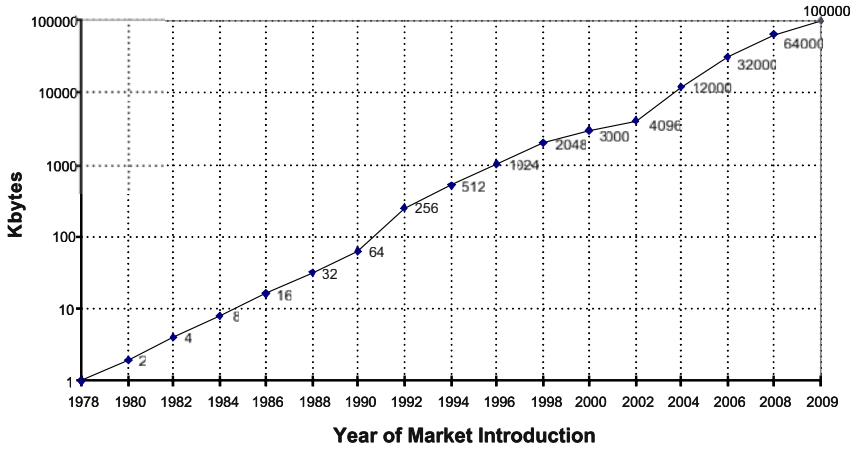


Figure 6-3. Code size evolution of high-end TV software.

Typically, a large part of this software concerns common services that any consumer product should have (EPG, Teletext (TXT), Content Browser, Media Application, Internet Connectivity, etc). This means that it requires huge investments to make it, maintain it, and improve it, while there is no differentiating advantage compared to competitors.

Although the infrastructure in term of Operating System (OS) becomes more and more standard (e.g. Linux), the expected behavior of the signal-processing graph can be very different. This is the case for different Hardware (HW) platforms of one vendor, as well as in between HW platforms of different vendors.

Philips Semiconductors has recently introduced its Nexperia Home Strategy in which all of its (digital) consumer device platforms for TV, DVD+RW, and Settop boxes will export the same interface, the Universal Home Application Programming Interface (UHAPI). This API will make porting of middleware between UHAPI compliant platforms easier. With the increasing overlap and exchange of functionality between these consumer

devices in the connected home this standardization increases the reuse. Moreover, such open standardized API will allow Independent Software Vendors (ISVs) to write software for a broad range of consumer devices more easily. It will help CE companies; they can buy the non-differentiating software.

2.3 Streaming platform

A streaming platform, like the Nexperia Home Platform, supports a range of products. The reason for introducing a streaming platform is to reuse such a platform for a family of products. A platform should provide an interface for a range of products to setup and control the streaming in a product. To increase the reuse of the middleware on top of the platform, such an interface should be stable. A stable interface should abstract sufficiently from the HW to allow independent HW evolutions.

One part of the streaming platform is a streaming infrastructure. Such an infrastructure provides a standard for connecting processing steps. A sequence of connected processing steps is called a streaming graph, where each node is a processing step and an edge represents the communication between two processing steps. Having a standard for connecting processing steps increases the modularity of the processing steps within the platform.

A streaming platform is a mix of hardware and software. It should provide the flexibility to make tradeoffs between HW and SW realizations of certain processing steps. In addition, the platform API must provide a sufficient abstraction that these kinds of tradeoffs can be made without impacting the middleware on top. Due to the diversity between products and the large number of different streaming graphs that have to be supported within a product, the total number of supported graphs in a platform is large.

A graph has to be controlled. With control we mean the ability to manage the flow of data through the signal processing chain. Examples of this are scalar settings, tuner frequencies, contrast settings, etc. Control also includes the default behavior of the platform in case there is a signal drop, a preset change, or a change in input resolution. Controlling a large number of streaming graphs, transitions in between, and streaming graphs running at the same time is even more complex. Within the platform reuse of software is essential when the platform evolves, both for the signal processing code and for the control code in the platform.

Although there is a standardized API, the platform itself should not be a monolithic block but flexible and configurable. Domain and product knowledge is essential to determine what flexibility is required in a platform. CE-manufacturers want to configure a streaming platform to derive their actual products from. Depending on their capabilities and market position

they might choose to add their own components to the basic HW/SW platform or use it as is.

Another example of required flexibility is to support a wide variety of standards. For most of the standards originating from the PC domain, SW implementations are available. That's why more and more of the encoding/decoding is (initially) realized in software or have a significant software part.

Software is essential in achieving the required flexibility in the platform. Software enables the implementation of new codecs and standards on an already existing HW platform. It provides a way to realize late-time requirements, enabling you to be agile with respect of the latest market trends. The downside of SW for signal processing is that it typically costs significant CPU cycles and memory.

2.4 Differentiation points

Differentiation points of a product form the competitive advantage. For a TV sold to end-customers, features like 100 Hz and natural motion are differentiation points. A streaming platform should allow realizing these differentiation points by providing sufficient flexibility.

A streaming platform, like the Nexperia Home platform, is also a product that is sold to CE companies. One can distinguish 3 main differentiation points for a streaming platform. First differentiation is by means of picture quality, especially for flat screen displays. Second differentiation is the ease of use of configuring the streaming platform, which is increasing in importance. This includes the possibility to integrate proprietary algorithms of CE companies in the platform. Third differentiation is to provide low-power solutions. There is a growing awareness that low-power solutions will be a differentiating feature. The most important differentiation point is the ability to do all 3 of them cost effectively.

Differentiation points will change over time. Stereo output on a TV is a good example. It was introduced in high-end TVs that where substantially more expensive than mono TVs. After some time, the stereo feature has become a commodity and was introduced in the lower-end products. More recently, 100 Hz is becoming a commodity feature.

To cope with innovation of features and degradation of them over time, one needs to have a family of products that are closely related, but differ sufficiently to position them in different price segments. A streaming platform should provide sufficient flexibility to create a high volume low-cost solution and a more expensive solution with a sufficiently large margin to drive the required innovation.

As TV changes over time, the amount of differentiating features is not growing much, while the set of standard features is significantly increasing. Features of today, might become commodity tomorrow, and new features will pop-up. A platform should support this type of evolution.

3. STREAMING INFRASTRUCTURE

This section will explain the concept of a streaming infrastructure used in the Nexperia Home platform. It starts with a simple streaming model that assumes memory-based communication. Memory-based communication means that the data that needs to be processed is communicated between 2 entities via memory. We start with a model of a single CPU that implements the required signal processing as depicted in Figure 6-1. We extend this step by step until we reach a realistic model that resembles the reality in current Nexperia Home platforms.

3.1 Simple streaming model

As mentioned, the streaming in a digital TV consists of a number of processing steps. These steps are represented as streaming components that can be connected as nodes in a graph, see Figure 6-4. A streaming component contains an algorithm that performs the processing step and a non-specific part to support the in/output of data to/from the algorithm and to support the control of the component, like start and stop.

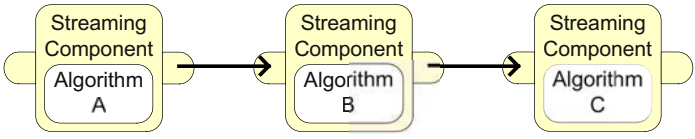


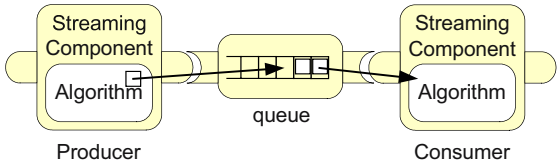
Figure 6-4. A graph of streaming components

A streaming component has input and/or output pins which represent the possible input and output data streams. Via these pins, components communicate the data to connected components.

A packet represents the unit of data communication in a memory based streaming model. A packet points to a piece of memory which is used to store the data that has to be transported from the producing component to the consuming component.

In the streaming model, two streaming components that are running on separate OS-tasks are connected via a queue that buffers data packets as

depicted in Figure 6-5. The producer puts the packets in the queue, and the consumer takes the packets out of the queue.



6-5. Two components connected via a queue

It is important to realize that the memory is the scarce resource, not the queue. The queue only contains packets that refer to the memory. In other words, the maximum size of the queue is not essential. Instead, the total number of packets representing the memory determines the maximum amount of buffering.

A packet pool creates the packets and allocates the required memory once during initialization of such a pool. At that moment, the number of packets in this pool and the size of the memory blocks they represent have to be specified. The memory blocks represented by packets in a pool are equally sized. This restriction prevents fragmentation within the packet pool and that is also the main reason why a packet pool is preferred above generic malloc/free functions. Such a pool is assigned to an output pin of a streaming components that uses packets of that pool as depicted in Figure 6-6.

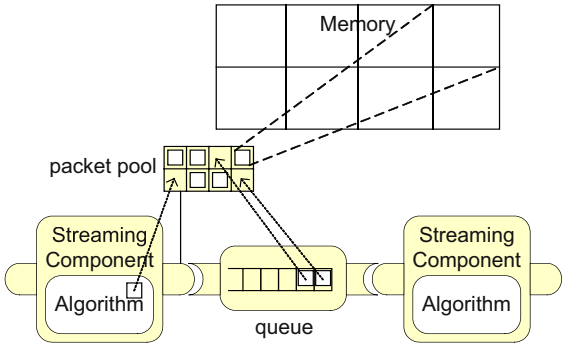


Figure 6-6. Packets are created by a packet pool and represent a block of memory.

The flow of a data packet is now complete as depicted in Figure 6-7. When the receiving component has consumed the data from a packet, the

packet is returned. One can see that the packet flow is a cycle through the queue and pool, i.e. the packets are reused. Essential in this cycle is the synchronization between the producing and consuming component. When a consumer tries to get a packet out of an empty queue, the consumer will block. As soon as the producer puts a packet in the queue, the consumer will be notified so that it can get the packet. This synchronization is based on the availability of packets in the queue (containing data) and on the availability of packets in the packet pool (empty).

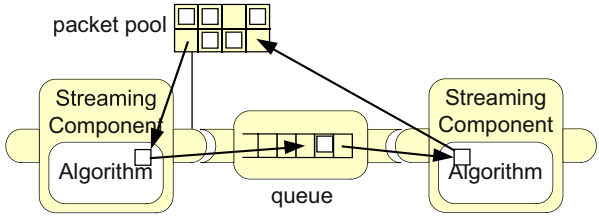


Figure 6-7. Packet flow through components, queue, and pool

An example of a streaming component is a video scaler. In Figure 6-8 the scaling algorithm is represented as a while loop that runs on a CPU as an OS-task. This component (1) receives packets representing video frames via its input pin, (2) gets an empty packet from the assigned packet pool to put the scaled video frame in, (3) scales the input to the output, (4) puts the new packet with the scaled video frame via the output pin to the next component, (5) and returns the input packet to the packet pool. This sequence will be repeated infinitely.

```
while (1)
{
  GetFullPacket(input,&inVideoPacket);(1)
  GetEmptyPacket(output,&outVideoPacket); (2)
  Scale(inVideoPacket,outVideoPacket); (3)
  ReturnPacket(inVideoPacket); (4)
  PutFullPacket(output,outVideoPacket); (5)
}
```

Figure 6-8. Scaling algorithm realized as an infinite loop.

3.2 Processing in HW

So far, the algorithm was running on a CPU. Some algorithms can run more efficiently on a specific HW, like an Application-Specific Integrated Circuit (ASIC), Field programmable Gate Array (FPGA), or Digital Signal Processor (DSP). A scaler is an example of an algorithm that can be realized much more efficiently on dedicated hardware due to the regularity of the algorithm. To incorporate HW processing, the streaming model is extended as depicted in Figure 6-9. The function (e.g. scaling) is divided into the SW streaming component and the dedicated hardware that executes the scaling algorithm.

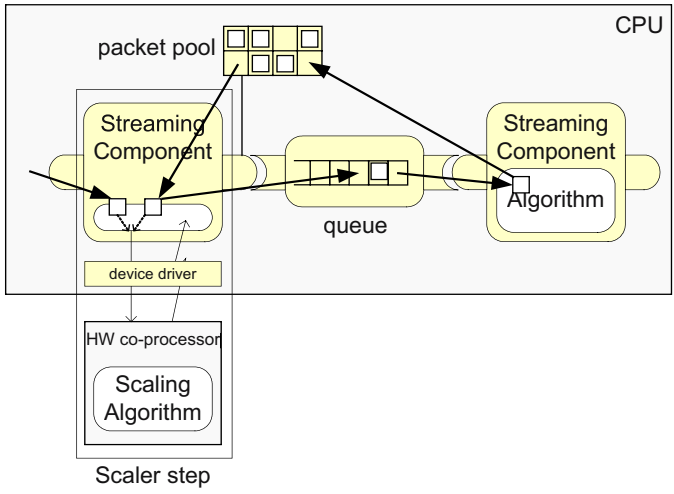


Figure 6-9. A scaler for which the processing is performed in HW

The SW streaming component is still responsible for synchronization, i.e. getting, putting and returning the packets. Instead of executing the algorithm itself (`Scale()` function in Figure 6-8), the software instructs the hardware to do the processing by setting registers of the HW. Information put in the registers is the physical location of the input and output data in memory, the physical format that is used to represent the frames, and the size of the input and output frames. After the scaler has been started, it directly reads the data from memory, processes it, and directly writes the data again to the specified location. Thus, the data transport (reading and writing) is directly to memory without the software (CPU) in between. Only the synchronization, i.e. passing packets, is handled by the software. The SW streaming component will block until the HW finishes. A common solution is that the HW triggers

an interrupt as soon as it is finished. The Interupt Service Routine (ISR) that handles the interrupt will unblock the SW streaming component. The SW streaming component then returns the input packet and forwards the output packet, as depicted in Figure 6-8.

By keeping the synchronization in software, the flexible packet management implemented in SW can be used and HW and SW processing components can be arbitrarily connected.

The physical representation of the data in memory depends on the HW co-processor. It can vary from one fixed place in memory, a fifo in memory with a start/end address and a wrap around, to a scatter-gather DMA that allows the data to be scattered in physical memory. The data representation in memory has impact on the queue and packet pool concept. In the Nexperia Home platform an input or output packet is typically a contiguous piece of memory.

3.3 HW synchronization

One can reduce the latency of a chain of streaming components by reducing the buffering in between components, which can be realized by decreasing the communication granularity. For example, an interlaced video frame can be communicated as 2 field packets or even as stripes. However, it increases the synchronization rate, which might be a problem if the synchronization is done on a CPU based on interrupts.

When two subsequent processing algorithms are realized in HW, as described above, the synchronization in between can be done by the CPU or directly by the HW. If the CPU does the synchronization, then the synchronization interrupts the execution (of other algorithms) on the CPU and generates overhead, especially since it pollutes the cache. To be able to reduce the latency and still effectively use the resources, both the synchronization and memory management for the data transport should be offloaded from the CPU. We will present two possibilities in the next paragraphs.

3.3.1 Memory based streaming

For memory based streaming in between two HW processors, memory is still used as intermediate storage in between the components. There are several options to realize the synchronization and management of the memory in HW: HW logic that knows the location of the queue administration in memory, to determine where the data has to be put/obtained, and that can determine whether there is space/data available. In Figure 6-10, such a logical fifo is depicted, which may use the C-heap

protocol, see Gangwal (2001). Another option is to use a small separate CPU for synchronization purposes only. The reasoning about what to choose is beyond the scope of this section.

A memory based HW processor has the advantage that it can still be connected to a SW streaming component, assuming that the synchronization and memory management logic is still accessible from software.

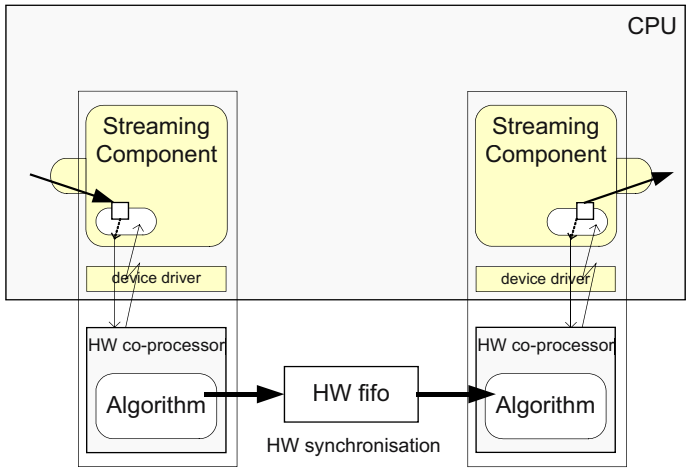


Figure 6-10. HW synchronization between two components

3.3.2 Direct streaming

Instead of using memory as intermediate storage for data transport, HW components can be directly connected via copper wire. This means that the HW components are very tightly coupled and that they have to execute synchronously. The streaming of an analogue TV, or the reception unit of a DVD recorder is done via copper wire connections, possible with some HW switches in between. In these cases, the tuner is (via an input-selection switch) directly connected to the PAL decoder (audio/video) using copper wires. More advanced peer to peer streaming is streaming via networks on chip, see Goossens(2003). Networks on chip use again memory for buffering, and as a consequence are less tightly coupled than the copper wire connected components.

4. DESIGN TRADEOFFS

In this section, we will discuss design tradeoffs for consumer products that impact the streaming processing architecture. To understand the tradeoffs, a number of forces that influence them are introduced.

The first force is power consumption of consumer devices, which is one of the differentiating points, as mentioned in Section 2. A reason is that a consumer does not like a PC-like blower in his TV making a lot of noise.

A second force is the amount of external memory in use, which should be as minimal as possible. The reason is that more memory increase the Bill Of Materials (BOM) costs. However, there are exceptions on this rule. For example, requiring 61 MB instead of 58 MB makes no difference, since 64 MB is a standard available memory sizes. So optimizing memory usage is important if the required amount of memory reaches a standard memory size. Another exception exists for small memories that fall outside the mainstream PC memories. These can be more expensive than larger PC memories.

The third force is the bottleneck in both memory bandwidth and latency. The trend of higher processing clock speeds makes this bottleneck even higher. External memory accesses are also power consuming. For these reasons, limiting the external bus traffic is therefore a clear force.

The fourth force is the continuous tendency to keep the chip area as small as possible to reduce the BOM as opposed to adding HW to reduce the software effort. HW support for certain features can directly be translated into chip area, and thus in costs and in the BOM. Not having HW support saves on the BOM, but may introduce additional software complexity. The additional software costs are less clear and much harder to predict.

The presented trade-offs made in a consumer device are discussed in more detail in the next sections. On different places in the platform, the tradeoffs are different, resulting in different choices. A streaming framework should facilitate this process by both providing enough flexibility and the right abstractions. It should allow different tradeoffs on different places in the graph, while still enabling the cooperation between the parts.

The flexibility versus the complexity of the streaming framework in itself is the final tradeoff.

4.1 Off-chip vs. on-chip memory

Memory used for transporting data in case of memory based streaming can be on-chip, or off-chip. The main force that drives the usage of on-chip memory is the access bottleneck of off chip memory. Due to the increasing

gap between high chip clocks and the external memory access times/bandwidths this force will only become stronger. The increase in bandwidth requirements for HD TV picture improvement algorithms makes this force even stronger.

On-chip memory has a number of advantages compared to off-chip memory. The bandwidth to memory on-chip is much higher, and the access latency to that memory is much smaller. In addition, it is more power efficient.

The disadvantage is that on-chip memory is relatively expensive. To make the expensive on-chip memory cost-effective, it has to be used efficiently. An additional disadvantage is the limited flexibility of on-chip memory, because the amount of it is fixed quite early in the design process. One way to use the on-chip memory efficiently is to use time-sharing, which means that a part of the memory is used by different streaming components over time. An example of course grain time-sharing is that one streaming component uses a piece of memory until the end-user switches to another mode of operation by means of a remote. A fine grain time-sharing example is that two streaming components use the same piece of memory after each other to process one frame. A disadvantage of time-sharing is that dependencies are created between parts that share the same memory. Another way to use on-chip memory cost-effectively is to keep the buffer in between two streaming components small. It is currently not cost-effective to buffer complete video frames on-chip (e.g. SD equals 800 KB, HD up to 4 MB). Little buffering means that two streaming components are tightly coupled, and that the communication granularity is small (e.g. stripes in stead of video frames).

The minimum amount of buffering between two components depends on the algorithms that are connected. As depicted in Figure 6-11, the smallest communication grain between a noise reduction algorithm and a scaler that go both from top to bottom through a video frame on line basis is a line. For an MPEG decoder producing macro-blocks, connected to a scaler consuming (complete) lines, the buffer has to be at least a stripe (row of macro-blocks). The required amount of buffering for this last example is even worse if I, P, and B frames are decoded since a MPEG decoder decodes these frames out of order.

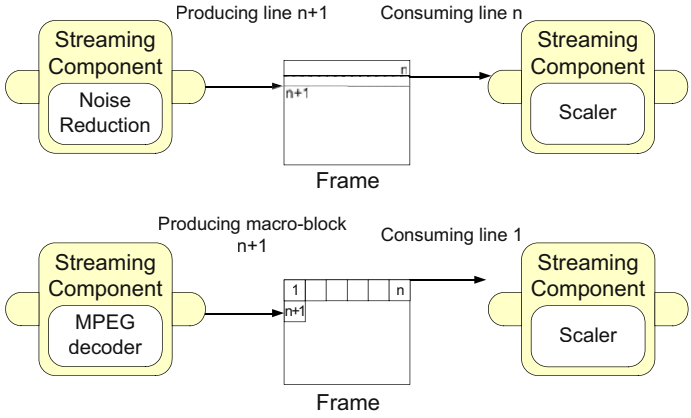


Figure 6-11. Minimum amount of buffering depends on producing/consuming behavior of connected streaming components.

In addition, for temporal algorithms that use older frames as reference (e.g. temporal noise reduction, MPEG decoder, natural motion), it is not cost-effective to keep the reference frames on-chip. The time between the generation of the reference frame and the accesses when used as reference is typically the period of producing a frame (e.g. 40 ms). It is too expensive to keep a complete frame on-chip for that long. Smaller grain communication requires higher synchronization rates that cannot always be realized in software efficiently.

To be able to make the trade-off between off-chip and on-chip, a streaming framework should be able to support both on- and off-chip buffering.

4.2 SW vs. HW processing

Processing algorithms can be implemented both in hardware and in software. Possibilities range from ASICs, FPGA, to weakly programmable HW, DSPs, and General CPUs. There is an ongoing trend towards (partly) implementing algorithms in software on a general CPU. There are different forces that cause this trend.

SW algorithms are in principle less hardware specific and can be ported from one platform to another. In practice, a substantial amount of effort is required for porting it to another HW platform. It is often underestimated how much effort is required to optimize the performance. However, the software flexibility allows changes/optimizations in a stage where the HW is already fixed. For example, when a standard is not yet completely fixed, a

SW implementation allows for making adaptations that are required to follow the latest trends, or for late time changes in the standard. Such an approach is valid in case the time-to-market is essential, typically for high-end products.

An additional advantage of software is that standards are often given as executable specifications that can run on a general CPU. Software allows for the transformation of an executable specification into an efficient implementation in a number of iterations.

