

Jani Boutellier

QUASI-STATIC SCHEDULING
FOR FINE-GRAINED
EMBEDDED
MULTIPROCESSING

FACULTY OF TECHNOLOGY,
DEPARTMENT OF ELECTRICAL AND INFORMATION ENGINEERING,
UNIVERSITY OF OULU;
INFOTECH OULU,
UNIVERSITY OF OULU



ACTA UNIVERSITATIS OULUENSIS
C Technica 342

JANI BOUTELLIER

**QUASI-STATIC SCHEDULING
FOR FINE-GRAINED EMBEDDED
MULTIPROCESSING**

Academic dissertation to be presented with the assent of
the Faculty of Technology of the University of Oulu for
public defence in Auditorium TS101, Linnanmaa, on 6
November 2009, at 12 noon

OULUN YLIOPISTO, OULU 2009

Copyright © 2009
Acta Univ. Oul. C 342, 2009

Supervised by
Professor Olli Silvén

Reviewed by
Professor Christoph Kessler
Professor Jarmo Takala

ISBN 978-951-42-9271-2 (Paperback)
ISBN 978-951-42-9272-9 (PDF)
<http://herkules.oulu.fi/isbn9789514292729/>
ISSN 0355-3213 (Printed)
ISSN 1796-2226 (Online)
<http://herkules.oulu.fi/issn03553213/>

Cover design
Raimo Ahonen

OULU UNIVERSITY PRESS
OULU 2009

Boutellier, Jani, Quasi-static scheduling for fine-grained embedded multiprocessing.

Faculty of Technology, Department of Electrical and Information Engineering, University of Oulu, P.O.Box 4500, FI-90014 University of Oulu, Finland; Infotech Oulu, University of Oulu, P.O.Box 4500, FI-90014 University of Oulu, Finland

Acta Univ. Oul. C 342, 2009

Oulu, Finland

Abstract

Designing energy-efficient multiprocessing hardware for applications such as video decoding or MIMO-OFDM baseband processing is challenging because these applications require high throughput, as well as flexibility for efficient use of the processing resources. Application specific hardwired accelerator circuits are the most energy-efficient processing resources, but are inflexible by nature. Furthermore, designing an application specific circuit is expensive and time-consuming. A solution that maintains the energy-efficiency of accelerator circuits, but makes them flexible as well, is to make the accelerator circuits fine-grained.

Fine-grained application specific processing elements can be designed to implement general purpose functions that can be used in several applications and their small size makes the design and verification times reasonable. This thesis proposes an efficient method for orchestrating the use of heterogeneous fine-grained processing elements in dynamic applications without introducing tremendous orchestration overheads. Furthermore, the thesis presents a processing element management unit which performs scheduling and independent dispatching, and works with such low overheads that the use of low latency processing elements becomes worthwhile and efficient.

Dynamic orchestration of processing elements requires run-time scheduling that has to be done very fast and with as few resources as possible, for which this work proposes dividing the application into short static parts, whose schedules can be determined at system design time. This approach, often called quasi-static scheduling, captures the dynamic nature of the application, as well as minimizes the computations of run-time scheduling.

Enabling low overhead quasi-static scheduling required studying simultaneously the computational complexity and performance of simple but efficient scheduling algorithms. The requirements lead to the use of flow-shop scheduling. This thesis is the first work that adapts the flow-shop scheduling algorithms to different multiprocessor memory architectures. An extension to the flow-shop model is also presented, which enables modeling a wider scope of applications than traditional flow-shop. The feasibility of the proposed approach is demonstrated with a real multiprocessor solution that is instantiated on a field-programmable gate array.

Keywords: multiprocessing, scheduling, signal processing

To Kaisu

Preface

The research described in this thesis was conducted in the Machine Vision Group of the Department of Electrical and Information Engineering at the University of Oulu, Finland between the years 2006 and 2009. Two semesters (autumn 2007 and spring 2008) of the work were carried out at the Processor Architecture Laboratory of École Polytechnique Fédérale de Lausanne, Switzerland.

I would sincerely like to thank Professor Olli Silvén for supervising the thesis. His broad expertise and constant encouragement made it possible to carry through this work. I would also like to thank Professor Paolo Jenne and Dr. Philip Brisk for making possible the fruitful work that was carried out during my stay in Switzerland.

I am grateful to Professor Christoph Kessler and Professor Jarmo Takala for reviewing the thesis. The corrections made on the basis of their comments considerably improved the content of the thesis. Much of the content of this thesis was possible only through the co-operation with my co-authors Prof. Shuvra S. Bhattacharyya, Alessandro Cevrero, Tamas Erdelyi, Pekka Jääskeläinen, Dr. Sébastien Lafond, Christophe Lucarz, Victor Martin Gomez, Dr. Marco Mattavelli and Veeranjaneyulu Sadhanala. I also wish to thank Gordon Roberts for the language revision. All errors still left are mine.

The generous financial support for this thesis was provided by the Graduate School in Electronics, Telecommunications and Automation (GETA), Tekes and the Nokia Foundation.

I would like to thank all my colleagues in the Machine Vision Group, especially Juho Kannala and Esa Rahtu, for daily discussions during lunch breaks. Finally, I would like to give thanks to my parents for supporting me over the years. Most of all, I want to thank Kaisu for her love and support during my studies.

Oulu, September 2009

Jani Boutellier

*Lead me in thy truth, and teach me:
for thou art the God of my salvation;
on thee do I wait all the day.
Psalm 25:5 (KJV)*

Abbreviations

3G	<i>Third Generation</i>
AC	<i>Alternating Current</i>
ASIC	<i>Application-Specific Integrated Circuit</i>
ASIP	<i>Application-Specific Instruction Processor</i>
ASP	<i>Advanced Simple Profile</i>
ATSP	<i>Asymmetric Traveling Salesman Problem</i>
AVS	<i>Audio Video Standard</i>
BCU	<i>Batch Control Unit</i>
CAL	<i>Cal Actor Language</i>
CISC	<i>Complex Instruction Set Computer</i>
CMOS	<i>Complementary Metal-Oxide-Semiconductor</i>
CMP	<i>Chip Multiprocessor</i>
CMU	<i>Co-Processor Management Unit</i>
COP	<i>Co-Processor</i>
CPU	<i>Central Processing Unit</i>
DG	<i>Directed Graph</i>
DC	<i>Direct Current</i>
DMA	<i>Direct Memory Access</i>
DSP	<i>Digital Signal Processing</i>
DVB-H	<i>Digital Video Broadcasting - Handheld</i>
EFSM	<i>Extended Finite State Machine</i>
ELLF	<i>Enhanced Least-Laxity-First</i>
EPFS	<i>Extended Permutation Flow-Shop</i>
FFT	<i>Fast Fourier Transform</i>
FIFO	<i>First-In First-Out</i>
FPGA	<i>Field-Programmable Gate Array</i>
Fps	<i>Frames per Second</i>
FSM	<i>Finite State Machine</i>
FU	<i>Functional Unit</i>
GOPS	<i>Giga Operations per Second</i>
GOV	<i>Group of Video Object Planes</i>

HDTV	<i>High-Definition Television</i>
HSDF	<i>Homogeneous Synchronous Data Flow</i>
HW	<i>Hardware</i>
I/O	<i>Input/Output</i>
IC	<i>Integrated Circuit</i>
IDCT	<i>Inverse Discrete Cosine Transform</i>
IDCT2D	<i>Inverse Discrete Cosine Transform - 2-Dimensional</i>
IP	<i>Intellectual Property</i>
ISA	<i>Instruction Set Architecture</i>
ISO	<i>International Organization for Standardization</i>
ISR	<i>Interrupt Service Routine</i>
ITU-T	<i>International Telecommunication Union - Telecommunication Standard-ization Sector</i>
JPEG	<i>Joint Photographic Experts Group</i>
LSI	<i>Large-Scale Integration</i>
LTE	<i>Long Term Evolution</i>
LUT	<i>Look-Up Table</i>
Mbps	<i>Mega Bits per Second</i>
MIMO	<i>Multiple Input Multiple Output</i>
MoC	<i>Model of Computation</i>
MPEG	<i>Moving Picture Experts Group</i>
MPSoC	<i>Multiprocessor System-on-Chip</i>
NP	<i>Nondeterministic Polynomial (time)</i>
OFDM	<i>Orthogonal Frequency-Division Multiplexing</i>
PE	<i>Processing Element</i>
PFS	<i>Permutation Flow-Shop</i>
PIO	<i>Parallel Input/Output</i>
PSDF	<i>Parameterized Synchronous Data Flow</i>
QR	<i>A decomposition of a matrix</i>
RAI	<i>Randomized Arbitrary Insertion</i>
RAM	<i>Random Access Memory</i>
RF	<i>Register File</i>
RISC	<i>Reduced Instruction-Set Computer</i>
RTU	<i>Real-Time Unit</i>
RVC	<i>Reconfigurable Video Coding</i>

SDF	<i>Synchronous Data Flow</i>
SDRAM	<i>Synchronous Dynamic Random Access Memory</i>
SFU	<i>Special Functional Unit</i>
SIMD	<i>Single Instruction, Multiple Data</i>
SMPTE	<i>Society of Motion Picture and Television Engineers</i>
SP	<i>Simple Profile</i>
SRAM	<i>Static Random Access Memory</i>
SW	<i>Software</i>
TCE	<i>TTA-based Codesign Environment</i>
TTA	<i>Transport-Triggered Architecture</i>
TV	<i>Television</i>
UMC	<i>United Microelectronics Corporation</i>
VC-1	<i>SMPTE 421M Video Codec</i>
VHDL	<i>VHSIC (Very High Speed Integrated Circuits) Hardware Description Language</i>
VLIW	<i>Very Long Instruction Word</i>
VLSI	<i>Very Large Scale Integration</i>
VO	<i>Video Object</i>
VOP	<i>Video Object Plane</i>
YUV	<i>A color coordinate system</i>
A	<i>Silicon area</i>
a, b, c, d	<i>An edge of a data flow graph</i>
d_m	<i>Distance between two jobs on machine m</i>
d_{opt}	<i>Optimal distance</i>
D	<i>A dataflow actor</i>
e	<i>An edge of a data flow graph</i>
E_i	<i>Memory element i</i>
f	<i>Frequency in Hz</i>
i	<i>Memory element index variable</i>
I_{MAX}	<i>Large integer constant</i>
I_n	<i>Slope index for job n</i>
J	<i>A set of jobs</i>
m	<i>Machine index variable</i>
M	<i>Number of machines</i>
n	<i>Job index variable</i>

N	<i>Number of jobs</i>
k	<i>Number of ports in a memory</i>
K	<i>Number of CAL actors</i>
K_p	<i>Throughput factor</i>
l	<i>Number of state transitions originating from the initial FSM state</i>
$L_{M=m}$	<i>Average scheduled operation length when number of machines is m</i>
L_o	<i>Average scheduled operation length</i>
L_p	<i>Average parallel schedule makespan</i>
L_{pX}	<i>Average parallel schedule makespan for average operation length X</i>
L_s	<i>Average sequential schedule makespan</i>
N_o	<i>Average number of operations</i>
$o_{n,m}$	<i>Offset of operation belonging to job n on machine m</i>
$O_{n,m}$	<i>Operation belonging to job n on machine m</i>
p	<i>Job index variable</i>
P	<i>A set of interface ports</i>
$ P $	<i>The number of different values transmitted by P</i>
P_m	<i>Processing element m</i>
S	<i>Speedup factor</i>
$S_{M=m}$	<i>Speedup factor when number of machines is m</i>
$t_{n,m}$	<i>Execution time of operation on machine m that belongs to job n</i>
T_s	<i>Average scheduling time</i>
T_t	<i>Sum of average scheduling time and average parallel schedule makespan</i>
T	<i>A set of actors</i>
U	<i>A set of actors</i>

Contents

Abstract	
Preface	7
Abbreviations	9
Contents	13
1 Introduction	17
1.1 Contribution of the thesis	20
2 Computationally intensive mobile applications	23
2.1 Video decoding	23
2.1.1 Reconfigurable Video Coding	27
2.2 Wireless baseband processing	28
2.3 Summary	30
3 Mobile devices for computationally demanding applications	31
3.1 Energy-efficiency	33
3.2 Time-to-market and design cost	35
3.3 Multiprocessor synchronization issues	37
3.3.1 Direct processing element interfacing	38
3.3.2 Offline scheduling of parallel PEs	40
3.3.3 Run-time scheduling	41
3.4 Conclusion	43
4 Representing and scheduling multiprocessing applications	45
4.1 About Models of Computation	46
4.1.1 Synchronous data flow	46
4.1.2 Parameterized synchronous data flow	48
4.2 Flow-shop scheduling	50
4.3 Granularity of modeling	53
4.4 Extended permutation flow-shop scheduling	54
4.5 Combining synchronous data flow and flow-shop	56
4.6 Relation of EPFS to more general scheduling approaches	58
5 Quasi-static scheduling of an RVC decoder	61
5.1 Related work	62