Algorithms implemented in more dedicated HW are less flexible, but are much more efficient in processing regular and thus potentially parallel algorithms. Examples of such regular algorithms are line- or pixel-based algorithms, like a scaler, or peak filter. Coding standards, like H.264 or DivX, become more and more irregular to get maximum coding efficiency. Due to the irregularity, the possible parallelism of the algorithm reduces, which makes it less attractive to realize it in HW. As a result, the overall algorithm may be implemented in software, while some computing intensive parts can be offloaded to special hardware.

In the lifetime of an algorithm, one often sees transitions. For example an MPEG decoder was initially realized in HW since it was not yet feasible in software. Then, due to Moore's law that led to an increasing computation power, it could be realized in software. However, when the standard is matured and MPEG decoding becomes a default processing step, a HW decoder is preferred for efficiency reasons as the SW solution costs many processing cycles, which can be better spent for new, emerging standards. For a digital TV that supports dual screen, the MPEG decoding for the second half may still be realized in software due to the fact that only in a limited set of the use-cases it is required to have two decoders.

The HW/SW processing tradeoff requires a streaming framework that can abstract from whether an algorithm is implemented in hardware, software, or a mixture of both. To enable seamless HW/SW transitions, a streaming component should be connectable to both a HW and SW component, while the actual control from client perspective of the processing step should not see any difference.

4.3 HW/SW co-design

To give insight in the problem space of adding hardware to save software cost, we will use the example of HW support for virtual memory as an example to solve fragmentation due to dynamic use-case switches.

As explained in Section 3, a packet pool prevents fragmentation over time for a static graph where the pool parameters are fixed. When switching between use-cases, the packet pools have to be resized, e.g. the number of

packets changes and/or the sizes of the packets change. In case the amount of memory claimed for the pool is not based on the worst-case use-case, it is possible that a larger piece of memory has to be allocated. The holes that appear during this process might not be of sufficient size to capture the new data, while the total amount of free space is sufficient. Obvious solutions for this are the addition of extra memory or the use of defragmentation algorithms. However, as the consumer industry is highly cost driven and extra memory is not considered a end user benefit, the first solution is usually not applied in contrast to the PC industry. The latter one is also not an option since it requires additional copying of data, or it requires a certain time by slowly moving the non-used parts of the queues in memory without copying. These options are often not applicable due to timing constraints of use-case switches in consumer products. Product specific solutions exist. These solutions are not isolated and exploit system wide knowledge of a product. As a consequence, these solutions have to be revalidated and changed for every new product again.

A feasible solution is the use of virtual memory. For this, standard solutions exist that prevent that physical memory is wasted beyond page size boundaries. This adds to the BOM but enables a generic solution to support packet pools with no fragmentation when they are resized.

In the trajectory of HW/SW co-design, domain knowledge is essential to make the right choices in this large design space. In our example, it might be sufficient to have virtual memory only for video frames, as they impose the largest fragmentation. Video frames are very large in size, meaning that the pages of the virtual memory can be large and still effectively used, which in turn reduces the size of the lookup table to realize the virtual memory. In total, it reduces the cost of the additional BOM, making the virtual memory support a more attractive alternative.

5. REAL-TIME IMPACT

A graph of streaming components that receives real-time input data, and produces real-time audio and video on the output has strict real-time requirements. If these requirements are not met, distortions of the audio and video will be the result. In this section, we will show that real-time requirements make the streaming architecture more complex.

A large class of processing algorithms is data dependent, e.g. MPEG encoders and decoders. This results in fluctuations in load. In case an algorithm is implemented in software, it is typically too expensive to reserve the worst-case amount of CPU cycles. The maximum load for decoding one MPEG frame can be 2 times the average, hence the required reservation

would be 100% on top on the average required. One of the reasons is that the load for decoding I, P, and B frames differs. A standard approach is to spread load peaks by introducing additional buffering. For example, a typical requirement for Mpeg decoding is that one frame has to be decoded in 40 milliseconds (ms). By introducing additional buffering of 2 frames, the requirement reduces to 3 frames in 120 ms. The maximum load for decoding 3 MPEG frames might be in the order of 1.5 times the average since it is a mixture of I, B, and P frames. Now the required reservation is reduced to 50% on top of the required average.

Increasing the buffering in a chain increases the end-to-end latency. There are a number of reasons why end-to-end latency is important. The first one is that it influences the response-time on user events like zapping. As soon as the user zaps, the tuner is changed. The end-to-end latency determines the minimum response time before the new channel is shown on the display. However, zapping response times of 500 ms are acceptable and that is not that restrictive on the end-to-end latency on the streaming graph. Another, much more stringent, constraint on the latency is imposed by lip-sync requirements. The end-to-end latency of the audio should be about equal to that one of the video. In case the audio rendering is done by the same device as the video, like in a set-top box, the audio can simply be delayed. However, in case that the audio of a DVD player goes directly to an amplifier and the video via a TV, the end-to-end latency of the video processing in the TV must be at most 35 ms. Let's calculate what it means to have a video capturer, 3 processing steps (noise reduction, de-interlacing, up-conversion), and a video renderer in case the communication granularity equals video frames. For simplicity, we assume that each process step takes only 10 ms. It takes 40 ms before a complete frame has entered the system. That cannot be improved; the bits of a frame are evenly distributed in 40 ms (25 frames/s). It will result in a 70 ms latency, as depicted in figure 6-12(a). In the latency calculation, we do not have to include the completion of the rendering for the whole frame. A pixel that enters the system at time 0, will leave the system at time 70 (first pixel of the frame will be rendered first).

In this example a processing step is only performed in 10 out of 40 ms, i.e. 25% of the time. If for each step special HW is used, it is only 25% effectively used.

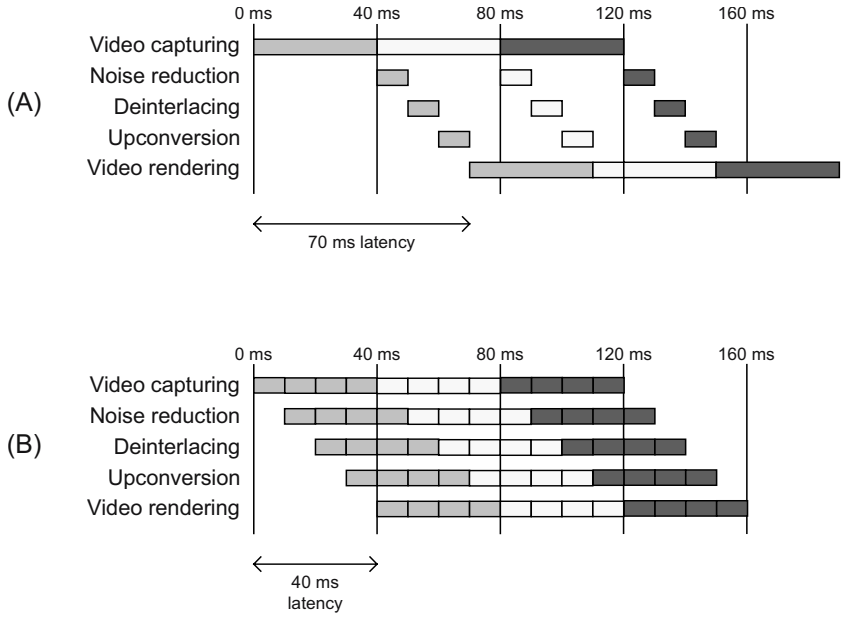


Figure 6-12. End-to-end latency calculations

One way to decrease the latency is to shorten the processing time of the three steps. It is clear in this example that we will never achieve a 40 ms end-to-end latency by increasing the processor power. Furthermore, the effectiveness of dedicated HW will decrease even further.

Another way to reduce the latency is to decrease the communication granularity. Lets assume we can reduce the granularity to ¼ frame and assume that processing of each ¼ frame takes 10 ms. The resulting end-to-end latency reduces to 40 ms, as depicted in Figure 6-12(b). Note that by assuming 10 ms per ¼ frame, the resources for each processing step are 100% used and cannot run on a shared resource.

Let's look at the limitations of the fine grain communication approach:

- As mentioned in Section 3, the possibility of reducing the communication granularity depends on the algorithm. For example the de-interlacing has an intrinsic delay of half a frame. The progressive frame can only be computed when receiving the second field. As a consequence, in the ¼ frame approach, the end-to-end delay will increase with 20 ms.
- The expected latency reduction to 40 ms as shown in Figure 6-12(b) will typically not be achieved. One cannot deduce that processing of ¼ frame is finished in 10 ms for each processing step from the knowledge that 1 frame is finished in 40 ms. When load variations

occur, the worst-case time for processing ¼ frame will take more than ¼ of processing a whole frame. These variations can be dealt by introducing additional processing power to finish within 10 ms, or by introducing additional buffering meaning that the latency will increase. So without increasing the processing power, the total latency will be more than 40 ms.

- Reducing the communication granularity even further, e.g. to 1/8 frame, reduces the latency again but increases communication overhead, inherent limitations of algorithms, and strict timing requirements become more and more dominant.

The tight coupling of processing steps and strict timing requirements restrict the possibility to spread out peaks in CPU load and busload. It will result in a more bursty behavior. The composition of parallel processing steps that have bursty behavior on shared system resources (e.g. bus) will start interfering. This interference will result in unpredictable behavior, especially when the load of the shared resources is high.

The BOM for consumer products drives that HW resources are used as effectively as possible. What we have shown in this section is that real-time constraints complicate the streaming architecture and negatively impact the compositionality of the system. Compositionality is essential to efficiently make a range of different products. Without a composable architecture it costs too much effort to create the products. It is a balance between being able to make a product at all and being able to sell it for a reasonable price.

Real-time constraints do not make the streaming infrastructure more complex. However, they do make tradeoffs in the streaming architecture more complex, like choosing HW/SW synchronization and the right communication granularity. Most-important, real-time constraints make the tradeoff between cost-effectiveness and the compositionality of the streaming platform much more complex. Both are essential in a consumer product.

6. CONTROLLING THE STREAMING GRAPH

A streaming component has to be controlled. The first half of this section will explain what streaming control is about. The second half will show the importance of being able to compose the streaming control, and show different approaches for structuring streaming control.

6.1 Streaming control

Besides starting, stopping, pausing, and connecting components, streaming control is the configuration and management of streaming components in a graph. Streaming control is an essential part of the streaming software architecture. The part in the streaming platform that is responsible for controlling the streaming components is denoted as manager, see Figure 6-13.

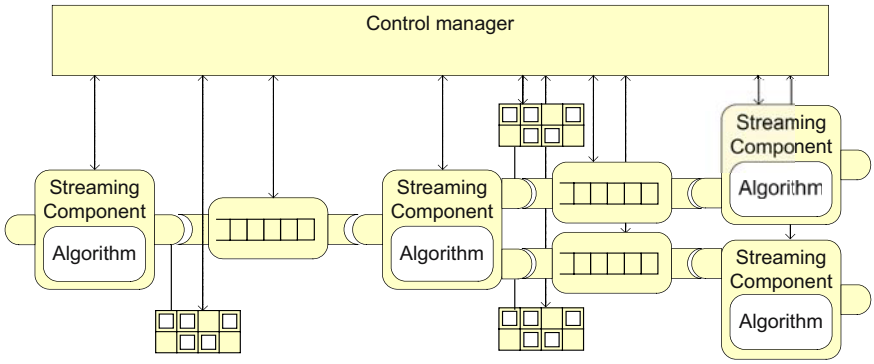


Figure 6-13.Control manager: controlling all entities of the streaming graph

Typically, the execution of the algorithm is decoupled from the streaming control. In case the algorithm is implemented in software, it runs on another OS-task. In case it is implemented in hardware, it runs in parallel on different HW. When a control command is given to the component, e.g. a new size for the scaler, it is stored in the streaming component (or in a register of the HW implementation). The implementation of the algorithm decides when to take this into account. There are some limitations when a control command can be taken into account by an algorithm:

- Often parameters of an algorithm cannot be changed halfway through the execution, e.g. it is no use to change the size when the scaler has already scaled half of the frame in the old size. This means that the response time of new setting is either dependent on when the setting is given or it should be given at a specific point in time.
- Being able to force parameters upon the algorithm means that the execution of the algorithm becomes dependent on the behavior of the streaming control. As a consequence, the real-time requirements of the streaming task are imposed on the streaming control tasks. Such dependencies make the execution architecture of the whole much more complex and should be prevented.

In addition to the control initiated from the control manager, the components themselves also give feedback towards the manager. These are typically callbacks or events on which the streaming control can subscribe, e.g. progress events and error events. The handling of these events should be done on other tasks than the streaming component tasks to prevent that real-time requirements of the streaming components are forced upon the manager.

6.2 Control compositionality

In a streaming platform, the separate streaming components have to be combined to a properly cooperating graph of components. Besides the data streaming in the platform, the control of the streaming components is required to get a streaming platform.

The control of a streaming graph includes connecting, configuring, and starting/stopping components, which is relatively simple. It is more complex to react on external changes, like a signal drop, Standard Definition (SD) to High Definition (HD) changes of the signal, or resolution changes in the input signal. These changes are detected by one of the streaming components, and should result in new configurations of streaming components, replacement of streaming components, re-assignment of resources, etc.

Which components are affected by an external change depends on the state of the system. The state of the system is the combination of the type of input, the states of the streaming components, and how they are connected. Examples of component states are: running, paused, muted, and resolution of a component. States may be relevant from functional point of view, or from resource usage point of view. Examples of connection states are: is an input going to the main screen, picture in picture, or SCART output, is a natural motion component used for the main picture or for the sub-picture. Due to the large different number of input types, streaming graphs and streaming components there is a large number of relevant states. It already requires substantial effort to build such a manager and this will only increase due to the increasing number of streaming components and combinations thereof.

Besides reducing the effort of building a control manager once, it is also essential to reduce the effort needed when a control manager evolves with the platform. This effort strongly depends on how the control manager is structured. We present three approaches to structure the control manager.

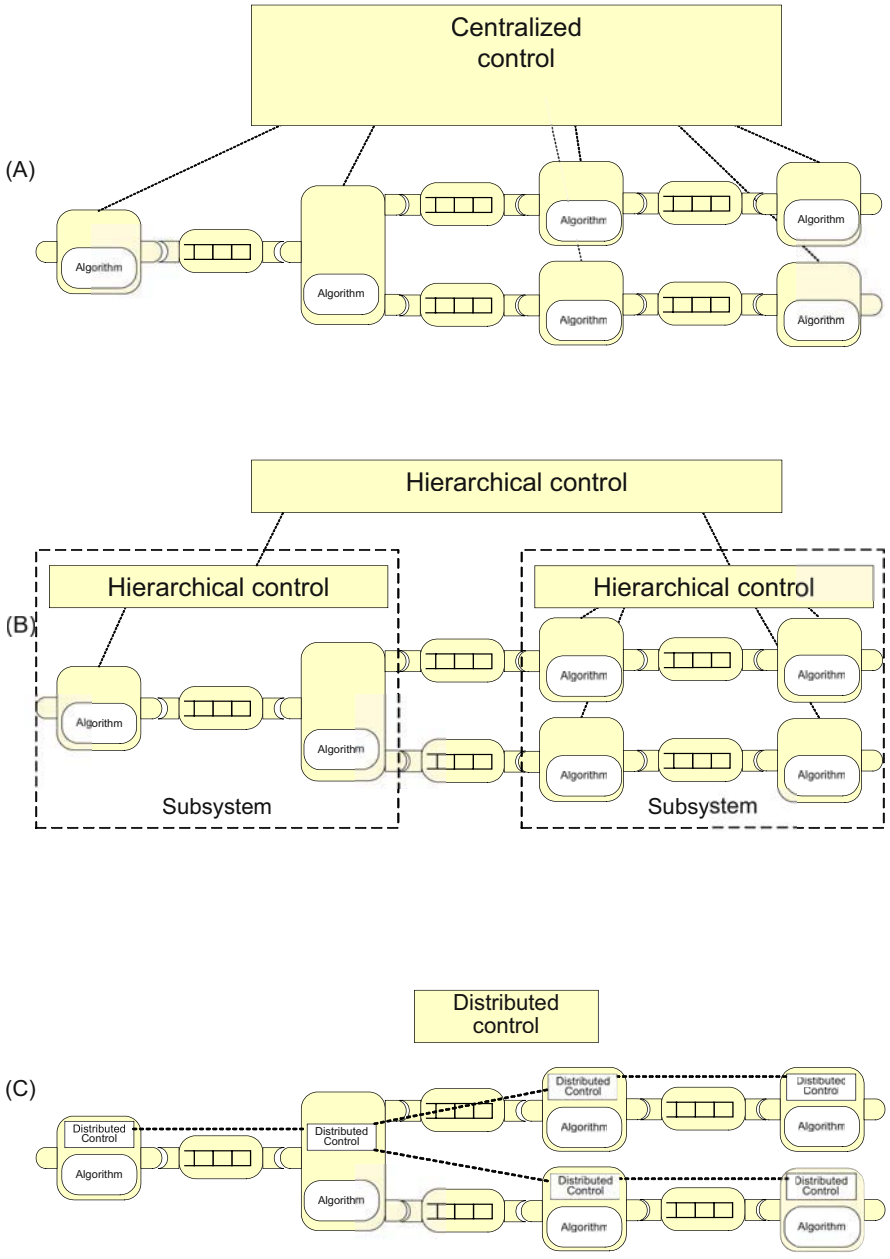


Figure 6-14. Centralized (A) vs. Hierarchical (B) vs. Distributed (C) streaming control

One approach is to centralize the realization of the control manager, see Figure 6-14(a). This spider 'knows' the state of the system, and has rules for

how to react on an event in each state. Such a manager will be large due to the large number of states. More important, it is dependent on the components in the platform since these determine the possible states in the system, and thus the states in this centralized control.

A second approach is that the control manager is hierarchically decomposed in subsystems, as depicted in Figure 6-14(b). Within a subsystem the required control is handled as much as possible. Only when interaction is required with another subsystem, it is handled via the central manager. The advantage is that the control of a subsystem will not change when components in another subsystem change. Also the central manager will only change when changes inside a subsystem cannot be dealt with within that subsystem.

A third approach is to distribute the control manager, see Figure 6-14(c). This approach is also known as horizontal communication, see Ommering (2003). Most of the knowledge on how to react on external events is captured in local rules, i.e. local to a streaming component. In addition to those local rules, mechanisms are required to distribute information over the graph. Typically, this information exchange is done via the connections.

The major benefit of the distributed realization is that the centralized streaming control has become minimal or disappears completely. This is optimal in terms of evolution of the control together with the evolution of the streaming components themselves. When a streaming component changes, also the local streaming control changes.

7. SUMMARY

A streaming platform is introduced in consumer products to enable independent middleware development on top of a standardized API. The realization of such a platform must be sufficiently flexible to incorporate customization and to allow for new requirements to support external influences like the realization of emerging new decoding standards, evolution of the platform over time, and applicability for a range of products. The Nexperia Home Platform is the streaming platform of Philips.

A streaming infrastructure is an essential part of a streaming platform. However, it is only a small part in terms of effort. In fact, the actual passing of the data and the required synchronization are the least concerns of the platform supplier.

The challenge of a streaming platform is to balance between a small BOM, a configurable streaming platform, and to allow evolution:

- To reduce the BOM, much effort is put in making HW/SW tradeoffs and in optimizing of SW processing algorithms. Both hardware and software

are essential in a streaming platform and are applied for different reasons at different places. The platform API should be such that these tradeoffs can be made without impact on the middleware. Philips has defined such a HW independent standard, called Universal Home API. The streaming infrastructure must give the flexibility to make such choices.

- Configuring a streaming platform still requires substantial effort. Configuring varies from setting some parameters to composing a platform from the building blocks, like streaming components.
- Evolution of a platform requires that parts of a platform can be removed, added, or replaced. For a smooth evolution, the platform should also be composable. One has to be careful not to make a platform too generic, otherwise it will lead to a non cost-effective platform. One should choose very carefully where to put the flexibility and where to fix the functionality.

REFERENCES

Ommering, R. v., 2003, Horizontal communication: a style to compose control software, Software- Practice & Experience Volume 33, issue 12

Gangwal, O. P., Nieuwland, A., and Lippens, P, 2001, A scalable and flexible data syncrhonisation scheme for embedded HW-SW shared memory systems, Proceedings of the International Symposium on System Synthesis, pp 1-6

Goossens, K., Dielissen, J., Meerbergen, J. van, Poplavko, P., Rădulescu, A., Rijpkema, E., Waterlander, E., and Wielage, P., 2003, Guaranteeing The Quality of Services in Networks on Chip, Networks on Chip, Kluwer, pp 61-82

Chapter 7

A ROBUST COMPONENT MODEL FOR CONSUMER ELECTRONIC PRODUCTS

Hugh Maaskant
Philips Research Laboratories, Eindhoven, The Netherlands

Abstract: The Robocop project defines an open, component-based architecture for the middleware layer in high-volume consumer electronic products. This architecture supports component trading, dynamic upgrading and extension of products in the field, and robust and reliable operation. The architecture consists of a development framework, an execution framework and optional download and resource management frameworks. The core of the architecture is the component model, which is defined at two levels. At the development level, a component is defined to be a collection of models and the relations between these models. These models allow system builders to reason a-priori about systems composed from the components. At the execution level a binary component model has been defined, combining elements of Object Management Group (OMG), Common Object Request Broker Architecture (CORBA), Microsoft's Component Object Model (COM), and Philips' Koala. Key elements of the executable component model are explicit dependencies, dynamic third party binding, and a well-defined lifecycle that includes explicit interaction points with the resource management framework.

Key words: architecture; software component; component-based development; software product families; robustness; software download; resource management.

1. INTRODUCTION

In today's consumer electronic products a large part of the functionality is realized in software. This software typically includes both control and media processing (a.k.a. streaming) functionality. As software is notoriously difficult and expensive to develop, manufacturers constantly search for more effective ways to construct the software for their products. One way is to construct the software for a family of products rather than for a single

P. van der Stok (ed.),
Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, 167-192.

product. A promising approach to software product family engineering is to compose (assemble) products from parameterized software components. A *software component* is a unit of deployment that can be reused in multiple products (i.e. instances of the product family). The functional diversity in the product family can be achieved by a combination of techniques, such as composing different components, composing the components differently, and assigning values to the parameters of the components during composition.

One of the many challenges in component-based development (CBD) is to ensure the robustness of the final products. Not only must the independently developed components fit together functionally, they also must fit with each other and the underlying platform in *extra functional* properties such as timeliness and resource usage. In current practice, these concerns do not align well with the primarily functional decomposition that typically drives the specification for the components to be developed.

Two interrelated concepts help to meet these challenges: component models and product line architectures. A *component model* defines what constitutes a component, including the interaction mechanisms between components themselves and between components and their environment. A *product line architecture* prescribes, among other things, how components can be composed into products, how diversity is handled, and what major design rules and interaction patterns the components must adhere to. Figure 7-1 illustrates that a product family architecture may include a component model, and that it addresses more concerns while having a smaller scope of applicability than a component model.

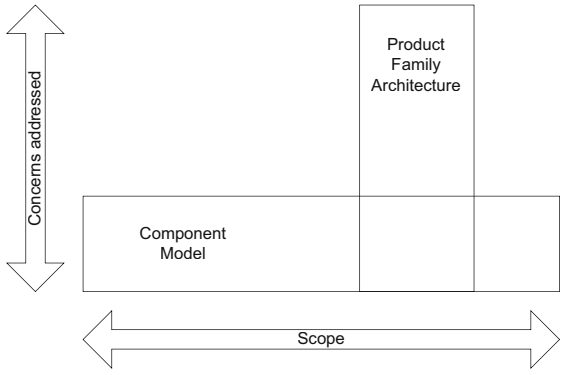


Figure 7-1. The relation between Product Family Architecture and Component Model

Thus a component model enables a product family architecture, mostly by addressing the mechanical parts of what constitutes a component and by providing facilities for diversity management.