13

5.2	The proposed approach	63
5.2.1	Preprocessing	63
5.2.2	Actor classification	65
5.2.3	EFSM unrolling	67
5.2.4	Parameter-specific system-level graphs	69
5.3	PSDF model of the system	71
5.4	Scheduling	72
5.4.1	Off-line scheduling	72
5.4.2	On-line scheduling	73
5.5	Experiments	73
5.6	Discussion	74
6	Efficient run-time sequencing of multiprocessor schedules	75
6.1	General flow-shop scheduling	75
6.1.1	No-wait timetabling without job ordering	76
6.1.2	Semi-active timetabling without job ordering	79
6.1.3	Palmer job ordering	80
6.1.4	Job ordering by ATSP solving	81
6.2	The performance of flow-shop algorithms	82
6.3	Discussion	88
7	The influence of memory architectures on flow-shop scheduling	89
7.1	Generalizing the pipelined interconnect	90
7.2	The memory architecture design space	93
7.3	Flow-shop scheduling with a single shared memory	94
7.4	Flow-shop scheduling with parallel memories	97
7.5	Experiments	99
7.5.1	Parallel memory architecture	104
7.5.2	Shared memory architecture	105
7.5.3	Experiments with varying numbers of PEs	106
7.6	A simplified memory conscious solution	107
7.6.1	Abandoning bypasses	108
7.6.2	Experiments	109
7.7	Application specific memory architectures	111
7.7.1	EPFS and memory architectures	113
7.8	Summary	114

8	Hardware support for scheduling and dispatching	117
8.1	Dispatching applications on multiprocessors	117
8.2	LSI circuit for scheduling and dispatching	119
8.2.1	Related work	120
8.3	Implementation environment	123
8.3.1	Detailed CMU functionality	124
8.4	System level experiments	126
8.4.1	Parallel MPEG-4 decoding	127
8.4.2	Orchestration of fine-grained PEs	130
8.5	Discussion	132
9	Conclusion	133
9.1	Future work	134
	References	135

1 Introduction

Present-day 3G mobile phones already perform more computations each second, than a typical desktop computer processor (Dally *et al.* 2008), and the demand for computational power is growing steeply as new standards are adopted for data connections or video coding (Silvén & Jyrkkä 2007).

Unfortunately, there is a major hindrance to the current tendency: high-performance portable devices are dependent on an in-built power source. As the computational technology advances in great leaps, the development of batteries has remained nearly static (Silvén & Jyrkkä 2007, Rabaey 2000). This has led to another great computing trend of recent years, which is striving for energy-efficiency. In this thesis, energy-efficiency means having a low average power dissipation while providing high computational performance, which has been previously quantified by the unit $\frac{pJ}{operation}$ in the work of Balfour *et al.* (2008).

It has been identified already some years ago that conventional single-processor systems are reaching their limit in achievable energy-efficiency. This limitation comes from the fact that the only possibility to increase the throughput of single-processor systems is to increase their clock frequency and instruction level parallelism. Nevertheless, this direction of development has led to processors (Balfour *et al.* 2008) that provide the energy-efficiency required by portable devices. But new challenges are coming.

The power consumption of an integrated circuit consists of two major parts: static and dynamic power consumption. Static power consumption takes place on every powered-on transistor on the chip, whether it is switching or not. Dynamic power consumption, on the other hand, takes place when a transistor changes state. Reducing the number of transistors on the chip (die size) clearly reduces the overall power consumption. Another, less evident way of reducing power consumption is to reduce the operating voltage of the chip. Halving the operating voltage will reduce the power consumption by a factor of four, but it also reduces the chip's maximum operating frequency by more than half. (Kim *et al.* 2003)

Parallel processing offers a way to produce high performance with a low operating frequency. Several processing elements (PEs) running at a lower frequency can offer the same performance as a single PE running with a high frequency (Horowitz *et al.*

2005). Naturally, several PE instances lead to an increased number of transistors, which again increases (static) power consumption. Kim *et al.* (2003) point out that *pipelined* implementations are a low power solution that outperform single PE architectures as well as parallel implementations in means of energy-efficiency. Figure 1 shows three architectures that provide equal throughput, but differ in chip size and operating frequency. The figure is drastically simplified, but highlights the main architectural differences.

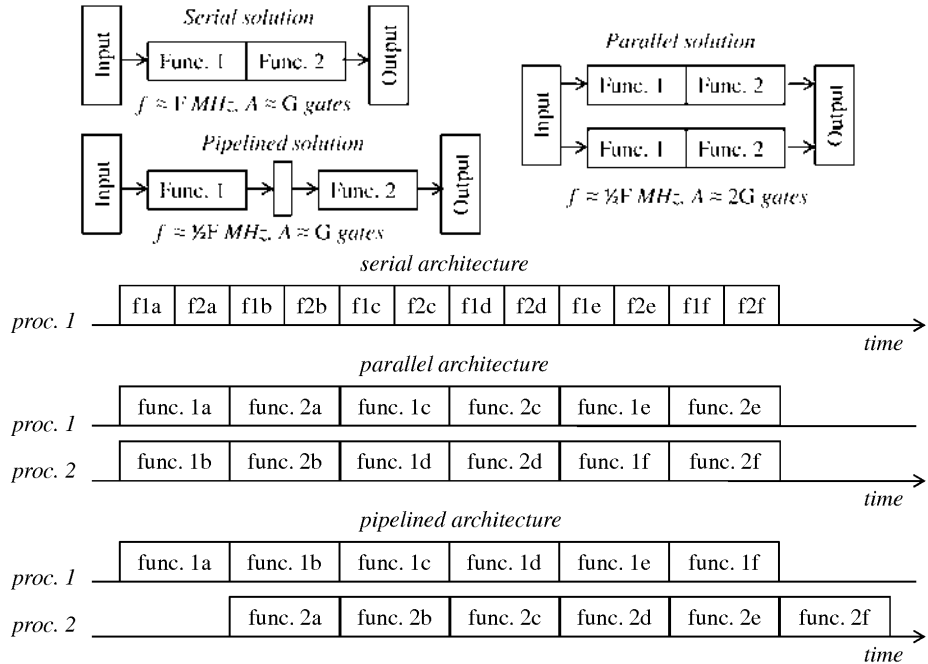


Fig 1. Block- and activity diagrams of three computing architectures that have the same performance, but differing area (A) and operating frequency (f).

In a serial implementation, a single PE performs all functionalities at a high speed, whereas the parallel implementation uses a second, identical PE to provide the same performance with a lower operating frequency. The pipelined implementation breaks down the computational functionalities to separate PEs, which all compute simultaneously, but are dedicated to perform certain tasks. The pipelined implementation represents *temporal parallelism*, and the parallel implementation represents *spatial parallelism* (Benoit *et al.* 2005). In each architecture, input and output, as well as the element

between Func. 1. and Func. 2 in the the pipelined solution, are clocked registers.

Pipelined implementations represent one kind of *heterogeneous* multiprocessing, which has been identified by Kumar *et al.* (2005) as being more energy-efficient than *homogeneous* multiprocessing. Heterogeneous PEs can specialize further into *application specific* processing elements that are a major contributor in achieving energy-efficiency. Unfortunately, even though application specific hardware offers the best energy-efficiency (Denolf *et al.* 2007), it is inflexible by nature. In multifunction devices, such as cellular phones, this causes controversy: dedicated hardware makes computations efficient, but also requires silicon area that can only be used for a specific purpose. As silicon area also contributes to the energy consumption (Kim *et al.* 2003), a balanced solution is required. Figure 2 depicts a coarse-grained hardware accelerator that performs a sequence of video processing tasks with great energy-efficiency. However, the accelerator’s interface does not allow the use of individual accelerator functions.

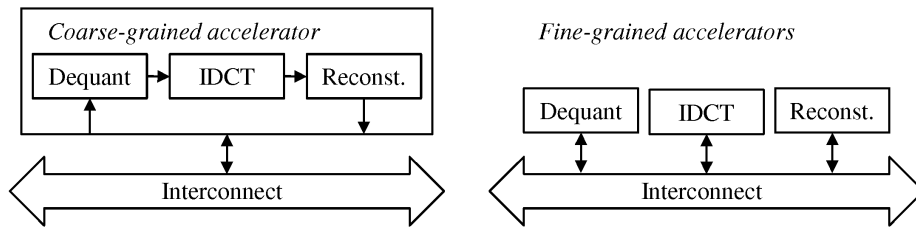


Fig 2. Coarse- and fine-grained hardware accelerators.

An attractive solution is to create fine-grained application specific processing units (Silvén & Jyrkkä 2007) that provide the same functionality as coarse-grained accelerators, but in separately accessible entities (Figure 2). Adopting this solution has beneficial side effects, as smaller circuit entities are faster to verify, which also improves the time-to-market of commercial systems and lowers device design costs (Rintaluoma *et al.* 2006). Unfortunately, these fine-grained processing elements are problematic, since the scheduling (Al-Kadi & Terechko 2009) or synchronization (Rintaluoma *et al.* 2006) of such a processing element can take much longer than the actual useful work that the unit does. Parallel processing poses new challenges to the system designers, both in the application domain (Agerwala & Chatterjee 2005) and hardware domain: the potential performance of a parallel system can be nullified by a carelessly planned synchronization scheme.

1.1 Contribution of the thesis

In this study, we consider the above mentioned parallel processing elements which have a level of granularity above microprocessor instructions, in a context of recently emerging high-performance mobile applications that execute in a piecewise predictable fashion. We will show how a modern video decoding algorithm frequently changes the utilized coding tools depending on the data content, and how the functionality of the baseband processing part of a wireless receiver varies in a piecewise predictable fashion. Adapting to these quickly changing and unpredictable method changes requires flexibility from the decoding system. Figure 3 sketches a situation that is common in video coding: the compressed video stream contains header packets that reveal the required video decoding tasks only after the header packet has been decoded. The application is piecewise predictable, since the system functionality is determined for a while after the decoding of the header.

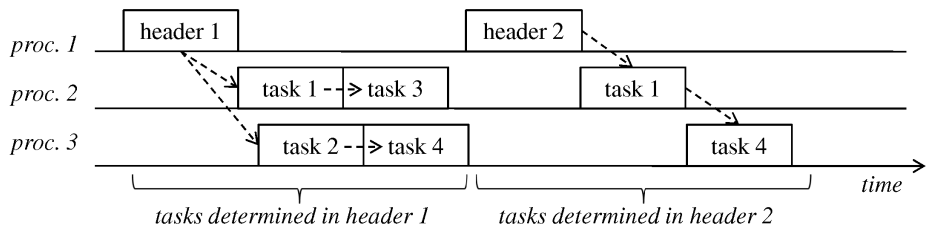


Fig 3. Piecewise predictable processing.

A traditional, but inflexible solution for allocation of processing resources is to allocate time considering the worst case scenario. Unfortunately, the overallocation of resources that ensues from this solution (Cortés *et al.* 2005), is not energy-efficient. The solution proposed in this thesis can quickly adapt to sudden changes of requirements and allocate resources on-demand. Making this possible required studying a new class of simple but efficient scheduling algorithms and simultaneously studying their computational complexity and performance. The requirements and restrictions of this run-time multiprocessor scheduling problem lead to the use of flow-shop scheduling algorithms that have been used in industrial production chain scheduling for decades. A considerable amount of work was done to adapt the flow-shop scheduling algorithms to different multiprocessor memory architectures.

Prior to this work, it was unclear whether it is possible to dynamically use a set of heterogeneous low latency processing elements without introducing a tremendous orchestration and synchronization overhead. This thesis presents as a concluding result a scheduling co-processor that can be used for a couple of different flow-shop variants and works with such low overheads that it becomes evident that the use of run-time scheduled low latency processing elements is worthwhile and efficient.

In addition to several smaller experiments, the feasibility of the proposed approach is demonstrated with a real multiprocessor solution that is implemented on a field-programmable gate array. The efficiency of the proposed solution is compared against a respective system that is statically (worst case) scheduled.

The context of this work is limited to data dominated signal processing systems that consist of multiple heterogeneous processing elements. We assume that all tasks that the system may execute have been assigned to processing elements at system design time. Assigning tasks to processors at run-time, as Kumar *et al.* (2007) and Al-Kadi & Terechko (2009) have done, is left out of the scope of this thesis. Similarly, hard real-time applications are not considered in this thesis. The methods we present, are based on bus based communication or customized datapaths between processing elements. Network-on-Chip (Atienza *et al.* 2008) interconnects are left out of the scope of this thesis. The target systems for our solutions are handheld embedded systems that use batteries as their power source.