Building a product as the composition of components that comply with an open component model lowers the technical barriers to the use of 3rd party components. Generic, i.e. non-discriminatory, software components can be developed by, and purchased from, specialists that serve a larger market and therefore can provide the components at a lower cost. However, compliance with the component model does not necessarily imply compliance with the full product line architecture. Some adaptations are still likely to be required, typically realized through glue code and wrapping. Depending on the richness of the component model, this can e.g. include thread decoupling in certain interfaces such as event call-backs. Adherence to the component model does at least ensure compatibility on a basic level of operation.

Finally, if the components can be composed at run-time rather than at design time, the appliance can be updated or upgraded in the field. This enables new business models where parts of the software for embedded devices can be sold separately.

1.1 The ITEA Robocop and Space4U Projects

The ITEA[1] project Robocop (Robocop 2003) was a two-year, multinational, multiple company research project with the aim to:

> Define an open component-based framework[2] for the middleware layer in high-volume embedded devices that enables robust and reliable operation, upgrading and extension, and component trading.

The systems targeted by Robocop are consumer products such as mobile phones, set-top boxes, digital TVs and network gateways. The project ran from July 2001 through June 2003 and was executed by a combination of larger (Fagor, Nokia, Philips) and smaller (SAIA-Burgess, Visual Tools) companies, research institutes (CSEM, ESI, Ikerlan) and technical universities (Eindhoven, Madrid).

The core of the Robocop architecture is its component model. Other key elements are the download and the resource management frameworks. A reference implementation of the component model and the frameworks has

[1] Information Technology for European Advancement, an eight-year strategic pan-European programme for advanced pre-competitive research and development in embedded and distributed software.

[2] A framework is defined as a partial architecture. The word is also used to denote an implementation of the partial architecture.

been built on Linux. Furthermore, the component model and frameworks have been ported to various architectures, some of which proprietary, for company specific demonstrators.

The Robocop project did not address all relevant issues. Therefore a follow-up project, Space4U (Space4U 2004), was defined to extend the architecture with frameworks for fault management, power management, and terminal management. Also, the component model was slightly modified (mostly simplified) based on newer insights and the experiences in building the demonstrators. At the time of this writing (mid 2004), the Space4U project is halfway through its two-year timeline.

This chapter introduces the Robocop architecture in Section 2. It presents a simplified view on the Space4U version of the component model in Section 3. Section 4 contains a very brief sketch of how the framework and component model can be applied to a streaming system.

2. THE ROBOCOP ARCHITECTURE

The Robocop architecture can be considered to be an infrastructure to design, build, trade and update middleware by means of components. The architecture is defined as a number of interrelated frameworks, where each framework addresses certain aspects.

The core frameworks are the development framework and the run-time (a.k.a. execution) framework. The *development framework* defines a number of aspects relevant for the development and trading of components. The *run-time framework* defines the execution environment for these components and defines certain aspects of their dynamic behavior.

The development and run-time frameworks are supplemented with the optional download and resource management framework. The *download framework* facilitates the controlled and secure downloading of components from a repository to a device. The *resource management framework* provides mechanisms for applications and components to negotiate their resource (CPU, bandwidth etc.) needs, and to obtain guaranteed budgets for these. As may be expected, the parts of the download framework that exist on the target device are themselves Robocop components. Apart from a few small parts interacting with the operating system scheduler and the network drivers, the resource management framework is also realized as a Robocop component.

Robocop defines components at two levels, the development and the executable level. These levels correspond directly with the development and execution frameworks.

At the development level, a component is a collection of models and the relations between these models. Each of the models addresses certain aspects of the component. Components are the unit of trading, whereas models are the unit of deployment[3].

One of these models is the *executable component*, which embodies the component as it can be executed on a given platform. The *executable component model* defines the concepts, protocols, and interfaces at an abstract level and specifies language and binary mappings for the supported platforms. The executable component model closely resembles traditional binary component models like Microsoft's Component Object Model (COM), and the Common Object Request Broker Architecture (CORBA) of the Object Management Group (OMG).

The word component is strongly overloaded, not only in the literature but also in Robocop. For briefness, an "executable component" is often referred to as "component", and the "executable component model" is often just called "component model". Sometimes the component at the development level is called "component package" to make the intention explicit.

The component models are discussed in detail in Section 3, The Robocop Component Model.

2.1 Development Framework

The development framework consists of a number of elements such as the identification of the stakeholder roles (e.g. component vendor, system integrator) and their relations, the component model, and tooling, such as an IDL compiler. For the purpose of this chapter, figure 7-2 illustrates the development framework as a process flow.

Based on the domain requirements for the products targeted, components are developed on *host* computer systems and published in one or more *repositories*. Components can be developed either internally or externally to system developers, i.e. organizations that develop products. How components are developed (methodology, tools, languages, etc.) is outside the scope of Robocop.

Published components may be "generic" in the sense that they still need to be tailored to run on a specific *platform* (defined as the combination of device hardware and operating system), or they may need to be adapted for any other technical or commercial reason. This may involve compilation and linking for the specific platform or the removal of certain models. Depending on business considerations, tailoring may be done by the original

[3] Selecting a subset of the models (and/or relations) from a component yields a new component. Therefore the component is merely a packaging for models.

component developer, by the repository caretaker, by a trusted 3rd party, or by the system developer.

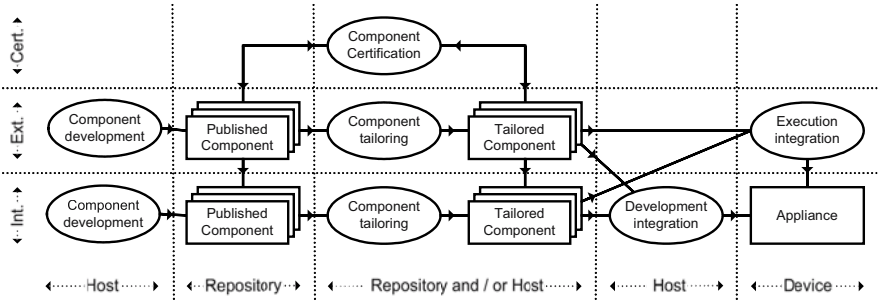


Figure 7-2. Robocop development framework – process flow view

Optionally, a trusted third party may certify that components comply with the Robocop architecture or any other standard. In principle, this can be done at any point in the development life cycle of a component.

The system developer assembles the product by integrating the tailored components with the platform and by adding applications; this process is called *development integration*. The component models can be used to support the integration process, e.g. by gaining better insight in the functioning of a certain component or by predicting certain properties of the composition by composing the relevant models as detailed in Section 3.1.1.

During the operational life of the product, components may be replaced for improved functioning (*upgrade*) and new components may be added for extra capabilities (*extension*). This is called *execution time integration*; it will typically be preceded by downloading the component to the device.

2.2 Run-time Framework

The run-time framework defines a partial architecture for a further unspecified, single device. The architecture is partial in the sense that it only defines the component model for the middleware layer and the run-time environment. It does not address any of the functional aspects of the components. Figure 7-3 shows the scope of the run-time framework.

The devices, the applications running on these devices and the final products are left unspecified. However, the devices are assumed to have limited resources (CPU, memory, bandwidth, power, etc.), and the applications are assumed to have real-time requirements. The product may

be connected to the outside world through some form of network and can be mobile or stationary.

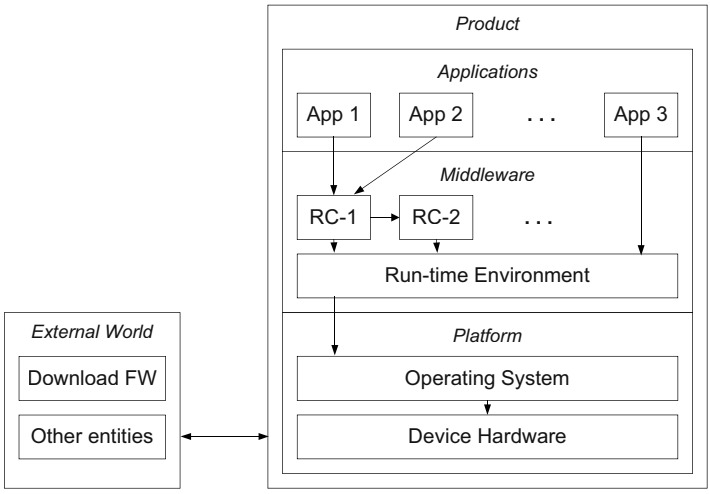


Figure 7-3. Runtime framework architecture

The product is modeled in three layers: the application layer, the middleware layer and the platform layer. The application and platform layers are outside the scope of Robocop. The *application layer* provides the end-user functionality of the product, which may include the user interface. The *middleware* layer is loosely defined to be everything above the operating system that is not an application. Typically, the middleware presents a programmatic interface to the applications that is expressed in terms of the product's domain model. It extends the underlying platform with domain functions and shields its idiosyncrasies. As can be seen in figure 7-3, the middleware is composed of executable components and a *Robocop Run-time Environment* (RRE), which is primarily the embodiment of the creational aspects of the executable component model[4] on the platform. The platform layer consists of the device hardware and an operating system layer. Very few assumptions are made on the operating system so that many different platforms can be supported.

[4] The RRE may also include the optional Operating System Abstraction Layer (OSAL) to facilitate portability of components and some other elements, but these are outside the scope of this chapter.

The terms application and middleware can be somewhat misleading as they can be interpreted in multiple ways. Typically, these terms refer to the "end-user functionality" and the "domain model" roles in the product's architecture. But in the Robocop definition, middleware consists of entities that strictly conform to the component model (plus the RRE), while applications are entities that, although they have all the *rights* that components have, i.e. they can interact with components and the RRE, they only have some of the *responsibilities*. In practice, and by design, these two views map well with each other. Still, it is entirely feasible that an entity fulfilling an application role is implemented as a Robocop component. It is even possible, but strongly discouraged, that some piece of software fulfilling a middleware role does not itself obey the component model.

2.3 The Download Framework

The download framework enables the dynamic upgrade and extension of products by downloading new components onto the product. The framework defines two entity types, repository and target, and three roles, initiator, locator, and decider. The initiator, locator, and decider roles can each either be implemented on the repository, on the target or on a separate host computer; not all permutations are meaningful, however.

A *repository* stores components on a host computer for future download to a target. A repository may also support searching or browsing operations; this is not further detailed in Robocop. The download framework supports multiple repositories. A *target* is a device to which the component will be downloaded. In the framework, each target device is individually identifiable.

The *initiator* is responsible for recognizing the need for a download transfer and for the initiation of that transfer. Various conditions may trigger a download; for example a digital TV service provider may update all digital set-top boxes of its subscribers to correct a faulty component when a correct version becomes available. Likewise, a DVD player may initiate the download of new components in order to be able to play a disk with a new data format. The *locator* serves as the well-known entry point that all parties must know about and with which they must register. It maintains the mapping between identities and network addresses of these parties, and is capable of determining which repositories contain a given component and which decider must be used for a given download. The *decider* performs a feasibility analysis before the download can take place. In this analysis, any combination of technical, resource, legal, and business criteria can be taken into account.

To determine the technical fit, both the target and the component are characterized by a profile. The *profile* contains various execution compatibility attributes, such as the instruction set architecture, available or required resources, the operating system, etcetera. The profile also contains attributes that are relevant for the transfer itself, such as the supported transfer file formats and protocols. The framework enhances robustness by not downloading a component unless it fits.

2.4 The Resource Management Framework

The goal of the resource management framework is to ensure that in an overloaded device the most important jobs still receive sufficient resources to allow them to perform their function with an acceptable quality of service. When multiple applications using multiple components are concurrently active, resource overloads may occur. This is especially true when different parties, not aware of each other, develop the applications or components. Note that in a dynamically upgradeable system, there may be no design time validation of the total set of applications in the product. The resources currently managed by the framework are CPU cycles and network bandwidth. Other resources can be added to the framework when desired.

The resource management framework manages the guaranteed availability of resource shares for resource consuming entities, here called consumers. A *consumer* is a set of threads that share a common application purpose. The concept of consumer is orthogonal to that of applications and components. Note, however, that part of the logic associated with resource management must be provided by the applications and components themselves, as only they have the required knowledge. The other, generic, part is implemented within the framework.

The framework recognizes two classes of consumers: non-resource-aware and resource-aware. A *resource-aware consumer* knows its resource needs for the various resources that are being managed. A *non-resource-aware consumer* does not know its resource needs and hence cannot be given any guarantees.

A special case of a resource-aware consumer is a quality aware consumer. *Quality-aware consumers* are resource-aware consumers that can provide a number of quality levels, and that can change their level dynamically. With each quality level a different resource need is associated. Quality-aware consumers provide a higher level of flexibility than ordinary resource-aware consumers because they have more options than simply ‘on’ or ‘off’. A software MPEG decoder component that can run in three modes (I frame only, IB frame or IBP frame decoding) and knows its resource needs at each level, is a good example of a quality aware component. Each

successive decoding level requires more CPU cycles per unit time, but gives a better image quality.

The resource management framework provides mechanisms through which resource-aware consumers can register their resource needs with the framework. The framework either admits the consumer, which implies guaranteed availability of the required resources, or, if this is not possible, it denies the request. Furthermore, the framework provides mechanisms for quality-aware consumers to register their quality levels and the respective resource needs. The framework optimizes the overall system quality by selecting the maximum quality level for each consumer, so that the sum total of all resource needs fit within the capabilities of the device. In this determination the framework takes the relative importance of the consumers into account. Changes in resource loads, e.g. due to mode changes that involve starting or stopping consumers, may lead to a renegotiated quality setting for the active quality-aware consumers. A system level configurable slack budget is allocated for non-resource-aware consumers.

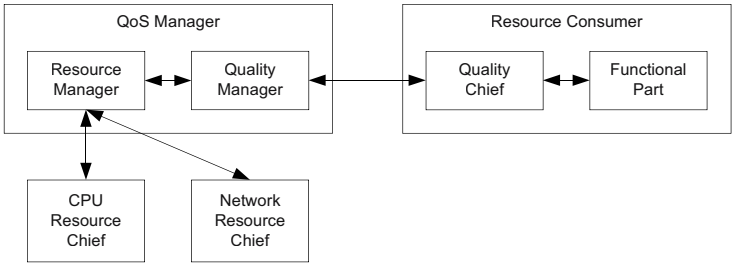


Figure 7-4. Resource Management Framework

The framework consists of a QoS manager, quality chiefs, and resource chiefs (see figure 7-4). The *QoS manager* is the central piece of the framework. Internally it consists of a quality manager and a resource manager. The *quality manager* communicates with the consumers to negotiate the quality level, and thus the resource usage. The *resource manager* aggregates the various resource types and communicates with the resources to set the budgets. The *quality chief* is a mandatory part of all quality aware consumers. It ensures that the functional part operates according to the negotiated quality level at any point in time. A *resource chief* manages a given resource type, e.g. CPU cycles. It measures the resource consumption of all consumers and enforces adherence to their budgets. The framework provides an implementation for the QoS manager

as well as resource chiefs for CPU and network bandwidth. The quality chiefs must be provided by the applications and components making up the system.

The framework is compositional. Multiple quality aware consumers can be combined into a higher-level quality aware consumer. This entails an entity that on one hand acts as a quality chief to the quality manager and on the other hand acts as the quality manager to the various quality aware consumers being combined (see figure 7-5). It maps the various combinations of the individual quality levels to a meaningful set of quality levels to present to the QoS manager.

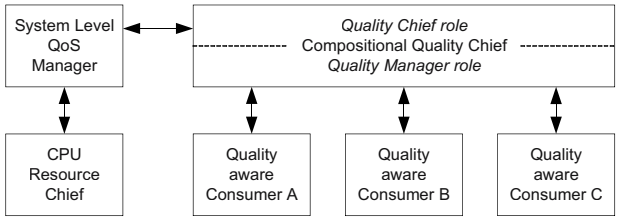


Figure 7-5. Compositional Quality Chief

For example, the software MPEG (Moving Picture Experts Group) decoder component mentioned above may in the streaming graph be followed by a quality aware image improvement component. Let's assume that the image improvement component can operate in three modes: off, medium, and full. This gives a total of nine settings, not all of which will be meaningful. A simple quality-aware application can provide a selection of these nine combinations by aggregating the quality levels and associated resource usage. In another example consider the same system with Picture in Picture (PIP) capability. If both the main window and the PIP window are generated by two independent instances of the streaming graph described above, it makes no sense to fully optimize the PIP window (and not optimize the main window). By placing this knowledge in an overarching application that again manages the individual pipelines' quality level, the pipelines themselves are context free, i.e. they do not have to be aware of PIP at all. This isolation is exactly what is needed to support software product families using a compositional paradigm.

2.5 Identification

In the Robocop architecture, virtually all artifacts are identified through a Globally Unique Identifier, known as a GUID[5]. A GUID can be generated with a virtual guarantee of being unique over space and time without relying on a central authority and with a fixed and reasonable size.

3. THE ROBOCOP COMPONENT MODEL

3.1 The Development Component Model

In the development framework, a component is defined as a collection of models and relations between these models (see figure 7-6). The set of models is open; models can be formal or informal, human and/or machine-readable. Anything that conveys some information about aspects of that component is considered a model. One of the defined model types is the executable model, also known as the executable component. The executable component is the binary representation of the component on the target architecture. There may be multiple executable models in a component, e.g. for different architectures: ARM or MIPS, or for different purposes: debug or production. The Robocop IDL (interface definition language), which will be explained in Section 3.2.4, is another model type. So far two types of relations between models have been defined: complies and implements.

[5] Also called UUID: Universally Unique Identifier.

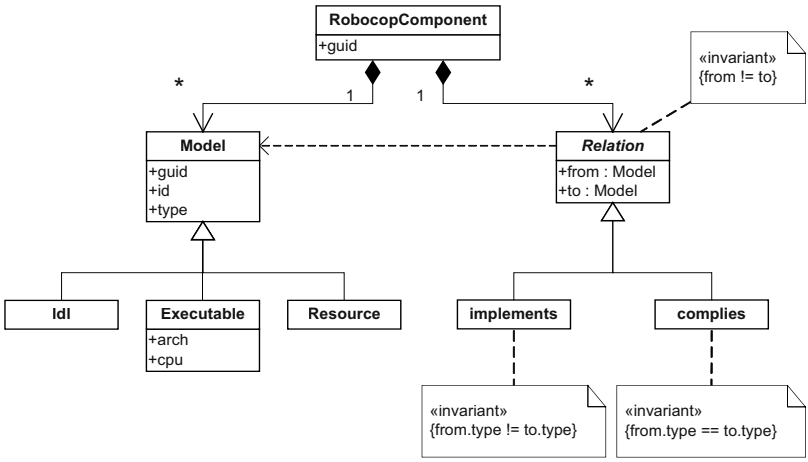


Figure 7-6. Robocop development component model

When a model "A" *complies* with a model "B" this means that "A" and "B" are of the same model type, e.g. both a resource model, and that all properties derived from "B" also hold for "A". Lets assume a particular type of resource model that consists of a single number denoting the maximum size (in bytes) of the data segment in an executable model of the component. An initial, rough, estimate may be given for this "DataSize" model as, e.g. 2048. Later, based on a real implementation for e.g. a MIPS processor, this may be refined to a new model "DataSize_MIPS" whose value is, say, 1536. The latter, more specific, model complies with the first one.

The implements relation is defined between two models of different types. Model "A" *implements* model "B" means that model "A" meets all properties specified by model "B". Typically the implementing model will be of the executable component type, and the implemented model type will be some attribute. In the above example, the executable component "Executable_MIPS" implements resource model "DataSize_MIPS". Note that by virtue of the complies relation between the resource models "Executable_MIPS" also implements the "DataSize" resource model (see figure 7-7). There is no need to explicitly specify this derived relation in the component package.

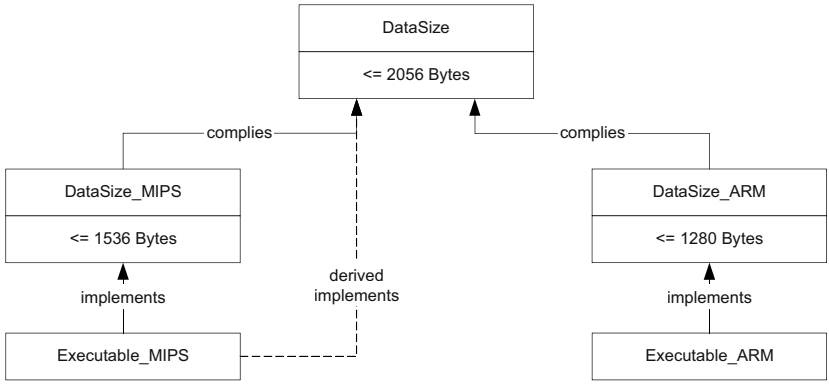


Figure 7-7. Example model relations

To fully exploit the power of these and any other foreseeable relation types, e.g. *refines* or *replaces*, further investigation is needed.

3.1.1 Use of Models

Specifying a component as a set of models is a key innovation in Robocop. There are various usages for this concept, including trading, composition support, and execution-time inspection of properties of components.

The first major reason for defining a component as a collection of related models was to support trading. During the conception and design of a product or product family, many kinds of information are needed for make or buy decisions of envisioned software blocks. Models allow vendors of components to make this information available under various commercial and licensing conditions. An example of this is that the functional specification model is available free of charge and licenses, the resource model is still free but requires a non-disclosure agreement, while the executable model requires a license and a fee.

Another trading related usage is that the existence of certain models and/or their content can be used as search criteria when identifying components that could meet the needs of a system integrator.

The second major reason for specifying a component as a set of related models is to support the composition paradigm. When the behavior, performance, resource usage etcetera of a composition of executable components can be predicted using a composition of the relevant models, this is a very powerful tool to support integration and to help alleviate the mismatch between the functional and extra functional decomposition

mentioned in the introduction to this chapter. While this is an area still requiring significant research[6], we deem it an important and distinguishing feature of the architecture. A very simple example is that the total size of the executable code segment of all components can easily be computed by adding the sizes of the individual component's executable code segments. Here the model is a single number denoting the segment size in e.g. bytes, and the composition operator is a simple addition. A more elaborate example is that given a behavioral simulation model of a number of executable components the behavior of the composition of these executable components could be simulated.

Figure 7-8 depicts how tools could be used to create a model for the composition of a number of components based on the individual models of the components.