The key contributions of this work are:

1. The first work to adapt run-time flow-shop scheduling to multiprocessor digital signal processing (DSP) and different memory architectures
2. Introducing the extended permutation flow-shop scheduling model
3. Proposing a co-processor for flow-shop scheduling and dispatching of fine-grained tasks

The VHDL implementation of the scheduler/dispatcher LSI circuit described in Section 8.2 was written by Alessandro Cevrero of the Microelectronic Systems Laboratory of École Polytechnique Fédérale de Lausanne, Switzerland. The Java implementation of the method described in Chapter 5 has been written by Veeranjanyulu Sadhanala of the Department of Computer Science and Engineering, IIT Bombay, India and Victor Martin Gomez of the Machine Vision Group of University of Oulu.

The contents of this thesis are organized in the following way: Chapter 2 introduces two piecewise deterministic applications that run on mobile devices, and Chapter 3

discusses the mobile embedded devices that are capable of running such applications. Chapter 4 covers modeling and scheduling of data-dominated applications. Chapter 5 shows how a concurrent video decoder model is transformed to be suitable for quasi-static scheduling. Chapter 6 shows the proposed scheduling methods, and in Chapter 7 the methods are adapted to different memory architectures and evaluated through experiments. Chapter 8 shows a co-processor that is dedicated for scheduling and orchestration of heterogeneous processing elements and shows a complete system that uses the proposed scheduling co-processor. Finally, a summary is presented in Chapter 9.

2 Computationally intensive mobile applications

In this chapter, we introduce two signal processing applications that require a considerable amount of computing performance and are commonly run on mobile devices. Mostly, we will concentrate on *video decoding* as it is used as the primary application example throughout this thesis. A short section is devoted for Reconfigurable Video Coding, which embodies the recent trends of signal processing and embedded computing. The second application example is *3G Long Term Evolution baseband processing* that has many similar requirements to video decoding.

2.1 Video decoding

Nowadays, the capability of playing back videos is commonplace on portable devices, and experts foresee that the popularity of portable video will grow considerably in the future. By the end of 2005, 6 million people had subscribed to a mobile TV service, and worldwide the number of subscriptions is expected to rise to 500 million by 2011 (Oksman *et al.* 2008). These numbers do not include the owners of handheld devices that can record and play back on-device video sequences, which is much larger than the number of mobile TV compatible devices.

Since video decoding is used as a main application example throughout this work, we shall now take a look at the structure and concepts of digital video. Although this thesis discusses mostly issues of MPEG-4 video, we shall use general video processing terminology here after first defining their respective names in the MPEG-4 standards (MPEG 2004, 2008a).

The basic entity of digital video is a *stream* that has the name *video object (VO)* in MPEG-4. Streams contain *video object layers* that are used in scalable video coding. Each layer on top of the *base layer* adds details to the video signal, but only the base layer is mandatory for viewing the video. For simplicity, we shall assume from here on that only the base layer is used.

The base layer contains *groups of video object planes (GOVs)* that are also often called *groups of pictures*. Each GOV consists of *video object planes (VOPs)* that are commonly referred to as *frames*. Respectively, each frame is built from 16x16 pixel

macroblocks, that contain several 8x8 pixel *blocks* (Pereira & Ebrahimi 2002). The hierarchy described is depicted in Figure 4.

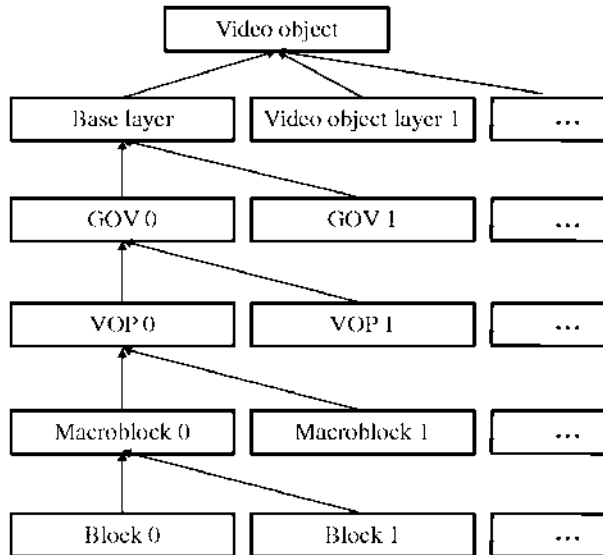


Fig 4. The object hierarchy of MPEG-4 video.

Although the recent MPEG-4 part 10 standard (MPEG 2008a) uses various block sizes, we assume in this work that the block size is always 8x8 pixels and that there are six blocks in one macroblock. The 16x16 blocks that exist in the earlier MPEG-4 part 2 standard (MPEG 2004), are processed as four 8x8 blocks. The number of blocks in a macroblock is dependent on the color coding scheme used. MPEG-4 video uses the YUV colorspace, where the luminance (structure) and chrominance (color) components of the picture have been separated. This colorspace is usually subsampled so that the luminance component is sampled with full resolution, but the chrominance only with half resolution to save on the amount of data, without greatly affecting the visual quality. This also explains why there are six blocks inside a macroblock: the luminance of a macroblock is sampled with four 8x8 pixel blocks and the chrominance with two 8x8 pixel blocks that are enlarged to size 16x16 at the last stages of decoding. The composition of a macroblock is displayed in Figure 5.



Fig 5. From left to right: Y, U and V components and reconstructed macroblock.

A compressed video sequence (stream) is created by *encoding* it to a standard-format *bitstream*. This bitstream is then transmitted to a receiver that has to perform decoding to restore the compressed bitstream to visually understandable frames. The encoding and decoding can take place on the same device or happen in physically separated locations, which requires bitstream transmission over different mediums. In this study, we shall only concentrate on the decoding part of this process.

Table 1. Maximum allowed macroblock decoding times.

Standard	Resolution	Fps	μ s/Macrob.
DVB-H class A (Stabernack <i>et al.</i> 2007)	176x144	15	673
DVB-H class C (Stabernack <i>et al.</i> 2007)	352x288	30	84
DVB-H class D (Stabernack <i>et al.</i> 2007)	720x576	25	25
HDTV 1080p (Kawakami <i>et al.</i> 2007)	1920x1080	30	4

For the embedded system designer, a challenging fact is the variability of macroblock coding in MPEG-4. An MPEG-4 frame of size 720x576 pixels contains 1620 macroblocks and the coding procedure of each of these macroblocks is dependent on the data content of the video stream. For the decoding process, this means that it is not possible to predict how a single macroblock has been encoded, without actually reading the *header* of that macroblock from the stream, where the coding is revealed. Expressed in a simplified fashion, the coding options for each of the six blocks are *prediction*, *texture coding*, or a combination of both. Prediction involves interpolating the block data recursively from one or more neighboring frames, whereas texture coding refers to a JPEG-like transform coding that relies on self-sustained image data. Generally, a combination of both is used: most of the macroblock information is acquired by interpolating the image data from neighboring frames and the remaining differences are encoded as texture data.

Table 1 shows a couple of modern-day examples that reveal how short the macroblock decoding time window is for the decoder. Since the decoding of a macroblock is the longest predictable unit in MPEG-4 video decoding, the macroblock time tells us that for HDTV resolution, the decoding system has to be ready to change the decoding process every 4 microseconds. Furthermore, this means that decoding subtasks need to be invoked over 5.8 million times a second, assuming that decoding a block requires four subtasks and the color coding scheme is YUV420. The time durations in the table assume that all blocks have to be decoded, and that each block has an equal amount of time available.

Efficient implementation of high-resolution video decoding can be accomplished by the use of dedicated hardware components (Stabernack *et al.* 2007). If the decoder is designed as one monolithic hardware accelerator, the design process will become expensive and time consuming, as well as resulting in an inflexible solution (Stabernack *et al.* 2007, Silvén & Jyrkkä 2007). Alternatively, designing a solution by implementing the computationally demanding parts in hardware that is controlled by an instruction processor, produces a result that is a tradeoff between monolithic hardware and full-software designs in terms of design time, design cost and energy-efficiency.

Table 2. MPEG-4 ASP decoding subtasks and their execution frequencies as listed in Stolberg *et al.* (2001). Numbers represent a sum of all subtask variants.

Subtask	Frequency
Inverse quantization	112000 coeffs/s
Inverse AC/DC prediction	15000 blocks/s
Inverse discrete cosine transform	61000 blocks/s
Motion compensation	322000 blocks/s
Image reconstruction	229000 blocks/s
Deblocking	243000 blocks/s

Stolberg *et al.* (2001) have studied the structure of a few representative MPEG-4 video sequences. A 1.5Mbit/s video stream had a resolution of 720x576 at 25 frames per second. In the worst case, this requires decoding 243000 blocks in a second, where each block needs to go through several subtasks before it is completely decoded. These subtasks are listed in Table 2. The *variable length decoding* task and *motion vector decoding* are not listed in the table. The subtasks listed in Table 2 emerge in different combinations and variants for each block that is to be decoded. The complete list of

decoding procedures is listed in the work of Stolberg *et al.* (2001).

Finally, it is necessary to introduce the various *profiles* of MPEG-4 that define which level of coding complexity is used for (de)compressing video. In terms used by MPEG-4 part 2, the most basic compression methods are implemented in the *Simple Profile* (SP) that is also suitable for mobile devices (Pereira & Ebrahimi 2002). The next level of complexity is called the *Advanced Simple Profile* (ASP), and it can improve the compression ratio up to 50% compared to the Simple Profile. This improvement is achieved by more advanced coding tools: quarter-pixel accuracy motion estimation, global motion compensation and bidirectionally predicted frames (Chien *et al.* 2003). The profiles are further divided into *levels* of coding. Levels are used to restrict the video resolution and bit rate used.

2.1.1 Reconfigurable Video Coding

The previously described MPEG-4 is only one of many existing video coding standards. VC-1 (SMPTE 2005), H.263 (ITU-T 2005), MPEG-2 (MPEG 2006) and AVS (Fan *et al.* 2004) are some of the latest, or most well known alternatives. Although the standards have been developed at different times, and by different groups, many of the coding methodologies used are in fact very similar.

Based on the observed similarity, work has been started to develop a new ISO standard named *Reconfigurable Video Coding* (RVC) (MPEG 2008b). The objective of this upcoming standard is to provide a unified toolset to do the decoding of video sequences, independent of the compression method used.

In RVC, advantage is taken of the observed similarities between the codecs: each decoding standard is described with a set of decoding functions that have been implemented in advance and stored in a *video tool library* (MPEG 2008c). Similar functional level reuse has also been proposed previously for multi-mode systems (Oh & Ha 2002) and in object-oriented programming (Booch 2007).

RVC is written in the high level CAL language (Eker & Janneck 2003) and tools are being developed to transform the RVC model into more implementation friendly formats: C language (Wipliez *et al.* 2008) and hardware description (Janneck *et al.* 2008). Because of its exemplary design, the RVC implementation of the MPEG-4 SP video decoder (See Figure 6) is referenced often in this thesis. Unfortunately, the RVC workgroup has not released other decoders at the time of writing and because of this, the block sizes and block counts of decoding are assumed according to the MPEG-4 Simple

Profile limitations.

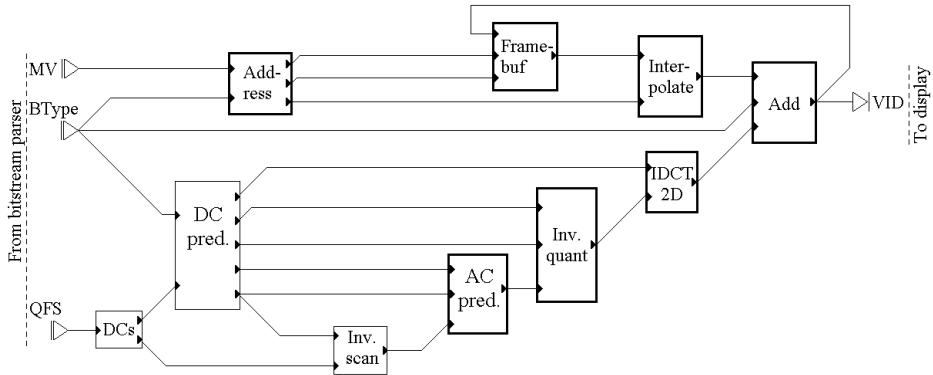


Fig 6. The RVC implementation of the MPEG-4 Simple Profile video decoder. (Boutellier *et al.* 2009c, published by permission of Springer.)