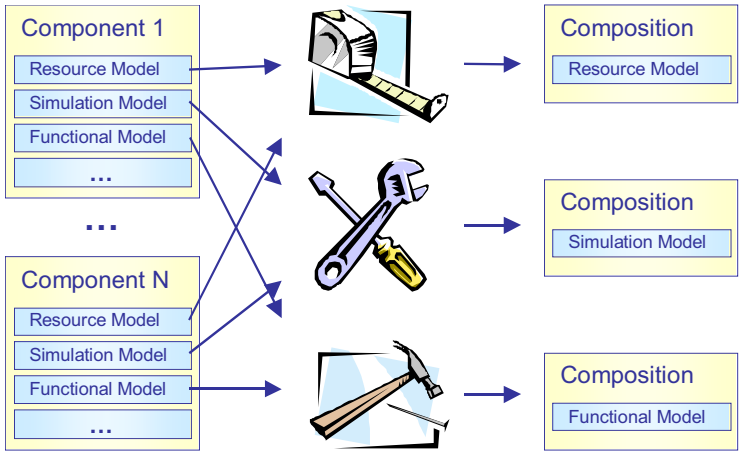


Figure 7-8. Composition of models

By assessing architecturally relevant aspects of the composition based on models, the confidence in the correctness of particular composition can be significantly increased. This is especially true in the case of models that can be automatically derived from the code (or vice versa) and for models that can be composed using some kind of automated tool.

One such aspect that we are currently investigating is that of threading analysis. Using specialized static code analysis tools, it can be determined whether a particular piece of code is thread-safe or not, and where threads

[6] Not all types models lend themselves to composition yet; this is one area where additional research is still needed.

are created. This knowledge can be used to create a threading model of all components. The composer tool can than check that non thread-safe interfaces are not called from multiple threads. This is a very common kind of fault, which is usually hard to find because it does not lead to reproducible errors.

Another, related, example is that of stack size. It is fairly simple to statically determine the sizes of the stack frames of functions. From there it is possible, in the absence of recursion, to determine the worst case stack segment needed by a component between all provided and all required interfaces. In the composition we could now calculate the maximum required stack segments of the product. A stack size that is specified too small is another common fault leading to hard to debug errors in embedded systems.

The third major use of models is found at execution time. Applications, other components, and the RRE may inspect models at run time to determine certain aspects of a component. Based upon this inspection different actions may be taken. For instance, during the activation of an executable component, the RRE may inspect the resource model(s) of the component to determine whether the platform still has sufficient resources (RAM, CPU cycles, etc.) to host the component.

Note that these models need to be accessible on the device, either as target-loaded models or through a network connection to the host on which they reside.

3.2 The Executable component model

3.2.1 Conceptual view

In Robocop all functionality is encapsulated in services that expose their functionality through interfaces, called provided interfaces. *Interfaces* group a number of semantically related named operations (a.k.a. procedure, function, or method) that can be invoked by the user of the interface. Interfaces are purely functional, i.e. they have no data members. Interfaces are first class citizens, they are defined independently of the services; hence multiple services can provide an implementation of the same interface definition. Services, then, provide a coherent set of functionality. To provide this functionality, services typically rely on access to other functionality; this dependency is modeled through required interfaces. Furthermore, each service has one special management interface, the service interface. The *service interface* contains operations for obtaining provided interfaces from the service, and for binding its required interfaces to the

provided interfaces of other services. This is analogous to the CORBA Home interface and allows 3rd party dynamic configuration of a network of services.

In this architecture, the executable component primarily serves as a container for one or more services (see figure 7-9).

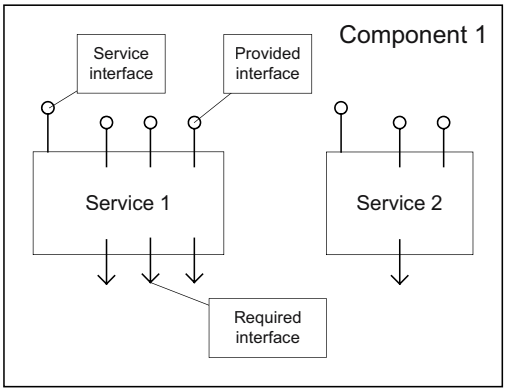


Figure 7-9. Executable component - conceptual view

Robocop follows the common convention where the user of some functionality is called the *client* and the provider is called the *server*. For product families it is important to minimize the coupling between clients and the servers. Minimal coupling not only allows independent evolution of clients and servers, it also supports product diversity by allowing clients and servers to be reused in different configurations. The coupling between clients and servers is minimized through a number of mechanisms, such as:

- **Provided interfaces** - the only[7] way to interact with a service is through its provided interfaces. Because interfaces are purely functional the implementation details of the server are hidden from the clients: the client cannot see -and therefore depend on- internal state variables etc. Thus it is possible to safely replace a server implementation with a new revision or even a totally different implementation without affecting the clients.
- **Required interfaces and 3rd party binding** - explicitly modeling of required functionality through interfaces that are dynamically bound to a

[7] Services also have attributes; an *attribute* is a syntactical shortcut for an interface with get and set value operations. Attributes are omitted from this discussion for briefness and simplicity.

provided interface by a 3rd party decouples clients from the instances of the servers. Clients specify what they need but not who will provide it; this is the responsibility of the 3rd party.

- **Multiple interfaces** - services support multiple interfaces, both for the provided and for the required side. Architecturally this facilitates narrow, well-defined interfaces. The operations in the service interface allow run-time querying and adaptation.

3.2.2 Run-time view

Services are similar to the object-oriented concept of class: a *service* is a template that must be dynamically instantiated. The instantiated services are called *service instances*; they are analogous to objects in object-oriented programming languages. Instantiation is the responsibility of the *service manager* (a factory), which is an integral part of the executable component. For each service type in the component there is exactly one service manager. The service manager can control how many service instances it will create. For some services a singleton may make sense, while for others the number of instances may be constrained by available resources.

3.2.3 The RRE and the Registry

The main responsibility of the RRE is to support the creation of service instances by instantiating and providing access to service managers. This functionality is made accessible to applications and services through an Application Programming Interface called the *client API*. Because of bootstrap issues, this API will typically be available as a static or dynamic library in the technology of the underlying platform.

The secondary responsibility of the RRE is to maintain the relations between services and executable components on one hand, and between executable components and their location in the underlying operating system's file system on the other hand. These relations determine which component should be used for creating a specific service instance, and where the executable file that contains that component can be located on the platform. These relations are made persistent in the *registry*. The RRE provides an optional API, the *registration API*, to update these relations.

3.2.4 RIDL

In Robocop all interface definitions are specified in an Interface Definition Language (IDL). The language is derived from CORBA IDL and

is called Robocop IDL or RIDL for short. The major changes are the omission of certain types that do not make sense in a resource constrained consumer device (`long long` and `long double`) or that are unsafe and thereby reduce robustness (`any`), and the addition of Robocop specific constructs.

Using IDL, components, services, interfaces as well as data types and constants can be specified[8] in a programming language neutral way. A tool, the RIDL compiler, reads the language and constructs programming language specific source code skeletons for the various constructs at both the client and server side. These skeletons contain the boilerplate code for creation and navigation of the component model artifacts; of course these still need to be manually edited to add the logic for the required functionality.

3.2.5 Interfaces and Objects

The programming model is based on the use of interfaces to access opaque objects. For this part, it faithfully follows the Microsoft COM model (Box, 1998; Microsoft, 1995; Rogerson, 1997)

At runtime an interface is represented as an interface instance. An *interface instance* refers to the implementation of the operations of the associated interface definition and to the data upon which the operations act. The *interface definition* specifies the operations and their signatures; it is the interface type. The data are a private part of the object's implementation and maintain its state.

The interface instance is a private structure of the server object; it may differ between different implementations (e.g. different services implementing the same interface). Therefore the clients are not allowed to know its structure; they only get a reference to the interface instance, aptly named *interface reference*.

Multiple interface references may refer to the same interface instance, i.e. multiple clients may access the object through that interface. Likewise, multiple interface instances may refer to the same data, which means that the object supports multiple interfaces. Figure 7-10 Shows a simple example of a server object supporting two interface types, `IntA` and `IntB`, accessed by two clients, `Client1` and `Client2`, where `Client1` accesses both interfaces and `Client2` only accesses the `IntB` interface.

[8] So actually IDL is a misnomer as higher-level constructs can also be specified. It is a well-known acronym, however, which is the reason we use it. In Space4U RIDL will be extended to an Architecture Description Language (ADL) which also specifies compositions of services and their bindings.

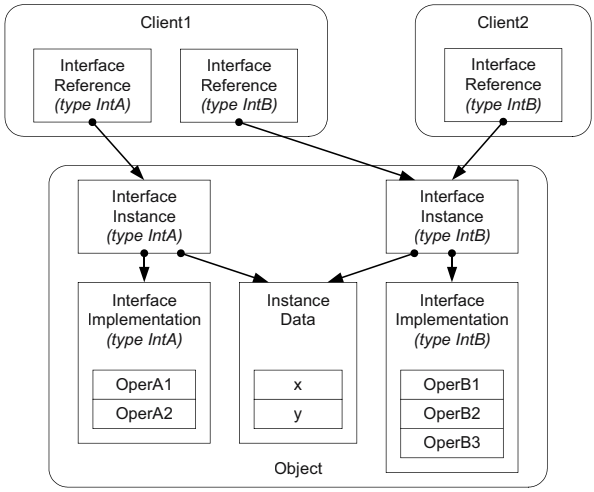


Figure 7-10. Interface Instances an References

Interface definitions are specified as a sequence of operations. Robocop also supports the concept of *interface inheritance*. When interface `IDerived` inherits from interface `IBase`, all operations of `IBase` are part of `IDerived` and have the same semantics (when only taking `IBase`'s operations into account). Interfaces can only inherit from a single other interface, but the inheritance chain can be arbitrarily deep.

When an object can be accessed through multiple interface instances, these instances must be of a different type. Clients can navigate from one interface instance to another through the `QueryInterface()` operation, which is part of the `RcIUnknown`[9] interface. The `QueryInterface()` operation takes an Interface ID (IID), which is a GUID for the interface type, as parameter, and returns an interface reference to the object's interface instance. If the object does not support that interface type, it returns the `null` reference. This is the equivalent of the dynamic cast operation in object oriented programming languages. Multiple interfaces and interface navigation are a very powerful mechanism to support product families in a robust way. A client can safely and dynamically determine the server's capabilities, expressed through interfaces, and adapt its behavior. This enables independent evolution of the client and server as well as different compositions.

[9] As a naming convention all Robocop specified global identifiers start with `Rc` to minimize name clashes.

Because multiple clients may use an object through multiple interfaces, it is not feasible to have one client control the object's lifetime (clients may be unaware of each other). Therefore a distributed co-operative lifetime control using reference counting is provided. Each interface supports the `AddRef` and `Release` operations as part of the `RcIUnknown` interface. For each additional interface reference that is created, `AddRef()` must be called. Once a client is done using an interface reference, it must call `Release()` as last operation through that reference. The object can maintain a running count of the number of outstanding references. Once that count goes to zero, there are no references and the object can destroy itself.

Every interface automatically inherits from `RcIUnknown`. Furthermore, the Microsoft COM rules for the `QueryInterface()` operation also apply to Robocop. Basically they are the identity rule where `QueryInterface()` for `RcIUnknown` through any interface reference on an object must always return the same interface reference value, and that `QueryInterface()` is reflexive, symmetric, and transitive (Microsoft, 1995).

3.2.6 Binary standard

In the end, interface instances need to be called and implemented in a programming language. Therefore the RIDL specification must be mapped to -several- programming languages. For C and C++, a well-known technique for mapping interfaces is v-tables: virtual function tables. For each interface instance a table containing pointers to an implementation for each of the operations is constructed. The object itself is a data structure that has a pointer to such a table for each supported interface and holds data that is private to the implementation. An interface reference is a pointer to the v-table pointer in the object (see figure 7-11). Adopting this language mapping automatically defines a binary standard[10], allowing the client and the server to be implemented in different programming languages.

[10] Besides the v-table, some other elements must be specified. These include how to pass exceptions, how to map in, out, and inout parameters, and the responsibilities for memory (de-)allocation.

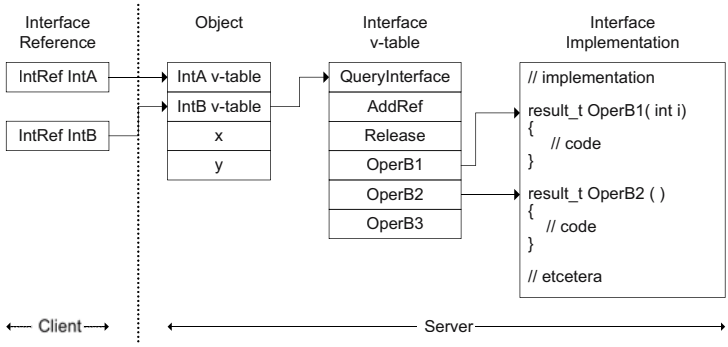


Figure 7-11. V-table based binary interface standard (simplified)

3.2.7 Services

Services extend the fine-grained notion of objects to a higher level of programming. The extensions include provided and required interface ports, attributes, and a creation framework.

Nevertheless, service instances are also objects as described in the previous section. A service must implement two management interfaces: the generic `RcIService` service interface as well as an interface that is specific for that service's definition. This latter interface, often called the *service specific interface*, extends the `RcIService` interface and can be used to type-safely manage the service instance. The RIDL compiler automatically generates the definition and the boilerplate implementation for this interface from the service specification.

Because the service instance is an object, it may implement more interfaces than just these two management interfaces. In fact, Robocop defines a number of optional interfaces that a service may implement. One such interface is e.g. `RcIReflection`, a generic interface for run-time inspection of a service instance.

The provided and required interfaces introduced in the conceptual view section are realized through *ports*, which are named interfaces on the service. Ports are not identified by their interface type, but by a name. The scope of the name is the service. This allows a service to provide and require multiple interfaces of the same type, e.g. an `SCar` service might require four `IWheel` interfaces on the `LeftFront`, `RightFront`, `LeftRear`, and `RightRear` ports. The service specific interface defines type safe operations to get a reference to a provided port as well as operations to bind an interface reference to a required port. The names of these operations are

derived from the port name. In the example of the wheels the `SCar` service specific interface would contain the following operations[11]:

```
void bindToLeftFront( in IWheel wheel );
void bindToRightFront( in IWheel wheel );
void bindToLeftRear( in IWheel wheel );
void bindToRightRear( in IWheel wheel );
```

If the `SCar` service would also provide an `ISteering` interface on the `SteeringWheel` port, the following operation would also be part of the service specific interface:

```
ISteering getSteeringWheel();
```

Port operations can also be made available by the service through the optional `RcIPorts` interface.

```
iref getProvides( in string name );
void bindTo( in string name, in iref intf );
```

These latter operations are generic and can, for reasons of type safety, only operate on the `iref` type, which is a reference to the RcIUnknown interface. A third party can, in combination with the operations from `RcIReflection`, use them to dynamically build a network without any a-priori knowledge on the two parties being bound. An example of this is a filter-graph manager that connects input and output ports of filters based on their types.

3.2.8 Dynamic Behavior

Services need to be instantiated dynamically; this is the responsibility of the RRE. The `getServiceInstance()` operation in the `RcIClient` interface will, given a service ID in the form of a GUID, identify the component that contains that service through the registry. The registry also maintains the relation between the component ID and the file in the local file system. Thus the RRE can, if necessary[12], instantiate the executable component in terms of the operating system. Depending on the operating system this involves starting an executable, loading a DLL or some similar operation.

The executable component must have one (static) interface `RcIComponent`, the *component interface*, which the RRE can call. This interface provides a.o. the `initialize()` operation, which gives the

[11] RIDL syntax, and omitting the exception definitions for simplicity.

[12] It may be that the executable component is already instantiated, e.g. as a shared library, due to a previous call to `getServiceInstance()`.

component a chance to initialize any static data, acquire resources (files, HW devices, etc) and do other run-ability checks.

After a successful call to `initialize()`, the RRE may call the other operations in this interface. One of these operations is the `getServiceManager()` operation which returns a reference to the service manager interface `RcIServiceManager`. This interface has one operation, `getServiceInstance()`, which returns a reference to the `RcIService` interface of the created service instance.

This three-step approach, first instantiate and initialize the component, then the service manager, and finally the service instance, allows for sanity and resource checks with meaningful exceptions at every stage of the creation process. If the optional resource management framework is available on a device, the RRE can call it for additional checks, e.g. for CPU resource usage, at well-defined points in the service creation process. This increases robustness because at every step in the process the creator of an entity and/or the entity itself are in full control. They have the option to dynamically verify that all preconditions for their correct functioning are met and, if they are not met, to fail in a prescribed and standardized way.

To facilitate the controlled tear-down of a component, the `RcIComponent` interface also has operations to finalize and unload the component. These operations are requests only as the component can not be unloaded if there are outstanding references to any of its service instances. The `finalize()` operation gives the component the option to ask its clients to release these references. As the implementation of this is highly component specific, there are no protocols or interfaces defined for how to realize such behavior.

4. ROBOCOP AND STREAMING

Within a device *streaming* is the process of continuously obtaining, processing, and rendering data, typically multi-media data such as digitized audio and video. For consumer products the input is usually from a network, be it broadcast or IP based, or from a medium, e.g. a DVD, source, while the output is often rendered on a screen and speakers. In many systems the processing is done in a combination of specialized hardware blocks, DSP co-processors, and software. In all but the simplest cases, a graph of processing nodes (decoders, filters, encoders) is dynamically built up. When the input data format changes, e.g. when going from stereo to surround sound, the graph may need to change. This is also true if another use-case is selected, e.g. when starting a picture in picture on the TV.

In the implementation of a streaming framework the Robocop component model can be used to great advantage. Each input, filter, or output node in the processing graph can be realized as a service. This means that the nodes can be dynamically created. When they are not needed anymore, they may destroy themselves, thereby freeing all their resources. Note that the service can contain a full software processing implementation or be an abstraction for some underlying hardware block. The services can have a standard control interface, e.g. for the basic commands of start, stop, and pause. Filters needing more specialized control interfaces can provide these through interfaces that either derive from the standard control interface or are provided by a different port. Transferring data can also be done through interfaces for input and output "pins" that can be implemented using ports. The service specific interface can be used to type safely connect output pins of one filter to the input pins of another filter. A "connection manager" can use this capability to dynamically configure different (sub)graphs. Obviously filters may be made resource or quality aware and fit in the resource management framework. The download framework may be used to update the device with new filter types, e.g. when a new coding or compression standard becomes available.

ACKNOWLEDGEMENTS

The Robocop architecture and component model are the results from an intense collaboration of many companies and people. I would like to thank the key contributors to the component model explicitly; you were worthy discussion partners (you know what I mean). From CSEM, Switzerland, Jean-Dominique Decotignie and Philippe Dallemagne; from Nokia, Finland, Petri Laine and Ronan Mac Laverty; from Philips, The Netherlands, Chritiene Aarts and Magnus Therning, and from the Technical University Eindhoven, The Netherlands, Michel Chaudron.

Furthermore, I would like to extend a special thanks to Jean Gelissen and Rob van Ommering, both at Philips Research. Jean deserves thanks because he tirelessly organized and managed both the Robocop and Space ITEA projects. Rob because he helped me with understanding software product families and because he constantly challenges me in various ways through his boundless creativity.

REFERENCES

Box, D., 1998, Essential COM, Addison Wesley
ITEA, http://www.itea-office.org
Microsoft, 1995, COM Specification, http://www.microsoft.com/com/resources/comdocs.asp
Robocop public website, 2003, http://www.extra.research.philips.com/euprojects/robocop
Rogerson, D., 1997, Inside COM, Microsoft Press
Space4U public website, 2004, http://www.extra.research.philips.com/euprojects/space4u

Chapter 8

ROBUST VIDEO STREAMING OVER WIRELESS IN-HOME NETWORKS

Jeffrey Kang, Harmke de Groot, Peter van der Stok, Dmitri Jarnikov, Iulian Nitescu, and Felix Ogg
Philips Research Laboratories, Eindhoven, The Netherlands

Abstract: More and more consumer devices in future homes will be inter-connected via wireless networks. However, due to their limited and fluctuating bandwidth and susceptibility to interference, the transport of content with real-time characteristics, such as video, is a problem. This paper presents a Quality-of-Service (QoS) architecture framework to achieve smooth, undisturbed video streaming over wireless networks, where still sufficient output quality is achieved even when the network conditions deteriorate. Our solution combines two techniques to address wireless video streaming issues, namely scalable video coding and network adaptation. The feasibility of our framework is proven by implementing receiving and decoding Signal-to-Noise Ratio (SNR)-scalable video streams on a resource-constrained consumer terminal platform. Extensive visual experiments and numerical measurements show that it is possible to achieve smooth output video, even with heavy interference from other electronic devices.

Key words: Wireless network, scalable video, streaming, Quality-of-Service, framework, MPEG, packet scheduling

1. INTRODUCTION

The vision of the Connected Home, a broadband powered environment of interconnected devices, experiences and services, is becoming more and more prominent (Digital Living Network Alliance, 2004). The Connected Home is typically characterized by a central broadband connection (e.g. xDSL) to the outside world, and a closed network inside the home (typically Ethernet based) with low-bandwidth communication. The realization of this vision is facilitated by the rapid development of wireless networking

P. van der Stok (ed.),
Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, 193-212.

technology (e.g. IEEE 802.11, Bluetooth, Zigbee). Indeed, having a wireless in-home network replaces dedicated connectors and wiring between different devices (e.g. TVs, Digital Versatile Disk (DVD) players and Personal Digital Assistants (PDA)), allowing them to interact with each other and exchange information and content with each other, such as music and movies. Such content is vital for offering Home Entertainment, an important function for in-home networks. However, using a wireless network also introduces potential problems, especially for applications which require real-time delivery of content. The cause of such problems is twofold: 1) the network is a shared medium with limited resources (e.g. bandwidth); if too many applications are competing for the network, then the minimum bandwidth required for a certain application can no longer be guaranteed; 2) the wireless network is susceptible to interference from household appliances (e.g. microwaves (Kamerman and Erkocevic, 1997) and DECT phones) and other networking (e.g. Bluetooth) devices, leading to rapidly fluctuating bandwidth, and a high data loss and corruption rate. In case of video, this results in serious artifacts such as hick-ups and frozen or corrupted images, which are unacceptable for the end user.

Scalable video coding (Van der Schaar and Radha, 2003) is a technique that can be used to cope with the above problems. The video sequence is encoded in a number of sub-streams, or *layers*, including a *base layer* (BL) containing acceptable quality video, and one or more *enhancement layers* (ELs), which further enhance the quality of the base layer. We make sure that the base layer can be independently decoded, then when the network bandwidth drops and only the base layer has been received, the decoder can still produce output images without interruptions and artifacts. Furthermore, by controlling the number of layers to display, we can achieve graceful degradation of the output quality, provided that the base layer is always received. In addition to the bandwidth limitations, the number of layers actually decoded and shown is also constrained by the available processing power inside the consumer terminal. For achieving the optimal perceived quality of service (QoS), a balance should be found between the terminal QoS and the network QoS (Jarnikov, 2003).

In this chapter, we present an architecture to transmit scalable MPEG-2 (Motion Picture Experts Group) video over a wired or wireless network, and decode the layers on the receiver side. Here we present the work on network QoS, where the number of layers actually displayed may vary at run-time depending on the arrival of the layers over the network. The feasibility of the scalable video scheme in a consumer product is shown by implementing the receiver on a resource-constrained consumer platform.

Section 2 gives an overview of related work in this area. Section 3 describes the basic concepts of scalable video and outlines different approaches. Section 4 describes our experimental hardware set-up. The architectures of our sender and receiver are described in Sections 5 and 6, respectively. Experimental results are presented in Section 7. This paper ends with conclusions and future work.