Implementing an energy-efficient RVC-compatible decoder is a challenging task because of the great flexibility offered by the upcoming standard. Extreme energy-efficiency generally requires the use of inflexible hardware components, whereas full RVC-compatibility requires very fine-grained coding tools: the tools have been defined as actors that sometimes perform small, individual operations, such as integer clipping that might be invoked in arbitrary patterns in decoders that are implemented in RVC in the future. Besides requiring flexibility at the time of configuring the decoder, even the simple MPEG-4 SP decoder implemented in RVC, requires run-time adaptability from the system, in the form of different block decoding patterns. This is a most clear application example that requires solutions such as the one presented in this thesis: methods for how to use fine-grained, dedicated processing elements with flexible low overhead run-time scheduling.

2.2 Wireless baseband processing

Video processing is not the only application that requires both throughput and flexibility from embedded systems. In fact, a third generation (3G) 14.4 Mbps mobile phone receiver can require 35 to 40 giga operations per second (GOPS), which is well over the 6-10 GOPS requirement of MPEG-4 part 10 encoding that represents the current state-of-the art in video coding (Silvén & Jyrkkä 2007).

The 3G Long Term Evolution (LTE) standard is under development and will support data rates of up to 100 Mbps (Salmela *et al.* 2009). The 3G LTE standard uses the orthogonal frequency multiplexing (OFDM) transmission technique with multiple input multiple output (MIMO) that is based on using several transmitter and receiver antennas.

The transmitter encodes the data to be transmitted to several partially overlapping frequency bands and uses an inverse fast Fourier transform (IFFT) to transform the signal to time domain before it reaches the transmitter antennas. The receiver antennas acquire the transmitted signal and transform it back to parallel frequency bands through FFT. Through estimated channel parameters of each used frequency band, the receiver needs to detect the actual data from the received signal, as it has been distorted on the way to the receiver.

Salmela *et al.* (2009) and Lattard *et al.* (2008) propose using multiprocessor systems to perform the numerous computations that are required in the receiver. Lattard *et al.* (2008) state that only a small amount of 3G LTE baseband processing can be performed on general purpose processors or DSPs, which means that dedicated hardware is required. The solution proposed by Salmela *et al.* (2009) includes the use of four dedicated application specific processors of the TTA type (see Section 3.2 for more information).

The computational blocks in the solution by Salmela *et al.* (2009) are based on four major computations: FFT, QR-decomposition, list sphere detection and turbo decoding. FFT has to be performed separately for the data of each antenna and needs to be combined before the later stages. A block diagram of this system is shown in Figure 7.

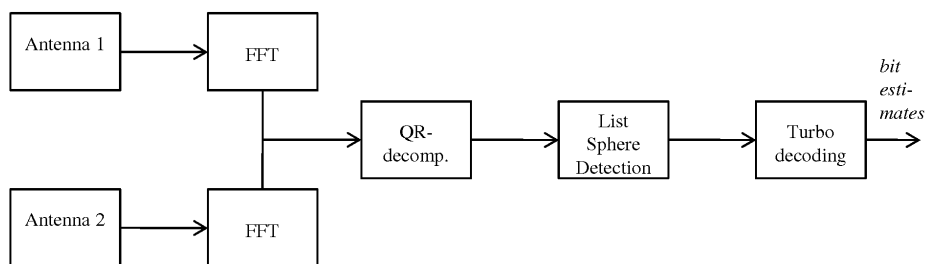


Fig 7. The computationally most burdening blocks of a two-antenna MIMO-OFDM receiver, adapted from the work of Salmela *et al.* (2009).

The QR-decomposition, list sphere decoding and turbo decoding implementations of Salmela *et al.* (2009) can be considered fine-grained hardware PEs, as the latency of

each function is less than 441 clock cycles (Salmela *et al.* 2009) and can be computed in parallel for each frequency band. However, the reception process also requires rapid adaptation to changing channel conditions, as QR-decomposition is only required when channel conditions change. Thus, an efficiently implemented receiver also requires run-time scheduling of the PEs. Static PE schedules also work, but perform unnecessary work with the QR-decomposition that does not need to be invoked as often as the other blocks.

2.3 Summary

The two applications presented that run on mobile devices, video decoding and 3G wireless baseband processing, pose two challenging requirements on the mobile system. First, the required computational performance starts from several GOPS, and is likely to grow fast with future standards (Silvén & Jyrkkä 2007). Second, both applications use adaptive coding that requires different functions to be executed by the receiver device, based on the data content that is unknown at system design time. In the next chapter, embedded system solutions for meeting these requirements are discussed.

3 Mobile devices for computationally demanding applications

Patterson & Hennessy (2005) state that embedded systems "are designed to run one application or one set of related applications, which is normally integrated with the hardware and delivered as a single system." This sentence expresses the essence of embedded systems from elevators to mobile phones.

The core of an embedded system is the processor which is often different from general purpose processors that can be found in desktop computers. The two most frequently used kinds of embedded processor are the RISC (reduced instruction set computer) and the DSP (digital signal processor). RISC processors contain a limited set of instructions that can all be executed in one clock cycle, which makes speedup through instruction *pipelining* easy. DSPs on the other hand, contain specialized and powerful domain specific instructions, such as multiply-accumulate (Wolf 2006).

Although a single embedded processor can have sufficient performance even for H.264 video decoding (Huang *et al.* 2008), multiprocessor solutions are favored in computationally intensive embedded applications, since they can provide better energy-efficiency (Kumar *et al.* 2005). The individual processors in an embedded system need not be programmable. Thus, throughout this thesis we shall call these units that perform computations by the name: *processing element* (PE). The concept PE is used for all kinds of processors, whether they are programmable processors, application specific processors, or hardwired application specific circuits.

From the designer point of view, embedded systems differ from general purpose computing systems by the fact that besides software, the system hardware is also a major design parameter. In general purpose computing, hardware is assumed to be fixed or transparent, whereas the embedded system designer needs to choose, and possibly even design, the hardware architecture according to the application requirements (Berger 2002).

Designers of embedded systems have to cope with many restrictions that do not exist for software designers of desktop systems, such as the previously mentioned energy-efficiency. Fortunately, embedded system designers also have more freedom in achieving these goals, such as the possibility of implementing dedicated hardware for a particularly demanding task. The task of selecting what is implemented in hardware, and

what is left over to software is known as the problem of hardware/software partitioning (Wolf 2003).

The related field of research that studies the problem of designing simultaneously the hardware and software is called *hardware/software codesign*. One of the key methodologies of HW/SW codesign (and partitioning) is to analyze the application, and decide which parts of the application are executed on dedicated custom hardware and which are left for software execution on a programmable processor. A simplified view of the architecture of an HW/SW codesign system according to Wolf (2003), can be seen in Figure 8.