2. RELATED WORK

The concept of scalable video coding already exists for quite some time, and numerous scalable video coding schemes have been proposed (e.g. Tan and Zakhor, 1999, Wu, F. et al., 2001, McCanne et al., 1996, Domanski et al., 2000). The type of coding scheme is not the focus of this paper, rather we concentrate on the architecture to allow smooth wireless streaming and decoding of scalable video. Scalability is part of the MPEG-4 standard (Radha et al., 2001). This standard has a scalability scheme which is very efficient in terms of bandwidth overhead. Our work makes use of the MPEG-2 standard, because currently MPEG-4 is not yet a commodity, and few off-the-shelf MPEG-4 encoders and decoders exist for consumer platforms.

In Basso et al., 1999, an architecture for real-time delivery of MPEG-2 streams is described. Error concealment techniques are used to cope with network packet losses. This architecture is targeted towards wired IP networks and only handles non-scalable streams.

Two rate-based congestion control algorithms are evaluated by Mohsin and Siddiqi (2002). Scalable video is used for flow control. Both schemes rely on a feedback channel from the receiver to the sender, resulting in a long delay in adapting to the network conditions. Moreover, in a congested network even the feedback reports may get lost. Our approach to adapting to fluctuating networking conditions does not require a return channel and therefore is able to quickly adjust to the fast changing network conditions.

A jointly designed video coding, packetization and encryption technique was presented by Wee and Apostolopoulos (2001) to address also security issues over wireless networks.

Three techniques to address wireless video transmission issues are discussed and combined in a high-level framework by Wu, D. et al. (2001), i.e. scalable video, adaptive network services, and network-aware end systems. The work described in this paper comprises the design and implementation of the first two techniques and leaves the third one as future work.

3. SCALABLE VIDEO

Scalable video coding is the process of encoding video frames into a base layer and one or more enhancement layers. There are different ways to encode these layers. The basic ones are summarized below.

- **Temporal scalability**. In this encoding scheme, the base layer is encoded without any bi-directionally predicted frames (B-frames). The B-frames are put in the enhancement layer.
- **SNR scalability**. This scheme uses the errors introduced by quantizing the DCT (Discrete Cosine Transform) coefficients in the base layer to encode the enhancement layer. The enhancement layers enhance the signal-to-noise ratio (SNR) of the video, hence the name.
- **Spatial scalability**. The base layer is encoded with a lower spatial resolution. Displaying it on a display with a higher resolution requires up-scaling the decoded images, the errors introduced during which are corrected by the enhancement layers. The ELs also contain some additional information. This technique can also be used to cope with different display resolutions.
- **Data partitioning**. This approach is similar to SNR scalability. The difference is that here the quantized DCT coefficients are split up into two bit streams, one containing lower frequency (critical) coefficients and the other containing the higher frequency coefficients.

The above scalability schemes are supported in MPEG-2, and there exist also scalability schemes which combine different mechanisms, see e.g. Domanski et al., 2000. A fairly recent form of scalability is Fine-Granular Scalability (FGS), as proposed by Radha et al. (2001). This scheme uses a single enhancement layer. The transmitter can decide to transmit portions of the enhancement layer, depending on the available network bandwidth. FGS scalability has been standardized in MPEG-4. We made a comparison (Jarnikov, 2003) between the above scalability schemes in terms of implementation complexity, error propagation risk, etc. We selected the SNR-scalability approach as the implementation within our architecture. Figure 8-1 shows a MPEG-2 compliant SNR-scalable video architecture for encoding a reference video stream into a base layer and an enhancement layer. The scalable stream produced by such an encoder is not very suitable for wireless transmission. This is because the encoder uses information from both the base layer and the enhancement layer (input to the IDCT block) for motion estimation for encoding the base layer. This means that any error occurred during transmission of the enhancement layer will also affect the base layer in the decoding process, leading to visible artifacts (especially if

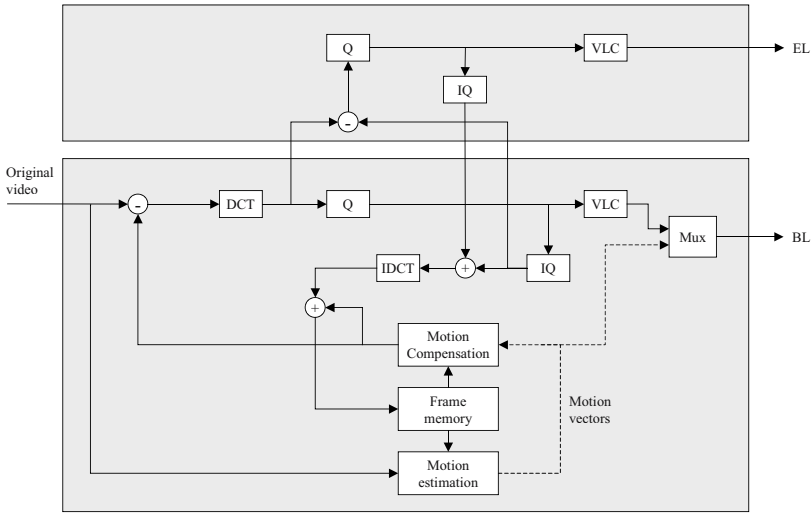


Figure 8-1. MPEG-2 compliant SNR-scalable video encoder architecture.

the error occurs in an I- or P-frame). Therefore we propose a slightly modified scalable encoder architecture. The major differences with the MPEG-2 compliant encoder are:

1. There is no dependency in the base layer on the enhancement layer, making the base layer stream completely independent. Removing this dependency prevents any errors in the enhancement layers from propagating to the base layer.
2. Each individual layer is encoded as a separate, MPEG-2 compliant stream. This allows us to treat and transmit each layer independently and facilitates the implementation of the decoder, since standard off-the-shelf MPEG-2 decoders can be used to decode the individual layers.
3. We encode only I-frames in the enhancement layers. This reduces the impact of frame errors in the enhancement layers, otherwise an error in an I-frame would propagate to later frames.

Our scalable encoder is shown for three layers in Figure 8-2. Contrary to Figure 8-1, our approach has a lower coding efficiency, hence a higher overall bit-rate is required for the same video quality (the overhead of our coding approach can be seen in Table 8-1). The resulting scalable video structure is depicted in Figure 8-3 (the arrows indicate inter-frame/layer dependencies). After decoding, the frames of the layers can be added together to reconstruct the high-quality images. Our scalable decoder architecture is presented in Section 6.2.

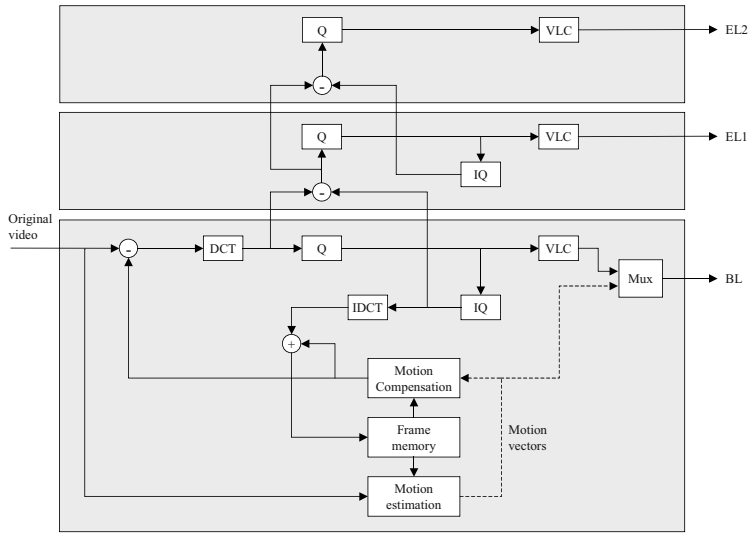


Figure 8-2. Modified SNR-scalable video encoder architecture.

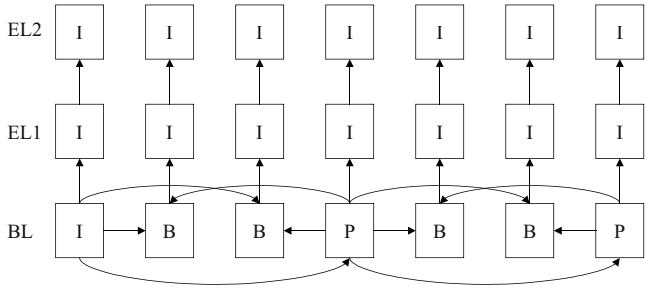


Figure 8-3. SNR-scalable video frames.

4. HARDWARE SET-UP

Our hardware set-up is shown in Figure 8-4. As the bulk of our work is focused on the network video receiver and on proving that scalable video is feasible on a CE terminal, the sender side in our set-up was still a PC for the moment. We expect that a sender can be a powerful in-home server anyway, storing all the media content in the home centrally. The scalable video receiver is a resource-constrained consumer set-top-box based on the

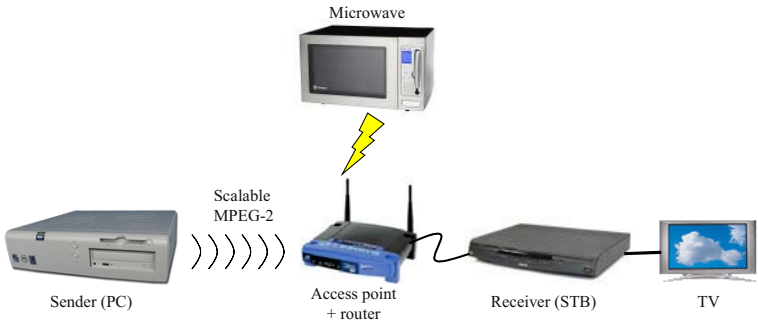


Figure 8-4. Hardware set-up.

Nexperia platform (PNX8525), the heart of which is formed by a MIPS and a TriMedia processor. The STB is connected to a TV set for display. The storage server PC is equipped with a Cisco wireless Ethernet card (IEEE 802.11b) for wireless transmission. The wireless communication was achieved by using a Linksys wireless access point with built-in router, which was connected via wired Ethernet to the STB. Even though the access point is not shared with machines outside our set-up, there are other public wireless access points spread throughout the building, which cause some interference during 'normal' circumstances. In a typical in-home network environment the interference would come from devices such as wireless phones and Bluetooth devices. To really test the performance of our scalable video approach with heavy disturbance from a realistic device in the home, we also placed a microwave in the vicinity of the access point. The experimental results are presented in Section 7.

5. SENDER ARCHITECTURE

The sender side of our wireless streaming architecture has two important elements. First, the network protocol we use for transporting the video streams is RTP (Real-time Transport Protocol), described by Schulzrinne et al. (1996). This is an application-level protocol, which makes use of UDP (User Datagram Protocol, described by Postel (1980)). RTP is preferred over TCP (Transmission Control Protocol, described by Postel (1980)) because it takes real-time delivery of packets into account and will not retransmit packets. TCP provides reliable delivery by acknowledgments and packet retransmission, however the number of transmissions may exceed the wireless network capacity with as consequence that the deadlines of the

packets are not met anymore. The architecture of our RTP sender is described in Section 5.1. Another important aspect is the use of an appropriate network packet scheduling scheme, in order to achieve that the base layer comes through as much as possible even when the network bandwidth drops. This packet scheduler is discussed in Section 5.2.

5.1 RTP sender architecture

For sending MPEG-2 video elementary streams using RTP, the streams have to be packetized in RTP packets in a special way, which has been standardized in RFC 2250 (Hoffman et al., 1998). Figure 8-5 shows our sender architecture for three layers (BL + 2 ELs). Each layer is stored in a separate file, and the path from file to the network consists of the following chain of processing steps: 1) a *file source* which reads from the file, 2) a *RFC2250 encoder* which transforms the MPEG-2 video to an RFC 2250 compliant format, 3) a *RTP MPEG encoder* which packetizes the video in RTP packets, and 4) a *UDP sender* which sends the RTP packets using a UDP socket. Such a chain can be instantiated multiple times depending on the number of layers. Each chain is running as a separate thread, and a timestamp-aware scheduling policy is used to make sure that the packets with the same timestamps in all three layers are sent (roughly) at the same time. A more detailed description of the network sender is given by Meijer (2004).

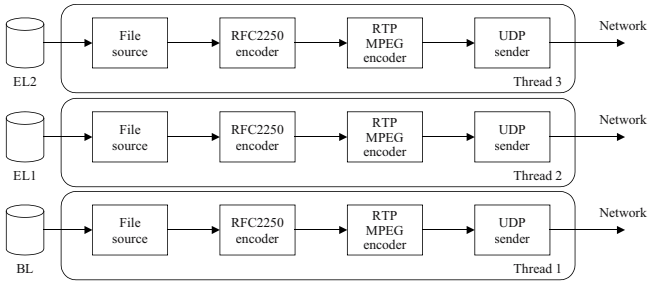


Figure 8-5. RTP network sender architecture.

5.2 Network packet scheduling

An important pre-condition of our scalable video approach is that even in the case of a drop in bandwidth, the base layer will (almost) always be received, to still produce some video output. Furthermore, we would like to

achieve that the base layer is received before the enhancement layers, since they are only useful if the base layer is already there. We try to enforce this as much as possible by incorporating a *prioritized* network packet scheduling scheme on the sender side. By assigning a higher priority to the base layer than the enhancement layers, it is assured that the frames in the base layer are always sent before the corresponding frames in the enhancement layers, in the case that they are buffered in the transmission queues when the network bandwidth drops. Having this prioritized scheduling scheme helps making sure that the video remains smooth in case of decreased bandwidth, but there is one problem. Occasionally, the network bandwidth may drop for a longer period of time. Such a temporary drop may have many causes, for example when a Bluetooth device is turned on, or by Rayleigh fading (caused by interference of the main signal by the same signal arriving over different paths) when walking around with a wireless device.

When this occurs, the transmitter queues for the enhancement layers will get fuller since only the base layer is sent. Then, when the bandwidth increases again, all the (out-of-date) packets in those queues have to be sent out first, therefore it takes some time before the enhancement layers catch up and high-quality video is displayed. Therefore, we present a new packet scheduling scheme called FirmPrio. In addition to prioritized packet scheduling, FirmPrio also takes delivery deadlines into account, and it will drop packets that can never be received in time. This saves network bandwidth, and also allows the video to recover faster because out-of-date packets are dropped. Such network adaptation service makes it possible to cope with fast varying network conditions without the need for a feedback channel from the receiver. It is possible to assign different deadlines to the layers. The relative importance of the base layer is also reflected here: the base layer stream is assigned an infinite deadline such that it will never be dropped; the assigned deadlines decrease for each higher enhancement layer. FirmPrio is described in more detail in Ogg, 2002.

6. RECEIVER ARCHITECTURE

The functionality of the receiver is partitioned into two parts: 1) the reception of the MPEG-2 scalable video streams, and 2) decoding the individual streams and adding them together to produce the final output. These parts were mapped onto the MIPS and TriMedia of the PNX8525, respectively. They are described in Sections 6.1 and 6.2.

6.1 RTP receiver architecture

Figure 8-6 shows the architecture on the MIPS side for receiving and parsing three video layers encapsulated in RTP packets. From the RTP point of view, the three layers are received in separate RTP sessions, each through a different port. Each incoming stream undergoes a number of processing steps before being fit to be passed to the MPEG-2 decoder:

1. *RTP reception*: the RTP packets are received via a UDP socket.
2. *RTP depacketization*: the RTP packets are split into the fixed RTP header and the RTP payload.
3. *Frame construction*: this step collects the RTP packets that belong to the same frame. It does so by examining the timestamps of the incoming packets; the packets with the same timestamp belong to the same frame. The frame constructor also checks for missing RTP packets by checking the RTP sequence numbers. Reordered and duplicated packets are discarded. If it is detected that one or more packets are missing, then the frame constructor tries to find out to which frame they belong and will discard the whole frame. This is the simplest error concealment technique; more sophisticated techniques can also be implemented. Discarding incomplete frames early also saves communication bandwidth between the MIPS and the TriMedia. The output of the frame constructor consists of the frames, frame sizes (in bytes), and the frame numbers (derived from the RTP timestamps).

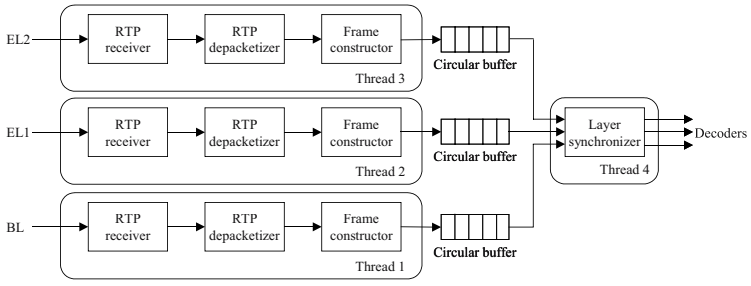


Figure 8-6. RTP network receiver architecture.

The above described steps are executed together as one sequential thread, which is instantiated for each video layer. The output of each layer is stored in a circular buffer for further processing by the *layer synchronizer*. This component (running as one thread) reads from each of the input buffers (one per layer), one frame at a time, and sends the frames to the decoder (Section 6.2). It does so in a blocking way for the base layer, and in a polling manner

for the enhancement layer buffers. If the enhancement layer buffer is empty (because packets and hence frames were dropped earlier in the chain) then it will be skipped and the next layer is read. Performing blocking reads for the base layer assures reception of the base layer at all cost, otherwise no output can be produced even if the enhancement layers have been received. The layer synchronizer also makes sure that the frames of the layers arrive at the decoder more or less in sync. Although the probability of one layer arriving too far behind the others is extremely small due to our packet scheduling scheme, the layer synchronizer provides a safety net to prevent that such a situation occurs and the layers cannot be combined after decoding to achieve sensible output.

6.2 Scalable decoder architecture

There are a number of possible implementations of the SNR-scalable decoder. They are discussed in more detail in Jarnikov, 2003. The layers produced by our encoder can each be decoded by a standard MPEG-2 decoder. For three layers, this means that three decoders have to be instantiated. Our scalable video decoder architecture is shown in Figure 8-7. Three MPEG-2 decoding chains can be identified, where the enhancement layer decoders are simplified because we only encode I-frames in those layers (Section 3). The addition of the layers works pixel-wise, and takes place after the decoding. This architecture is not optimal in terms of memory requirements, because the decoded frames of all layers must be temporarily stored. In addition, if implemented in software, it consumes more processor cycles because three separate IDCTs are required whereas only one is enough if the addition takes place before the IDCT. However, we chose such architecture for our implementation because our PNX8525 platform is equipped with a hardware MPEG-2 decoder, which is capable of decoding up to six streams at the same time. Therefore, the overhead of the extra IDCTs is nihil. Note that despite the fact that the encoder and decoder are not symmetrical in terms of the number of DCTs/IDCTs (one in the encoder and three in the decoder), the decoded stream is still correct due to the property that the DCT transformation of the sum of the functions is equal to the sum of the DCT transformations of each function.

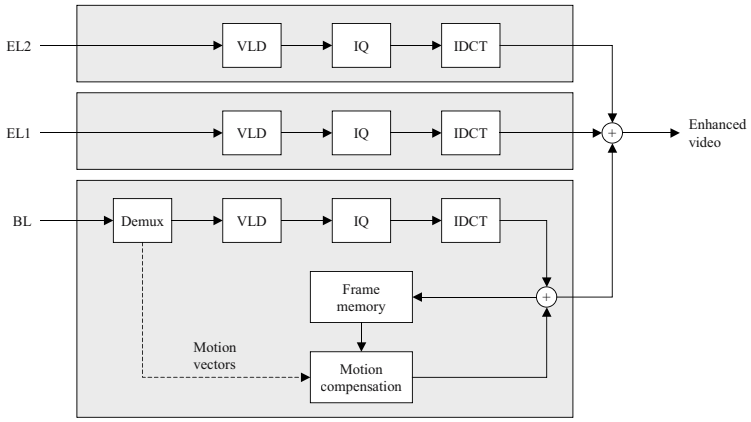


Figure 8-7. Scalable decoder architecture.

The adder module works based on interrupts, which are raised by the video output unit every 40 ms (corresponds to a frame rate of 25 frames per second). During each interrupt, it tries to read from all three layers, perform the summation, and send the result to the video output. It also keeps track of an internal timer to judge whether the frames of the layers are in sync, and that they are on time to be displayed. Late frames will be dropped. The adder always blocks for the base layer frame, and polls for the other layers. This approach is similar to the layer synchronizer on the network side (Section 6.1). The adder will add and show a frame of a particular layer only if: 1) it is on time with respect to the internal timer, and 2) the base layer and all the lower enhancement layers have arrived. For instance, if a frame that belongs to EL2 has arrived and the corresponding frame in EL1 has not, it will still be dropped. With FirmPrio we try to minimize the probability that this happens by assigning lower priorities to higher layers, such that they will be dropped more often than lower layers when the network bandwidth is not sufficient. When the adder cannot produce any output, the previous frame is repeated by the display.

7. EXPERIMENTAL RESULTS

We used a number of video test sequences to evaluate our scalable video approach over a wireless network. The 'Matrix' sequence is characterized by fast movements and lots of scene changes, while the 'Penguins' sequence has slow movements and fewer scene changes. Several non-scalable reference sequences had been generated, at different bit-rates. We then encoded the

scalable sequences, which were comparable with their reference counterparts in terms of quality, but with a significantly higher overall bit-rate (due to our coding scheme). All sequences had been encoded with a GOP size of 15 and consist of 8100 frames (almost 5.5 minutes). The sequence headers are repeated every 3 GOPs. This improves the robustness of the system because the decoder is able to re-synchronize itself if the first frame (which usually contains the sequence header) has been dropped or if no frames have been received for a long period of time. The characteristics of the test sequences are summarized in Table 8-1. The overhead of our scalable video coding technique can be seen here. For instance, sequence ref1 has a bit-rate of 3.5 Mb/s, while that of its corresponding (i.e. with about the same quality) scalable counterpart (scal1) is 4.5 Mb/s.

Table 8-1. Test sequences ('Matrix' and 'Penguins').

Sequences	Bit-rate (Mb/s)			
	Total	BL	EL1	EL2
Ref1	3.5	n.a.	n.a.	n.a.
Ref2	3.75	n.a.	n.a.	n.a.
Ref3	4	n.a.	n.a.	n.a.
Scal1	4.5	1.75	1.5	1.25
Scal2	5	2	1.5	1.5
Scal3	5.5	2	2	1.5

Figure 8-8 depicts the number of RTP packets which the different test sequences are packetized into. The numbers given for the scalable sequences are the sum of the number of packets of all three layers. Figure 8-9 depicts the packet distribution among the different layers.

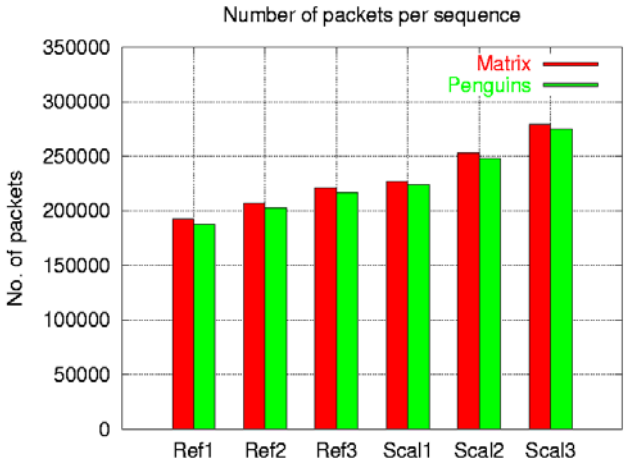


Figure 8-8. Number of packets per sequence.

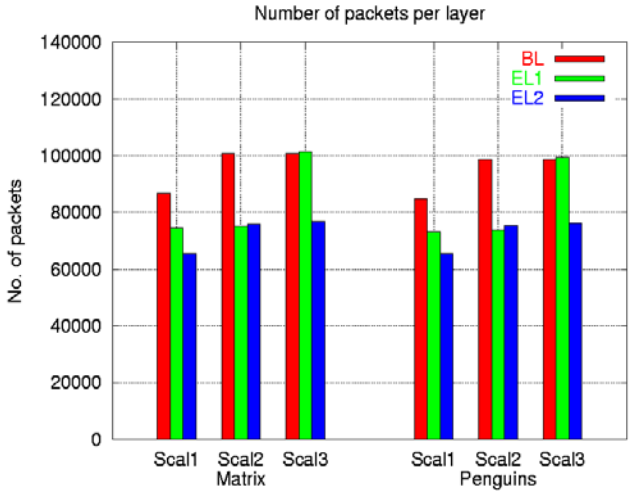


Figure 8-9. Number of packets per layer.

Figure 8-10 shows the packet allocation over the different frames for one of the reference sequences. One can see that different numbers of packets are assigned to the I-, P- and B-frames, where the highest peaks in the figure indicate the I-frames, followed by P-frames. The B-frames get the least

number of packets. Figure 8-11 shows the packet allocation over frames for the layers of one of the scalable sequences.

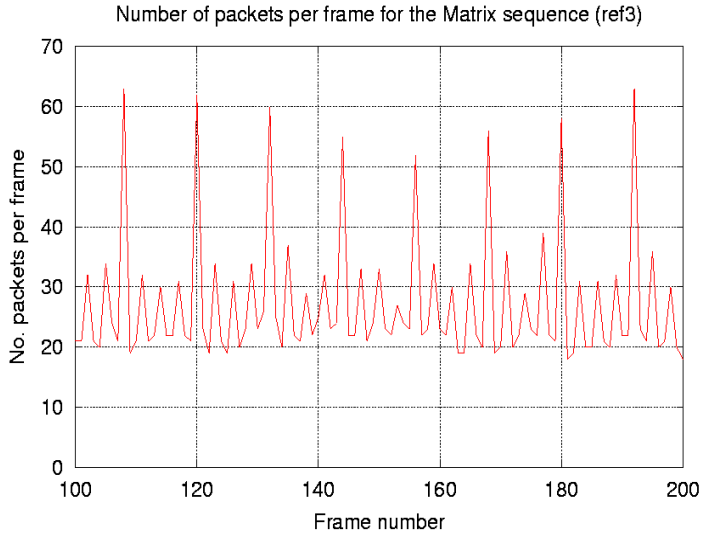


Figure 8-10. Packet allocation over frames for a reference sequence.

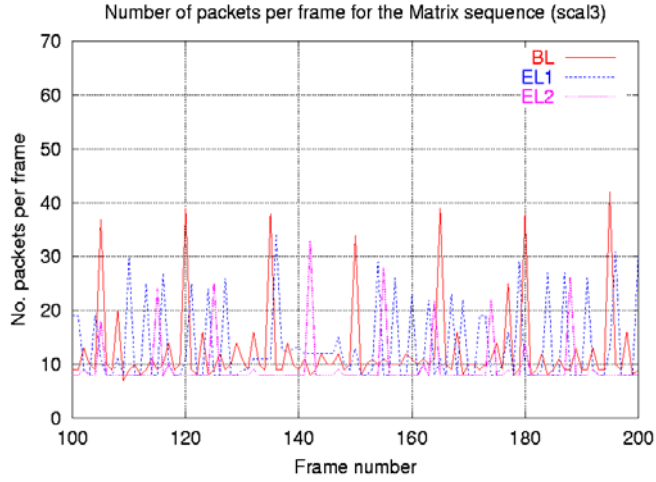


Figure 8-11. Packet allocation over frames for a scalable sequence.

For the visual perception of the different sequences streamed over the wireless network, several user tests have been conducted. They are reported in Chapter 9 of this book. Here we present numerical measurements to back up the visual experiences.

The measurements for each sequence were performed 15 times, spread over different periods during the day, in order to reduce the effect of temporarily high disturbances (e.g. due to busy traffic on other WLAN equipment). Figure 8-12 shows the average percentage of lost packets and the corresponding percentage of frames which have not completely been received (i.e. of which at least one packet was lost) and hence have been dropped for the reference sequences. Note that the percentage of missing frames is much higher than the percentage of missing packets due to discarding of incomplete frames. Furthermore, the perceived video quality may be much lower than is suggested by the number of missing frames, because one missing frame may lead to multiple distorted frames on the display, since it may be used as a reference frame for the rest of the GOP (I/P-frames). Figure 8-13 shows the results for the scalable sequences. As can be seen, the number of packets lost and the resulting number of frames lost in the base layer is nearly zero for all the sequences, resulting in video output without distortions. Furthermore, as the total bit-rate of the video stream is increased (from scal1 to scal3), EL2 is affected first, then EL1, and the BL is not affected.

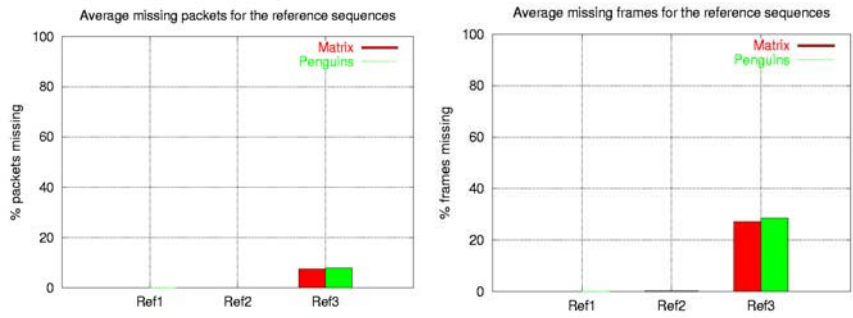


Figure 8-12. Measured average packet loss (left) and frame loss (right) for the reference sequences.

We also performed some experiments to evaluate the consequence of interference from the microwave. To this end, a reference sequence was streamed for three minutes, where after the first minute the microwave was turned on to full power for a period of one minute. A plot of the amount of completely received frames over time is shown in Figure 8-14. It can be seen

that under normal circumstances most of the frames arrive at the receiver

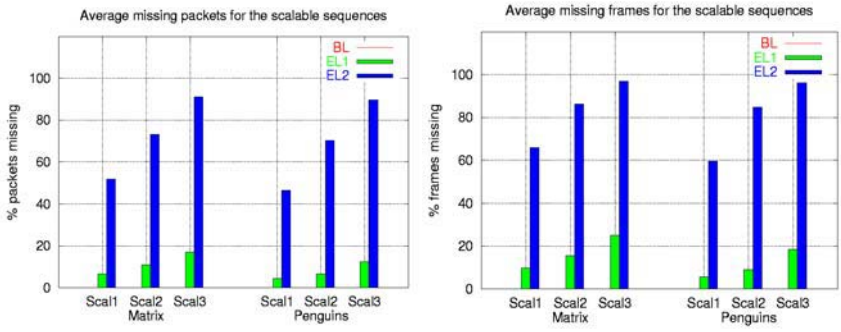


Figure 8-14. Measured average packet loss (left) and frame loss (right) for the scalable sequences.

side, but when the microwave was on the number of complete frames dropped dramatically, and with heavy variations. This reveals the burstiness of the microwave disturbance.

The same experiment was done for the scalable sequence, and the result is shown in Figure 8-15. The effect of the disturbance can be clearly seen here: the base layer is not affected at all, the enhancement layers are dropped

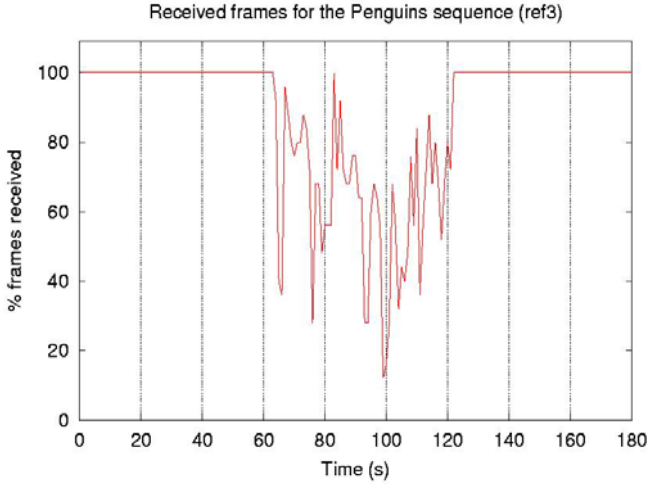


Figure 8-13. Received complete frames over time for a reference sequence, with microwave interference.

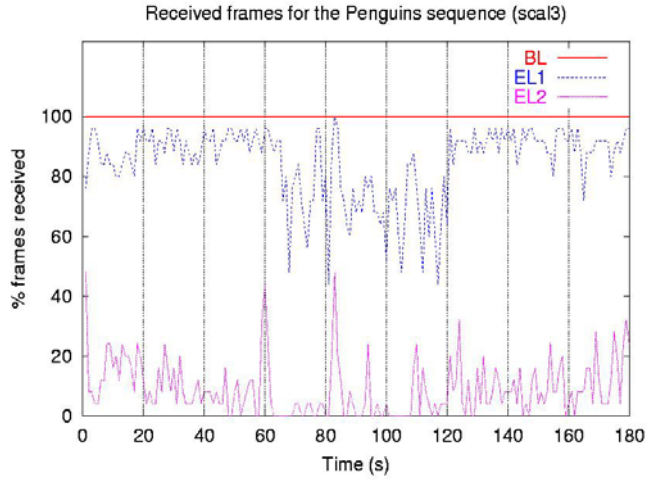


Figure 8-15. Received complete frames over time for a scalable sequence, with microwave interference.

more on average, and with higher variations.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an architecture for achieving smooth, high-quality video over wireless networks by making use of scalable video coding. Both the sender and the receiver design have great impact on the output quality. We have chosen an encoding scheme which, at the expense of a lower coding efficiency, facilitates the decoder implementation and minimizes error propagation in the enhancement layers, and allows different layers to be treated separately during transmission. This allows us to apply a prioritization scheme between the layers at the sender side, such that in case of a drop in network bandwidth the base layer will be affected much less than the enhancement layers, thereby achieving smooth wireless video. In addition, being able to drop late packets allows for quickly reaching the highest possible quality after recovering from a temporary drop in the network bandwidth. At the decoder side, a mechanism to properly deal with missing packets and frames and to synchronize the layers is needed. Experiments showed that it is possible to achieve output video with nearly no distortions even with heavy interference from the microwave. Our

solution is suitable for dealing with fast bandwidth changes (in the order of tens of milliseconds), no matter what the cause is.

Future work involves adding feedback mechanisms from the receiver to the sender, for instance to notify the sender about the number of layers actually received over a period of time. This could allow the sender application to adapt to the network conditions (Wu, D. et al., 2001) with a larger time constant by changing the number of layers to transmit, or by adjusting the sizes of the layers by transcoding. Also, it is possible to include better error protection mechanisms for the base layer to reduce artifacts even further, for instance by applying (real-time variants of) TCP (e.g. as proposed by Liang (2003)) instead of RTP. Designing a controller which in addition to the network QoS, also takes the terminal QoS into account, is also subject of future work. Experiments with a portable wireless receiver (instead of the stationary STB) showed that only the base layer was displayed. This can be explained by the fact that in this setting, the video travels along two wireless paths, namely first from the sender to the access point, and then from the access point to the portable receiver. The transmission between the sender and the access point is regulated by FirmPrio, but the transmission between the access point and the receiver is not. We also want to investigate how to install FirmPrio on the access point in order to achieve high-quality video also on the portable receiver. Furthermore, we want to investigate alternative coding methods to reduce the scalable coding overhead.

ACKNOWLEDGEMENTS

We would like to thank Koen Vrielink and Leon van Stuivenberg for helping us with the initial hardware set-up, Ralph Meijer for delivering the RTP sender, Tim Everett for providing the STB platform streaming framework and the MPEG-2 decoder implementation, and Michael van Hartskamp for his initial work on wireless video streaming. Further thanks to Michael van Hartskamp and Paul Stravers for their thorough review of this paper.

REFERENCES

Basso, A., Cash, G.L., and Civanlar, M.R., 1999, Real-time MPEG-2 delivery based on RTP: Implementation issues. In *Signal Processing: Image Communications, vol. 15*, Elsevier.

Digital Living Network Alliance, 2004. DLNA Overview and Vision White Paper. Downloadable from http://www.dlna.org.

Domanski, M., Luczak, A., and Mackowiak, S., 2000, Spatio-temporal scalability for MPEG video coding. In *IEEE Transactions on Circuits and Systems for Video Technology, vol. 10,* pp. 1088-1093.

Hoffman, D., Fernando, G., Goyal, V., and Civanlar, M., 1998, RTP Payload Format for MPEG1/MPEG2 Video. RFC 2250, Network Working Group.

Jarnikov, D., 2003, *Towards Balancing Network and Terminal Resources to Improve Video Quality*. SAI Technical Report, Eindhoven University of Technology.

Kamerman, A., and Erkocevic, N., 1997, Microwave Oven Interference on Wireless LANs Operating in the 2.4 GHz ISM Band. In *Proceedings of IEEE PIMRC '97*.

Liang, S., 2003, *Unifying the Transport Layer of a Packet-Switched Internetwork*. PhD thesis, Stanford University.

McCanne, S., Jacobson, V., and Vetterli, M., 1996, Receiver-driven Layered Multi-cast. In *IEEE Transactions on JSAC, vol. 16, no. 6*.

Meijer, R., 2004, *Volund - a research vehicle for networked video streaming*. MSc. Thesis, Eindhoven University of Technology.

Mohsin, W., and Siddiqi, M., 2002, *Scalable Video Transmission and Congestion Control using RTP*. EE384B Multimedia Networking and Communications, Department of Electrical Engineering, Stanford University.

Ogg, F.H.G., 2002, *Smoother Streaming over Wireless Networks - Real-time Scheduling the IP Transport of Video Data*. MSc. Thesis, Eindhoven University of Technology.

Philips Semiconductors Nexperia website, http://www.semiconductors.philips.com/products/nexperia/digital_video/pnx8525.

Postel, J., 1980, User Datagram Protocol. RFC 768, Information Sciences Institute.

Postel, J., 1981, Transmission Control Protocol. RFC 793, Information Sciences Institute.

Radha, H., van der Schaar, M., and Chen, Y., 2001, The MPEG-4 Fine-Grained Scalable Video Coding Method for Multimedia Streaming over IP. In *IEEE Transactions on Multimedia, vol. 3, no. 1*.

Van der Schaar, M., and Radha, H., 2003, *Scalable Video Coding - Principles, Algorithms and Standards*. Elsevier Science Ltd.

Schulzrinne, H., Fokus, G.M.D., Casner, S., Frederick, R., and Jacobson, V., 1996, RTP: A Transport Protocol for Real-Time Applications. RFC 1889, Internet Engineering Task Force, A/V Transport Working Group.

Tan, W., and Zakhor, A., 1999, Real-Time Internet Video Using Error Resilient Scalable Compression and TCP-Friendly Transport Protocol. In *IEEE Transactions on Multimedia, vol. 1, no. 2*. pp. 172-186.

Wee, S., and Apostolopoulos, J., 2001, Secure scalable video streaming for wireless networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*.

Wu, D., Hou, Y., and Zhang, Y.-Q., 2001, Scalable Video Coding and Transport over Broadband Wireless Networks. In *Proceedings of the IEEE, vol. 89, no. 1*.

Wu, F., Li, S., and Zhang, Y.-Q., 2001, A Framework for Efficient Progressive Fine Granularity Scalable Video Coding. In *IEEE Transactions on Circuits and Systems for Video Technology, vol. 11, no. 3*.

Chapter 9

PERCEIVED QUALITY OF WIRELESSLY TRANSPORTED VIDEOS

Reinder Haakma, Dmitri Jarnikov, and Peter van der Stok
Philips Research Laboratories, Eindhoven, The Netherlands

Abstract: Wireless networking technology will interconnect the consumer devices in the future homes. The capacity of the wireless technology is just sufficient to transport one or two high quality videos. When the wireless transmission is perturbed by the switching on of a microwave or a Bluetooth telephone, many artifacts appear on the screen during the display of the video. Two "scalable video" techniques are proposed to remove the artifacts. These techniques have a different effect on the quality of the video as perceived by the user. An experiment is presented which evaluates the effects of the two techniques dependent on their settings. Conclusions are drawn on the best setting dependent on the operational transmission conditions and the transmitted video.

Key words: Wireless network, packet loss, scalable video, robust video streaming, Quality-of-Service, MPEG, perceived video quality.

1. INTRODUCTION

The vision of the 'Connected Home' is based on the presence of a home network. This home network connects PCs, telephones, and consumer electronic devices. It is expected that the larger part of the network will be wireless to minimize the required amount of cabling. A disadvantage of the wireless medium is its sensitivity to the transmission conditions, which lead to possibly bursty data losses during communication. Wireless networks loose data packets more often than their wired counterparts. In case video is streamed over a wireless network, the data loss may well result in an unacceptable video quality. Losses can be recuperated from by resending

P. van der Stok (ed.),
Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, 213-239.

lost fragments. Under this common technique, the throughput of the network may become less than the required throughput, also leading to an unacceptable quality decrease.

One way of dealing with data losses is buffering. In the extreme, the entire video is copied to a hard-disk drive of the destination device. After arrival of the complete video, it can be played out without quality problems. However, for life broadcasts this is not an appropriate solution. A less extreme approach is to buffer part of the video, say half a minute to a few minutes, to cater for the changing throughput. The disadvantage of this solution are delays in life broadcasts, slow zapping from channel to channel and, last but not least, a higher bill of material.

In this article, we explore solutions for devices that only buffer two to three video frames in the local memory. In this situation, scalable video coding algorithms are positioned to reduce artifacts manifest in the rendered video due to packet loss. The scalable video code makes it possible to control what part of the video is lost under adverse transmission conditions. The various scaling techniques influence the rendered video on different technical aspects, such as the image quality of individual video frames or the video frame rate. This leads to different types of losses in video quality. In this paper, we investigate the relation between these technical aspects and the overall viewer-perceived video quality.

The results of this chapter represent the first step to the understanding how different encoding techniques may affect the perceived quality. This step investigates a steady state environment in which a video with a constant bit-rate is streamed. The chapter investigates to what extent, within the given steady-state, viewers perceive the effect of different loss-control techniques. An experiment is presented that measures the quality perception of the users as function of the chosen technique. This will be the base to investigate in a later stage how perturbations in time affect the perceived quality of wirelessly transported video.

The chapter starts with an explanation of the video coding and network transport of the video as far as needed to understand the experiment. In Section 3 the loss control techniques are explained in detail because they are the motivation for the described experiment. The following Section 4 shows the experimental results in graphical form. Section 5 concludes with a discussion of the experimental conditions, the participants and an interpretation of the statistical results.

2. VIDEO TRANSPORT TECHNIQUES

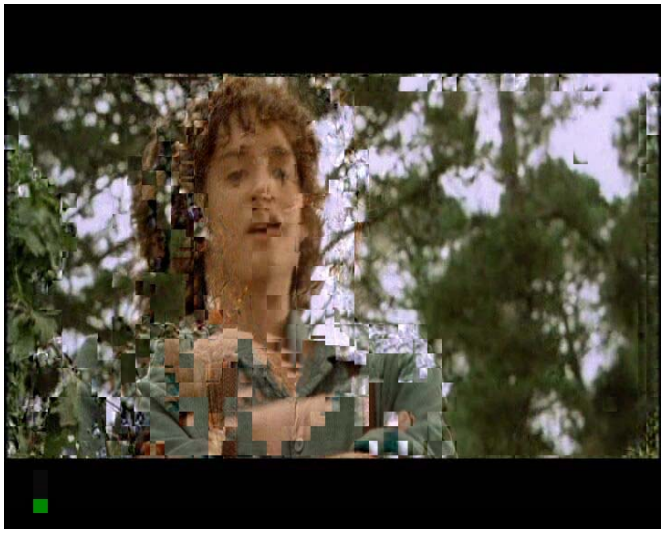


Figure 9-1. Consequences of uncontrolled packet loss.

For streaming video over a network, the video has to be encoded, transported and decoded. Together, transmission techniques and video encoding techniques determine to what extent the video quality is affected by network losses and perturbations (Kamerman and Erkocevic, 1997). When no measures are taken, artifacts, as shown in Figure 9-1, may appear.

2.1 Networking techniques

Figure 9-2 shows a typical video transmission set-up. In the example, the video is transmitted from a DVD player source to a television screen. This video material is encoded according to a Moving Pictures Expert Group (MPEG) standard.

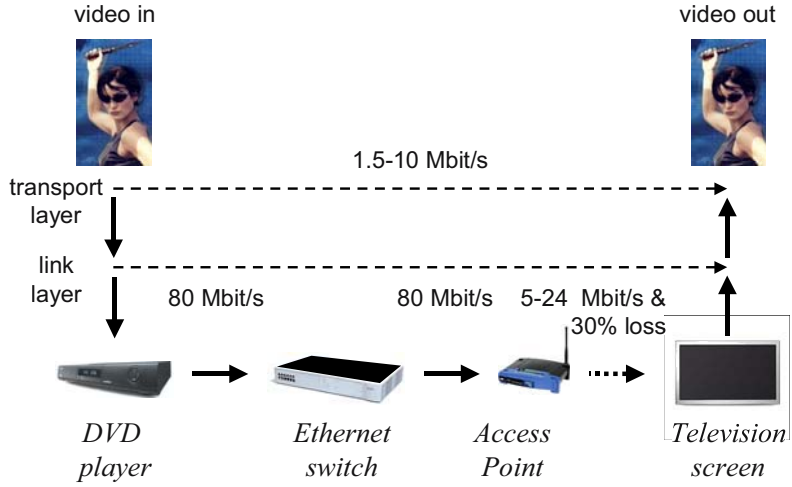


Figure 9-2. Schematic overview of video transport.

At the transport level, video frames containing a complete picture, are sent over. Video material of low quality digital Standard Definition TeleVision (SDTV) are generated with a bit rate of approximately 1.5 Mbit/s, while high quality SDTV video yields about 10 Mbit/s [1]. At the Link layer the frames are decomposed into packets, the unit of transportation over the link. In a first step, an Ethernet cable connects the DVD player to an Ethernet switch. This switch adds the packet to a queue and passes it on via another Ethernet cable to an Access Point (AP). In the AP, the packet is also cached and after that sent on over the wireless link to the final destination, the television screen. The capacity of a wired switched Ethernet link typically is around 80 Mbit/s. The capacity of wireless links varies between 5 Mbit/s for IEEE 802.11b and 24 Mbit/s for IEEE 802.11g. Losses of up to 30% can occur in the wireless link. Excluding overload conditions, the other losses in the chain are negligible in comparison.

The transport protocol of the Internet Engineering Task Force (IETF), called Real-time Transport Protocol (RTP), is used to transport the video stream from source to destination over the indicated path, (Shulzrinne et al., 2003). RTP is an application-level protocol that, in its turn, uses the User Datagram Protocol (UDP), (Postel, 1980). RTP uses timestamps to order packets and removes packets that arrive too late.

[1] Please, note that these numbers are indications. The actual numbers vary from video to video and standard to standard.

As a consequence, packets are lost in case of transmission problems while the transmission of new packets continues. At frame level, this means that correctly transmitted video frames are rendered at the right moment, while incomplete video frames are skipped. Under bursty losses, the timing of the video material relative to the start of the video will remain in tact, but one or more frames may get lost before transmission is re-established.

2.2 Video encoding techniques

The networking techniques will guarantee the timing of the video, but frames may get skipped due to packet loss. Packet loss may influence the video quality in two ways: packet loss can reduce the image quality of the individual frames and it can cause a loss of frames in the rendered video. We rely on the robustness of the scalable video encoding techniques to reduce the impact of packet loss on the rendered video.

To actually control the part of the video code that will be lost in case of perturbations, the scalable video coding technique can be used (van der Schaar and Rahda, 2004; Domanski et al.,200; Wee and Apostolopoulos, 2001; Wu, Hou, Zhang., 2001; Wu, Li, Zhang, 2001; Vetterli et al. 1997). This technique allows video material to be encoded in multiple streams by splitting the video in multiple layers and assigning one stream to one layer. The Base Layer (BL) contains video of a basic quality level. Extra Enhancement layers (ELs) may contain additional information for each frame When BL and all ELs are correctly transmitted, the rendered picture is of higher quality than when BL and a lower number of ELs are received, which on its turn yields better quality video than when the base layer is received only.

By splitting the video into different streams, the amount of data transmitted can be adapted to the available transmission capacity: the communication *bandwidth*. The base layer is transmitted with highest priority, while higher-level enhancement layers are transmitted at lower priority than lower-level enhancement layers. In case of transmission problems, the networking protocols will drop packets containing video data from higher enhancement layers (Ogg, 2002; Rahda et al., 2001). Due to a lower priority, these layers have an increased chance of being transmitted late, and thus of being skipped. This mechanism leads to graceful degradation under perturbations: useful partial frame information that is received gives still recognizable pictures, where without the layering entire frames are skipped because one or more packets are lost (Hofmann et al., 1998).

Next to this, the MPEG2 coding standard, standardized by MPEG, offers facilities that can be used to better control frame skipping due to packet loss (Basso et al. 1999). Frames can be encoded in three different ways: as an I-frame, a P-frame or a B-frame. I-frames are self-contained. All information needed for decoding an I-frame is available in the frame itself. To decode P-frames, the information of the previous I-frame is needed. To decode B-frames, the information of both the previous and the next I- or P-frame is needed. Typically, B-frames account for about half of the bit-rate of the video.

When losses occur on the wireless link, this can lead to packet and frame loss. To get the highest possible video quality, it is best that B-frames are skipped first. By removing B-frames, bandwidth fluctuations up to 50% can already be taken care of, see Figure 9-3. If still necessary, P-frames can be skipped after that, while I-frames should only be cancelled as a last resort.

Figure 9-3 provides an example of this approach. The upper part shows the original video fragment to consist of an I-frame followed by three P-frames with two B-frames between every two consecutive I- and P-frames. The horizontal axis represents the time line, while the vertical axis represent the number of bits in a frame. We see that with high regularity a frame, with its identification written below, is displayed. The lower part of the 9-3 shows the received video fragment after the majority of B-frames have been skipped due to a drop in available bandwidth to about 60%. The figure also shows that when no frame is received the formerly received frame is displayed. For example frame B3 is displayed twice in succession, and frame P1 is displayed four times in succession.

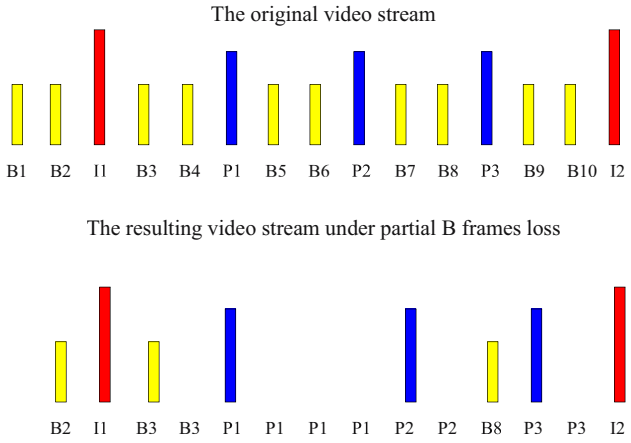


Figure 9-3. An example of a controlled loss of B-frames.

Summarizing, to counter the effects of random losses due to network perturbations, prioritization is introduced:

1. Stream prioritization makes that within multi-layer encoded video, first frames in the highest enhancement layer are skipped while base-layer frames are the last to be skipped. The effect is that variations in bandwidth result in variation of the image quality of the frames in the rendered video. This is called SNR scalability. Each individual layer conforms to the MP@ML profile.
2. Frame prioritization makes sure that within a stream first B-frames, then P-frames and only after that I-frames are skipped. The effect is that variations in bandwidth result in variation of the number of frames in the rendered video. This can be seen as a form of temporal scalability.

The two approaches can also be combined. This raises the question of how the two techniques relate to each other with respect to the rendered video quality as perceived by viewers.

3. PERCEIVED VIDEO QUALITY – AN EXPERIMENT

The video coding techniques described above are expected to provide an improved perceived video quality with respect to traditional video coding techniques in situations where the video is delivered over unreliable

communication channels, in particular wireless communication channels. As already indicated in Section 2.1, a characteristic of wireless channels is that packet loss typically happens in bursts. Therefore, the question of how these video coding techniques influence the perceived video quality can be split into two separate questions: What is their effect upon perceived video quality under ideal circumstances, when a constant video bit-rate is deployed, and what is the effect under highly varying bandwidth situations.

For the experiment, the video is encoded with a fixed number of bits per second, called the *constant video bit-rate*. The variation results from the way the bit-rate is distributed over the different frames or layers. The experiment ascertains how for a fixed video bit-rate, the quality is affected by the scalable video techniques.

3.1 Selected coding schemes

An experiment was conducted studying the impact of the coding schemes upon the perceived video quality in the constant video bit-rate situation. In particular, the study was conducted to provide insight in the trade-offs between image quality, frame rate, and layer splitting. Within the experiment, two coding schemes were put to the test. In the first coding scheme, the video material is split into two layers, a base layer and an enhancement layer. The second coding scheme used only a single layer.

Within the two-layer coding scheme, the impact of the division of the video bit-rate over base-layer and enhancement layer upon the perceived video quality is studied. In the experiment, the base-layer was allocated either a third (33%), half (50%) or two thirds (67%) of the video bit-rate. This is shown in Figure 9-4a where the horizontal axis represents the four bit-rate distribution types and the vertical axis the video bit-rate in percents of the reference video bit-rate.

In addition, the trade-off between image quality and frame rate was studied for the enhancement layer: How is the perceived video quality affected when, within the enhancement layer, the image quality is improved at the cost of the frame rate. In the experiment, the enhancement layer was either kept unchanged or it was allocated an extra one third (33%) of the video bit-rate for increasing the SNR of the frames. Skipping enhancement-layer frames compensated for this in such a way that the video bit-rate was equal in both conditions. The latter is shown in Figures 9-4b-d for different base-layer percentages, where the horizontal axis represents the time that a frame is displayed. For example in Figure 9-4c, the base layer takes 50% of the video bit-rate with value 50% on the vertical scale. The enhancement layer takes 50% + 33% of the video bit-rate which is 88% on the vertical

scale. By removing two out of five times the enhancement layer, the average total video bit-rate remains 100%.

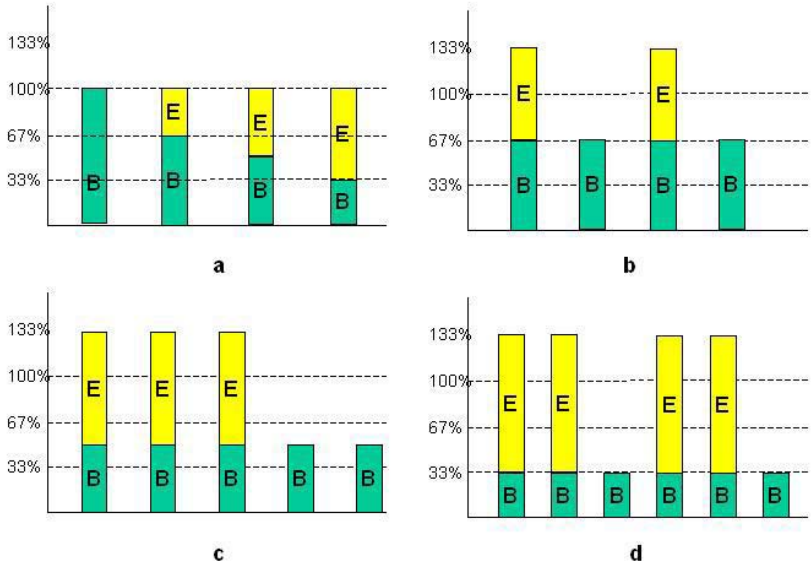


Figure 9-4. Bit size distribution over Base and Enhancement layers

The single-layer coding scheme is basically equivalent to the two-layer coding scheme, except that the total video bit-rate is allocated to the base-layer. Also within this coding scheme, the trade-off between image quality and frame rate was studied: Each base-layer frame was either given an extra one third, half or two-thirds of the bit-rate. This was compensated for by skipping frames: 25%, 33% and 40% of the available frames were skipped respectively, as shown in Figures 9-5a-c. In this figure an uncolored frame represents a skipped frame. In this way, the total delivered bit-rate was equal to 100% of the video bit-rate in all conditions.

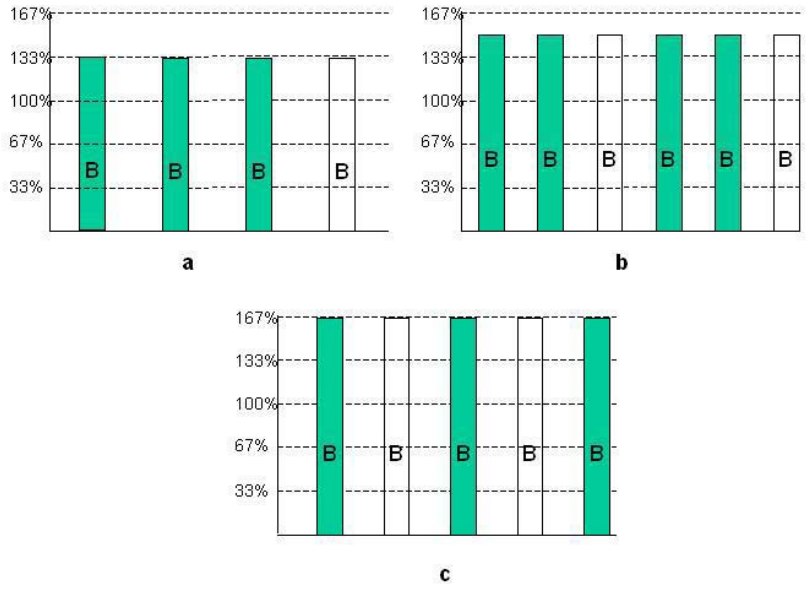


Figure 9-5. Base layer skipping

3.2 Experiment

The experiment was largely organized according to the double-stimulus impairment scale (DSIS) method as described in Recommendation ITU-R BT.500-11, see ITU-R (2002). Two groups of ten post-masters students from the Technical University Eindhoven were asked to score the quality of 18 video clips. These clips showed the same 15 seconds video fragment encoded in different ways. A video fragment was selected with considerable movement. This type of video fragment was considered to give better rise to perceivable quality differences than clips with little movement.

The students were sitting about three to four meters from a 107-centimeter, wide-screen plasma television. They were asked to score the video quality of the clips in comparison to a reference clip. For scoring each clip, subjects were presented with (1) the reference clip, (2) the clip under test, (3) the reference clip again and (4) the clip under test again, each one separated by a 5 seconds long, gray screen. The reference is a video, which consist of one layer without skipped frames. After showing these clips,

subjects were given time to score the quality of the clip under test in comparison to the reference clip. They could score the quality by marking a position on a line. The line had tick-marks at equal distance labeled '-4', '-3', '-2', '-1' and '0'. The students were asked to indicate the level degradation of the clip under test in comparison to the reference clip, with the '0' tick-mark indicating no observed difference in video quality between the clip under test and the reference clip.

After the organization of the experiment was outlined, three test runs were conducted to make the subjects acquainted with the experimental procedure and the scoring of video quality. The three clips were selected to illustrate the variety in perceived quality within the experiment: To convey that at times quality difference would be small, the reference clip itself was presented as the clip under test. So, effectively subjects scored the reference clip against itself. In addition, subjects were asked to rate two clips that were expected to receive low scores. One was encoded using a two-layer encoding, for the other the single-layer encoding with frame skipping scheme was used.

After the test run, subjects were asked to score the 18 clips. The first nine clips showed the nine versions of the original video clip, encoded using the coding methods described in the Section 3.1. The original video clip served as a reference. The bit-rate of both the test clip and the reference clip was limited to 3 Mb/s. The nine clips were shown in no particular order.

The other nine clips were encoded in the same way and also used the original clip as a reference. The difference was that the bit-rate of both the test clip and the reference clip was now limited to 6 Mb/s. The order of the clips was the same as in previous sequence. This condition was added to the experiment to study whether the influence of the video bit-rate upon the difference in perceived video quality. The 3 Mb/s situation will be called the low bit-rate condition, and the 6 Mb/s situation the high bit-rate condition.

Overall, the experiment has a within-subject design with the score for each clip as the dependent variable. For the two-layer encoding scheme, the controlled variables are:

- Bit-rate: the total video bit-rate;
- BL percentage: the percentage of the video bit-rate allocated to the base layer;
- EL image quality: the increase of the image quality in the enhancement layer as a percentage of the video bit-rate. This increase is compensated for by skipping frames in the enhancement layer.

For the single-layer encoding scheme, the controlled variables are:

- Bit-rate: the total video bit-rate;

- BL image quality: the increase of the image quality in the base layer as a percentage of the total bit-rate. This increase compensated for by skipping frames in the base layer.

The Tables 9-1 and 9-2 provide an overview of the experimental conditions. Table 9-1 describes the 12 conditions for the two-layer encoding technique, while Table 9-2 indicates the 6 conditions for the single-layer encoding technique.

Table 9-1. The experimental conditions for the two-layer coding scheme.

Video bit-rate	low	Low	low	low	low	low	high	high	high	high	high	high
BL percentage	33%	33%	50%	50%	67%	67%	33%	33%	50%	50%	67%	67%
EL image quality	+0%	+33%	+0%	+33%	+0%	+33%	+0%	+33%	+0%	+33%	+0%	+33%

Table 9-2. The experimental conditions for the single-layer coding scheme.

Video Bit-rate	low	low	low	high	high	high
BL image quality	+33%	+50%	+67%	+33%	+50%	+67%

4. EXPERIMENTAL RESULTS

4.1 Results for the two-layer coding scheme

The 12 scores of each of the 20 subjects for the video clips using the first coding scheme have been analyzed using a linear mixed-effects model. The results are shown in a series of graphs with on the vertical axis the score and on the horizontal axis the parameter under discussion. A high score (close to zero) means a low degradation of the perceived quality relative to the reference clip.

The analysis showed that overall the scores were differing from zero. The intercept was significant [$F(1,209)=198$, $p<.0001$]. The analysis also showed the main effects to be significant: bit-rate [$F(1,209)=545$, $p<.0001$], base-layer percentage [$F(1,209)=134$, $p<.0001$] and enhancement layer image quality [$F(1, 209)=15.6$, $p=.0001$]. On average, Figure 9-6 shows that the scores for the high bit-rate clips are higher than for the low bit-rate clips. Figure 9-7 shows that the scores decrease with a decreasing base-layer percentage. In Figure 9-8, the increase in image quality in the enhancement layer, accompanied by enhancement layer skipping (Figures 9-4b-d), gave a small, but significant decrease in the scores.

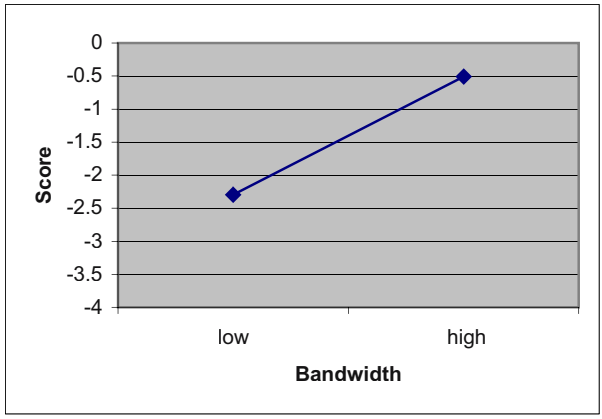


Figure 9-6. Average scores for low and high bit-rate.

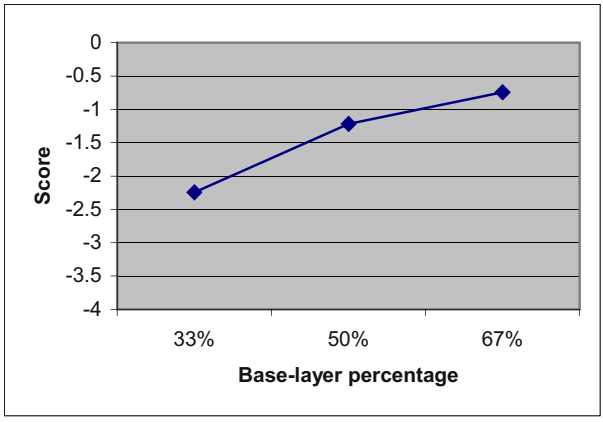


Figure 9-7. Average scores for each of the three base-layer percentage levels.

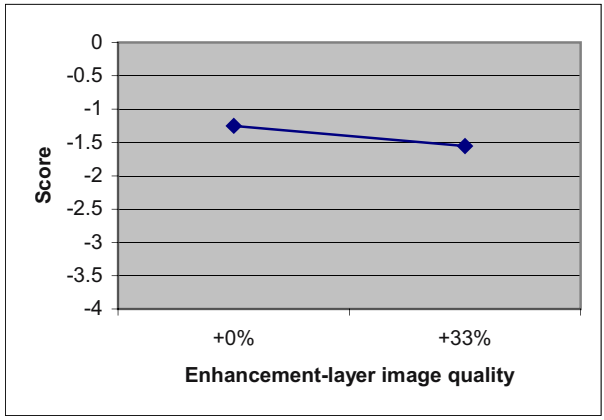


Figure 9-8. Average scores as a function of enhancement-layer increase.

The interactions between bit-rate and base-layer percentage [F(2,209)=32.9, p<.0001] and between base-layer percentage and enhancement-layer image quality [F(2,209)=5.85, p<.005] were also found to be significant. The interaction between bit-rate and enhancement-layer image quality and the three-way interaction between bit-rate, base-layer percentage and enhancement-layer image quality proved not to be significant.

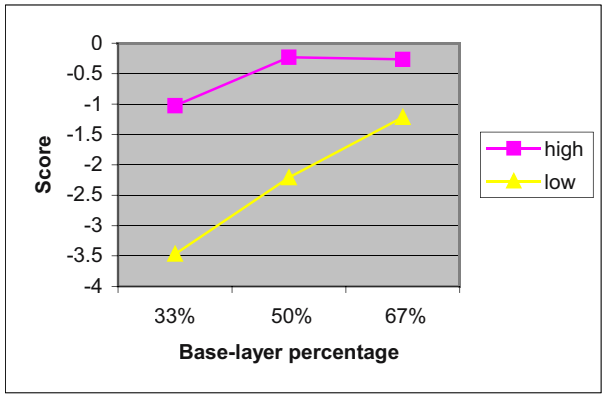


Figure 9-9. : Average scores as a function of bit-rate and base-layer percentage.

Figure 9-9 indicates that the score increase with base-layer percentage is governed by a ceiling effect in the high bit-rate situation: On average,

increasing the base-layer percentage from 33% to 50% gives rise to an increase in the scores, both in the low bit-rate as the high bit-rate situation. Another increase of the base-layer percentage, from 50% to 67%, still gives an increase in scores for the low bit-rate situation. However, in the high-bit-rate situation, the scores remain about the same.

Figure 9-10 shows that the score increase with base layer percentage depends partially on the enhancement-layer image quality: interaction between enhancement layer percentage and enhancement layer image quality. On average, the difference between the scores at the two levels of the enhancement-layer image quality is larger when the base-layer percentage is 33% than when the base-layer percentage is 50% or 67%. The interaction shows when the lines are not parallel. Parallel lines indicate "no interaction".

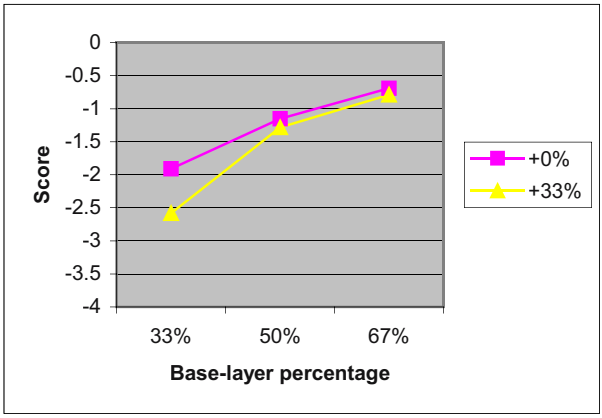


Figure 9-10. Average scores as a function of base-layer percentage and enhancement-layer image quality.

4.2 Results for the single-layer coding scheme

In this section the trade-off between image quality and frame rate is analyzed (see Figure 9-5). An increase of the frame size is compensated by the skipping of frames (base layer skipping). The 6 scores of each of the 20 subjects for the video clips using the single-layer coding scheme have been analyzed using a linear mixed-effects model.

The analysis showed that none of the main effects and interactions was significant. However, the intercept was significant [$F(1,95)=68.6$, $p<.0001$]. The scores are significantly lower than 0. Figure 9-11 shows that on average

scores do not vary with bit-rate. Figure 9-12 shows also a flat curve for base layer increase.

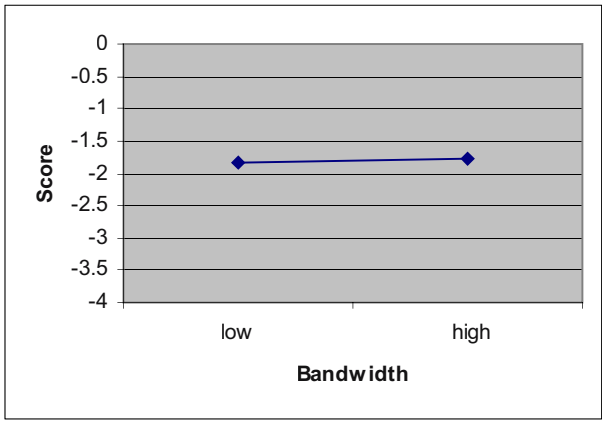


Figure 9-11. Average scores as a function of bit-rate.

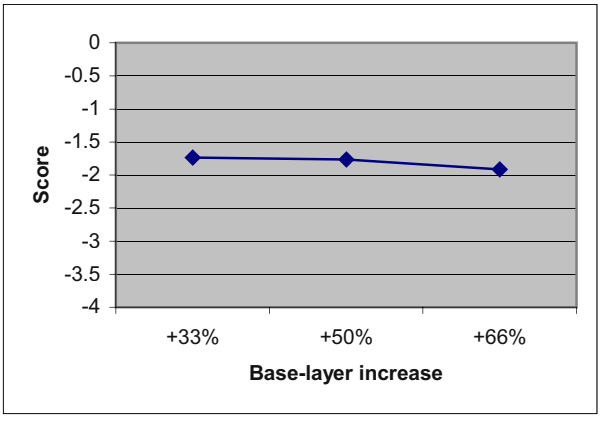


Figure 9-12. Averages scores as a function of base-layer increase.

5. DISCUSSION

5.1 About the experimental set-up

The experiment followed the double-stimulus impairment scale (DSIS) method, variant II, as described in Recommendation ITU-R BT.500-11. However, the rating was done differently because the DSIS labeling of score points was considered to be inappropriate. It was expected that the clips would not cover the entire quality range suggested by the standard labels and consequently the subjects would use only scores 0 to 2. Therefore, a continuous scale was used instead with a reference point for equal quality. Subjects had to decide for themselves how positions on the scale related to the perceived quality level.

It was expected that this would introduce an extra source of "between subject variability". Evidence that this is indeed the case can be found in Figure 9-13. For each subject, it shows the mean score over all experimental conditions together with an indication of the standard deviation (2σ). In order to check whether this source of variance between subjects had an influence on the outcomes of the experiment, the scores were normalized and the statistical analysis repeated. The scores of each subject were normalized by applying a linear transformation so that mean and variance of the participant's scores were equal to the mean and variance of the scores of all participants, see Eq. (8.1). Repeating the statistical analysis on the normalized scores revealed no differences in significant main effects and interactions. This was according to expectation. The model used to analyze the data comprised, next to the general error term for unexplained variance, an error term for each subject to compensate for the difference between within and between subject variability.

$$NormalizedScore_{ij} = ((Score_{ij} - \mu_j) * \sigma^2 / \sigma_j^2) + \mu \qquad (8\text{-}1)$$

where

$NormalizedScore_{ij}$ is the normalized score of subject j in condition i,
$Score_{ij}$ is the score of subject j in condition i,
μ_j is the mean score of subject j over all conditions,
σ_j is the standard deviation of the scores of subject j over all conditions,
μ is the mean score over all conditions and subjects, and
σ is the standard deviation of the scores over all conditions and subjects.

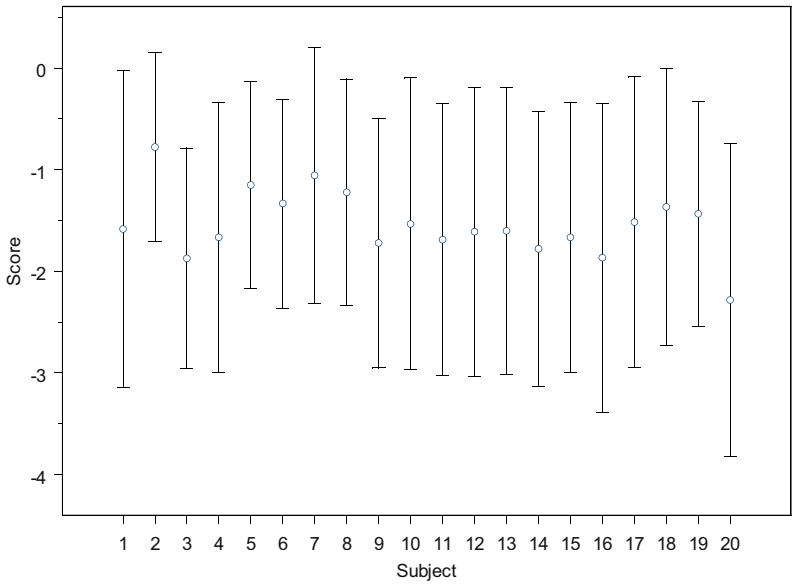


Figure 9-13. The means and two times the standard deviation of the scores for each subject.

5.2 Findings around the two-layer coding scheme

This section describes the interpretation of the findings around the two-layer coding scheme. An important first finding is that the intercept significantly differs from zero. The overall average score is –1.41. This means that overall subjects observe degradation in perceived video quality when these techniques are applied compared to untreated video.

The experimental set-up may partially explain this finding. Participants may have been biased toward quality degradation by the layout of the score line. That line had tick-marks in for quality degradation, not for quality improvements. There is some evidence for this effect. In the test set, subjects were asked to rate the low bit-rate reference frame against itself. The average score was -.45±.22, while an average score of zero was expected. This means that the bias of subjects may partially explain the finding, but it does not fully explain it. A full-scale point difference is still unaccounted for.

So next to this, the use of a two-layer encoding technique is likely to be responsible for lowering the perceived video quality. This is in line with expectations because the split of the video into two video streams, the base layer and the enhancement layer, introduces overhead in the encoding of the

video material. The bit-rate used by this overhead cannot be used for the encoding of the actual video material, thus causing a decrease in perceived video quality.

The experiment showed the average bit-rate (3 and 6 Mbit/s) to be significant. On average, the degradation in video quality is less in case of high bit-rate (6 Mb/s) than in the low bit-rate (3 Mb/s) situation. An obvious explanation is that the perceived video quality of high bit-rate video clips as shown on the screen used in the experiment, is high enough that the degradation due to re-coding is less noticeable. The low bit-rate situation turns out to be more critical in this respect.

An alternative explanation is that subjects did not rate the perceived quality relative to the reference clip, but scored the video clips on an absolute scale instead. The scores in the high bit-rate situation would be better because the perceived quality of the video in the high bit-rate condition was better, not because the degradation in perceived quality was less. We discard this alternative explanation because subjects were explicitly asked to rate the quality of the clips with respect to the reference clip. In addition, subjects were constantly reminded of this. The reference clip was shown over and over again: two times in every condition.

The experiment also showed the main effect 'base-layer percentage' to be significant. The degradation in perceived video quality is less for higher base-layer percentages. This is likely to be due to the difference in encoding of base-layer frames and enhancement layer frames. Base-layer frames are encoded more efficiently by using motion compensation: Encoding of base-layer frames may use information from previous base-layer frames. In contrast, enhancement-layer frames are encoded independently of each other. This is done because of the increased probability that enhancement-layer frames are skipped. If a video frame is skipped, all frames that rely on information from this frame cannot be decoded, even when they are received in good order. It could well be that choosing a video clip with considerable movement has contributed to the effect size of the base-layer percentage factor.

Finally, the experiment showed the main effect 'enhancement-layer image quality' to be significant. On average, the degradation in perceived video quality increases when the image quality of the enhancement-layer frame is increased at the cost of the number of frames in the enhancement layer[2]. The increase in image quality does not compensate for the loss in

[2] Note that, as long as the base layer is transmitted reliably, skipping frames in the enhancement layer does not result in a drop in frame rate of the rendered video. Skipping frames in the enhancement layer leads to variations in sharpness of the frames in the rendered video, not in the number of frames.

frame rate. However, the impact of this effect on the perceived quality is smaller than that of the other two factors.
This indicates that the perceived video quality is lowered when the average bit-rate of the video stream exceeds, to a limited extent, the available average bandwidth. In this situation, the base-layer information will be transmitted in full, while only part of the enhancement-layer frames is transferred. A practical consequence of this finding is that the video encoding had best be tuned to the available bandwidth. Dropping enhancement-layer frames causes a degradation of the perceived video quality. However, we also see that the effect size is smaller than the effect size of base-layer percentage. So, the load balancing between the base-layer and enhancement layer is more important than tuning the overall load to available bandwidth, as long as the base-layer can be reliably transmitted. What happens when the base-layer can no longer be transmitted, will be discussed in the next section.

In the experiment, we also found a significant interaction between bit-rate and base-layer percentage. The differences in quality degradation between a higher and a lower base-layer percentage are smaller in the high bit-rate situation than in the low bit-rate situation. The data indicates a ceiling effect in the high bit-rate situation.

An explanation for this finding is that in the high bit-rate situation, the perceived video quality is so high that subjects find it harder to detect quality degradation. When the bit-rate is high and the base-layer percentage is 50% and 67%, the average scores of subjects are above -.45, the average score when the low bit-rate reference frame is compared to itself.

An equivalent trend can be observed when studying the interaction between bit-rate and enhancement-layer image quality, see Figure 9-14. Also here, the difference between the scores in the high bit-rate condition is smaller than in the low bit-rate condition. However, statistically the interaction is not significant [$F(1,209)=2.02$, $p=.16$]; the hypothesis that this trend is observed by chance can not be rejected.

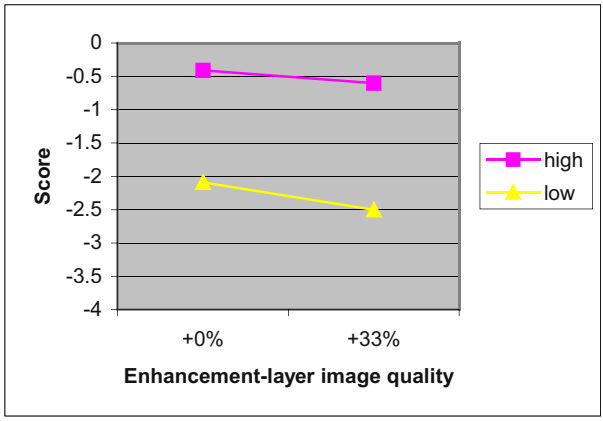


Figure 9-14. Average scores as a function of bit-rate and enhancement-layer image quality.

The other significant interaction was the interaction between base-layer percentage and enhancement-layer image quality. The difference between the average scores at the two levels of the enhancement-layer image quality is larger when the base-layer percentage is 33% than when the base-layer percentage is 50% or 67%. We have no explanation for this finding. The practical consequence of this finding is that when a low base-layer percentage is required, it is more critical to adjust the video bit-rate to the available bandwidth.

5.3 Findings around the single-layer coding scheme

This experiment is about skipping base-layer frames. Only B frames are skipped such that a maximum of 40% of the available capacity was cancelled. When a base-layer B frame is skipped, the formerly well-received frame is shown. This makes that the movements in the video appear less smooth, more jerky. Participants could well observe this effect: The intercept was significantly different from zero. Skipping base-layer frames causes a drop in perceived video quality.

There was no significant effect of bit-rate upon the scores. It could have been expected that, in the low bit-rate situation, the increased image quality would compensate better for frame-skipping than in the high bit-rate situation, assuming that the value of extra image quality is higher in the low bit-rate situation than in the high bit-rate situation. The results do not show this. Either the temporal effects are the dominant factor influencing the degradation of the perceived quality or the experiment is not sensitive enough to measure the effect.

Also no significant effect of 'base-layer increase' was found. The number of base-layer frames skipped has no influence upon the video quality degradation. Also this is a counter-intuitive finding. The number of frames skipped increases with base-layer image quality. If the temporal effects are driving the perceived video quality, a negative correlation between the scores and base-layer increase could be expected. The experiment provides no evidence in that direction.
A possible explanation for this could be that the range of number of frames skipped (25%, 33% and 40%) is too small to observe an effect. Another explanation could be that the increased SNR compensates for the reduction in frame rate. However, if this were the case, we would have expected the bit-rate effect to be significant assuming that in the high bit-rate situation the added value of the extra image quality would be less than in the low bit-rate situation. Our working assumption is that the experiment is not sensitive enough to measure such an effect.
In comparison to skipping enhancement-layer frames, skipping base-layer frames has a stronger negative effect upon the perceived video quality. Skipping a third of the enhancement-layer frames reduces the average scores only with 0.3. The reason for this is that skipping enhancement-layer frames does not introduce any jerky-ness in the movement in the video. Only the base-layer frame is shown. So, the effect of skipping enhancement-layer frames is a temporal variation in the image quality of the video frames. In contrast, skipping base-layer frames cause a previous frame to be shown at an 'incorrect' moment, but the image quality remains the same.

Although we are unable to provide an overall explanation for the findings of this part of the experiment, the practical consequence is clear. Skipping more than 25% of the base-layer frames lowers the perceived video quality and should be avoided. An open question is how quickly the perceived video quality degrades for smaller percentages. The answer to this question could provide an indication of how much risk can be taken in raising the base-layer percentage. A higher base-layer percentage increases the perceived quality, but also increase the risk that a base-layer frame will not be transferred and has to be skipped.

5.4 Related work

An experiment related to the single-layer coding part of our experiment was conducted by Hauske et al. (2003). The experiment also studies the influence of frame rate and image quality upon the perceived video quality. The application area of their experiment is different: It is about showing video on a mobile phone, instead of on a television screen. They used six

different video sequences. Their experiment comprised six bit rate – frame rate combinations with bit rates ranging from 30 kbit/s to 128 kbit/s and the frame rates ranging between 5 and 15 frames per second. They asked subjects to score four absolute criteria: total quality, smoothness of movement, quality concerning blocking effects and information value.

One of their important findings is that total quality, quality concerning blocking effects and information value could be taken together indicating the quality of the individual frames. Smoothness of movement proved to be a different quality criterion. This provides a possible explanation for our finding that base-layer increase does not influence the score. For subjects, the quality of the video may be more driven by image quality than frame rate.

Just as we do, they also report the unexpected result that an increase in frame rate while maintaining equal bit rate, does not lead to an increase of total quality. They report a very small increase in total quality when increasing the frame rate from 5 to 7 frames per second for a 50 kbit/s bit rate, and a decrease in total quality when increasing the frame rate from 7 to 16 frames per second for a 64 kbit/s bit rate. Their finding that the smoothness of movement positively correlates with frame rate, was in line with expectations.

Masry and Hemami (2001) report an experiment that studies the influence of frame rate and image quality upon the perceived video quality. They used eight different video sequences with different levels of motion shown on a television screen. In the experiment, they varied the bit rate between 40 and 800 kbit/s and the frame rate between 10 and 30 frames per second (fps). The experiment was performed using the Single-Stimulus Continuous Quality evaluation (SS-CQE) method of ITU-R BT.500-8. They report that ‘in general, the smoother motion at 30 fps did not offset the corresponding decrease in frame quality over sequences coded at 10 and 15 fps at the same bitrate. Viewers preferred the lower frame rates, and slightly favored encodings at 15 fps over those at 10 fps.’

McCarthy et al. (2004) found a similar result when studying the acceptability of streamed video for smaller screens (CIF size video on a desktop computer and QCIF size video on a handheld device). They acquired acceptability metrics for soccer video material from soccer fans. Their conclusion is that users prefer high-resolution images to high frame rate. However, they did not try to keep the bandwidth/video bit rate constant. Therefore, their study does not provide an immediately indication on how to make the trade-off between image quality and frame rate. Evidence that the acceptability of a lower frame rate may depend upon the content of the video, is provided by Apteker et al. (1995).

The paper that is closest to the multi-layer coding part of our experiment is from Zink et al. (2003). They studied the impact of variations in the amount of transmitted layers upon the perceived video quality. They used the stimulus comparison method (SC) of the ITU-R BT.500.10. The experiment comprised five different sequences. Only in two of the twelve experimental conditions, the two encodings that were compared, had the same average video bit rate. The number of layers varied from two to four. The bit rate was fixed but different for each layer. Varying network conditions were simulated by varying the number of layers rendered. Changes in the number of layers used, were a few seconds apart. The authors summarize the results of the experiment as follows:

- The frequency of variations should be kept as small as possible.
- If a variation cannot be avoided, the amplitude of the variations should be kept as small as possible.

It is hard to compare our experiment with this one. Contrary to this experiment, we kept the overall average bit rate constant. And our frequency of variation was much larger because we conducted frame-skipping resulting in repeated layer variations within groupings of three to five frames. In addition, we did not vary the amplitude and frequency of the variations. We only had two levels for EL image quality: +0% (no variation) and +33%. The results of the two experiments agree in that variations can better be avoided.

5.5 Future work

The experiment described above explores the effects of different video encoding techniques that are robust to variations in video bit-rate, upon the perceived video quality. The situation in which the encoding techniques were tested, a fixed video bit-rate, was such that only the potential disadvantages of the tested techniques were revealed. Future work has to show what the advantages of the different encoding techniques are. This means measuring the perceived video quality difference when using these robust encoding techniques in comparison to traditional encoding techniques under variable bandwidth conditions. In such experiments, the advantages in robustness should outweigh the disadvantages that became manifest in the current experiment.

The current experiment has provided us with some groundwork for such experiments. Hypotheses could be formulated around the following statements:

a) The percentage of the bit-rate that is allocated to the base layer, should be as large as possible.

b) The limiting factor in increasing the base-layer percentage is the probability that base-layer frames are not transmitted due to variations in available bandwidth. This probability should be kept small.
c) How small this probability should be, needs further investigation. The perceived video quality is definitely reduced when more than 25% of the frames of a 25 frames per second video fragment are skipped.
d) It is advisable to tune the amount of bit-rate allocated to the enhancement layer to the total bit-rate.
e) However, the skipping of enhancement layer frames only has a relatively small influence on the perceived video quality. Therefore, the risk of loosing enhancement-layer frames does not have to be minimized.
f) The higher the total bit-rate used, the less the robust encoding will deteriorate the perceived quality. However, saturation effects will limit the effects at higher bit-rates.. The size and quality of the video display will limit how useful it is to move to a coding scheme with a high bit-rate.

6. CONCLUSIONS

This first experiment taught us a few very valuable lessons. Techniques to compensate for losses during video transmission over a lossy wireless medium lead to a perceived quality decrease. The frame skipping in the base layer has a profound effect on the perceived quality for the steady state experiments performed here. Frame skipping in the enhancement layer has a smaller impact on the quality. This finding reinforces us in the conjecture that scalable video can be used to reduce the effect of variable transmission conditions. The most important lessons are:

- Using SNR layering, the perceived quality degradation is less pronounced with a “high” bit-rate of 6 Mbit/s than with a “low” bit-rate of 3 Mbit/s Using frame skipping no such bit rate dependent effect was noticed.
- The base layer should be as large as possible for SNR scalable video.

These experiments are done under steady state conditions. A natural and necessary continuation is to measure the perceived quality as function of the communication channel’s degradation interval and degradation severity.
It should not be forgotten, though, that the transmission of video over a lossy wireless medium leads to artifacts that are completely unacceptable to the user. A good balance should be struck between the application of these techniques and the probability that artifacts will appear.

ACKNOWLEDGEMENTS

We would like to thank Maddy Janse and her students. The help of Gerard Hollemans around the data analysis and the statistics was much appreciated.

REFERENCES

Apteker, R.T., Fisher, J.A., Kisimov, V.S. & Neishlos, H., 1995, Video Acceptability and Frame Rate. IEEE Multimedia, Vol. 3, Issue 3, 1995, pp. 32-40.

Basso, A., Cash, G.L., and Civanlar, M.R., 1999, Real-time MPEG-2 delivery based on RTP: Implementation issues. In *Signal Processing: Image Communications, vol. 15, No 1*, Elsevier, pp 165-178.

Domanski, M., Luczak, A., and Mackowiak, S., 2000, Spatio-temporal scalability for MPEG video coding. In *IEEE Transactions on Circuits and Systems for Video Technology, vol. 10,* pp. 1088-1093.

Hauske, G., Stockhammer, T. and Hofmaier, R., 2003, Subjective Image Quality of Low-Rate and Low-Resolution Video Sequences. In *8th International Workshop on Mobile Multimedia Communications, October 5-8 2003 Munich, Germany*, pp 37-42.

Hoffman, D., Fernando, G., Goyal, V., and Civanlar, M., 1998, RTP Payload Format for MPEG1/MPEG2 Video. RFC 2250, Network Working Group.

ITU-R, 2002, Recommendation BT.500-11: Methodology for the subjective assessment of the quality of television pictures." International Telecommunication Union, Geneva, Switzerland, 2002.

Kamerman, A., and Erkocevic, N., 1997, Microwave Oven Interference on Wireless LANs Operating in the 2.4 GHz ISM Band. In *Proceedings of 8th IEEE PIMRC '97*, pp 1221-1227.

Masry, M.A., Hemami, S.S., 2001, An Analysis of Subjective Quality in Low Bit Rate Video. In *IEEE Intl. Conf. on Image Processing 2001*, Thessaloniki, Greece, October 2001, pp 465-468.

McCarthy, J.D., Sasse, M.A. and Miras, D., 2004, Sharp or smooth?: comparing the effects of quantization vs. frame rate for streamed video. In *Proceedings of the 2004 conference on Human factors in computing systems (CHI 2004), Vienna, Austria*, pp. 535-542.

Ogg, F.H.G., 2002, *Smoother Streaming over Wireless Networks - Real-time Scheduling the IP Transport of Video Data*. MSc. Thesis, Eindhoven University of Eindhoven.

Postel, J., 1980, User Datagram Protocol. RFC 768, .Internet Engineering Task Force.

Radha, H., van der Schaar, M., and Chen, Y., 2001, The MPEG-4 Fine-Grained Scalable Video Coding Method for Multimedia Streaming over IP. In *IEEE Transactions on Multimedia, vol. 3, no. 1*, pp 53-68.

Van der Schaar, M., and Radha, H., 2004, *Scalable Video Coding - Principles, Algorithms and Standards*. Elsevier Science Ltd.

Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V., 2003, RTP: A Transport Protocol for Real-Time Applications. Internet Engineering Task Force.

Vetterli, M., McCanne, S., and Jacobson, V., 1997, Low-complexity Video Coding for receiver-Driven layered Multicast, *IEEE JSAC, Aug. 1997, pp 983-1001.*

Wee, S., and Apostolopoulos, J., 2001, Secure scalable video streaming for wireless networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*.

Wu, D., Hou, Y., and Zhang, Y.-Q., 2001, Scalable Video Coding and Transport over Broadband Wireless Networks. In *Proceedings of the IEEE, vol. 89, Issue 1,* pp 6-20, Jan. 2001.

Wu, F., Li, S., and Zhang, Y.-Q., 2001, A Framework for Efficient Progressive Fine Granularity Scalable Video Coding. In *IEEE Transactions on Circuits and Systems for Video Technology, vol. 11, no. 3*, pp 332-344.

Zink, M., Künzel, O., Schmitt, J. and Steinmetz, R., 2003, Subjective Impression of Variations in Layer Encoded Videos. In *Eleventh International Workshop on Quality of Service (IWQoS 2003), Monterey, CA, USA*. Springer Verlag, June 2003, pp. 137-154.

Philips Research Book Series

1. H.J. Bergveld, W.S. Kruijt and P.H.L. Notten: *Battery Management Systems.* 2002 ISBN 1-4020-0832-5
2. W. Verhaegh, E. Aarts and J. Korst (eds.): *Algorithms in Ambient Intelligence.* 2004 ISBN 1-4020-1757-X
3. P. van der Stok (ed.): *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices.* 2005 ISBN 1-4020-3453-9