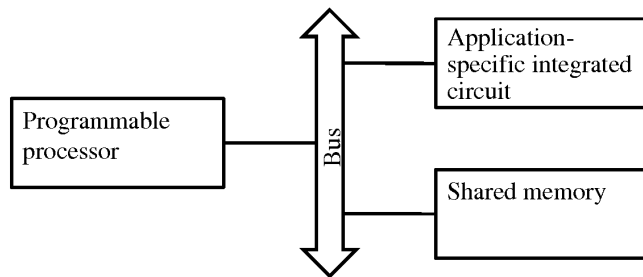


Fig 8. A generalized target architecture for embedded hardware/software codesign.

The codesign architecture of Figure 8 is simplified in the sense that real architectures may contain more than one programmable processor or application specific integrated circuit (ASIC). Wolf *et al.* (2008) describe several state-of-the art multiprocessor system-on-chip (MPSoC) architectures that contain more than four PEs.

MPSoC systems can be divided to two major classes: homogeneous and heterogeneous. A homogeneous MPSoC contains several identical processors, whereas a heterogeneous MPSoC contains at least one PE that differs from the others. Heterogeneous systems can be further divided into single instruction set architectures (ISA) and multi ISA MPSoCs. With a single ISA architecture all processors can execute the same instructions, although with different performance. The PEs on multi ISA architecture on the other hand, do not support the same instruction set (Kumar *et al.* 2005).

Kumar *et al.* (2005) state that heterogeneous PEs can provide a major improvement in power and throughput. This thesis also advocates the use of heterogeneous MPSoCs for computationally intensive applications. The architecture examples that are shown later in this study contain a single instruction processor along with several dedicated, heterogeneous PEs.

The *stream processors* Imagine (Ahn *et al.* 2004) and Storm-1 (Khailany *et al.* 2008) represent homogeneous multiprocessing. These stream processors utilize the data parallelism offered by streaming applications (such as video decoding) through several parallel data lanes that contain a very long instruction word (VLIW) processing element. Each VLIW lane executes an identical series of VLIW instructions on stream data in a Single Instruction, Multiple Data (SIMD) manner. In stream processing terminology, the PEs run *computational kernels*, and the data is described as streams that flows through several kernels that perform computations with the data. The computational kernels are implemented VLIW code that is executed on the VLIW PEs. These two stream processors offer a high throughput and are programmable. Homogeneous multiprocessors suitable for portable devices are discussed in the work of (Kaxiras *et al.* 2001).

3.1 Energy-efficiency

In Chapter 1, we outlined the major requirements that markets have from mobile consumer devices. The requirements that mostly affect the work of designers are device cost, time-to-market, performance, battery lifetime and versatility.

Battery lifetime is probably the most important criterion of these requirements (Ferri *et al.* 2008), since practically all designer choices affect the power consumption of the system. Therefore, it is extremely important for the designer to be aware of the factors that affect power consumption in an embedded system.

From an electric circuit point of view, the power consumption can be divided into two major categories: dynamic and static power consumption. Dynamic power consumption takes place when the embedded system toggles the states of transistors by arithmetic, memory accesses and the like. Static power consumption, on the other hand, happens independent of the system activity, through leakage currents in the silicon of the integrated circuit (Kim *et al.* 2003).

Years ago, dynamic power consumption used to be the main source of circuit power usage. Progress in digital circuit technologies has changed the ratio, and nowadays static power usage has also become very significant. This is mainly caused by the increasing chip density and thus thinner insulations between wires, which again leads to higher leakage currents. For the embedded systems designer, this means that achieving good energy-efficiency requires keeping the silicon area of the chip small. (Kim *et al.* 2003). However, very recently static power usage has been limited by the *High-K + Metal Gate*

technology (Rahal-Arabi & Park 2008).

In high-performance computing, thermal issues have gained a great deal of attention in recent years. The power dissipated by a microprocessor chip per unit of area, is growing steeply as the transistor densities increase (Borkar 1999). However, recent studies have shown that for low power MPSoCs (such as the ones found in mobile phones), thermal issues should not pose a problem. This is working under the assumption that the MPSoC operating frequencies are below 500MHz, no three-dimensional packaging or *Low-K dielectrics* are used (Paci *et al.* 2007).

As the embedded system designer notices that the application requires higher throughput than the system can offer, the traditional solution has been to increase the clock frequency and complexity of the central processing unit of the system. Unfortunately, this approach has severe limitations in the embedded domain, because the energy-efficiency of the system decreases rapidly as the clock frequency is increased (Kim *et al.* 2003). This is the main motivation for multiprocessor solutions becoming popular in high-performance embedded systems (Horowitz *et al.* 2005).

In terms of energy-efficiency, heterogeneous MPSoC systems overcome homogeneous systems (Wolf 2004), especially when some of the PEs are dedicated *hardware accelerators* (Denolf *et al.* 2007). A hardware accelerator is a highly customized integrated circuit that is capable of performing a limited set of computations with extreme efficiency. The tools provided by the previously discussed co-design approach help the embedded system designer to decide which parts of the application should be mapped as hardware accelerators and what should be left to the programmable processors.

Application specific instruction processors (ASIPs) present an intermediate solution between hardware accelerators and general purpose programmable processors. ASIPs can offer customized instruction sets, as well as a processor architecture that has been tailored for the application (Jain *et al.* 2001). The application specific instructions can also be derived automatically by analyzing the application code, as Pozzi *et al.* (2006) have done.

In the work by Rabaey (2000), a diagram is presented where dedicated hardware, ASIPs and general purpose-processors are placed on an energy-efficiency – flexibility scale. Apart from the three solutions that we have already described, the work of Rabaey (2000) mentions reconfigurable processors that he places between dedicated HW and ASIPs in the energy-efficiency – flexibility scale. The work by Todman *et al.* (2005) presents an overview of the different architectures and design methods of reconfigurable

computing. In this thesis, a field-programmable gate array (FPGA) is used as a testbed to perform experiments. In our experiments, the FPGA is programmed at the time of design and it remains fixed at run-time. However, reconfigurable computing also provides the possibility to change the logic configuration at run-time. Although this would enable interesting possibilities also in the context of this thesis, we shall leave it out of the scope of this work. The energy-efficiency of FPGAs is rather poor: a relatively recent study (Kuon & Rose 2007) reveals that the dynamic power consumption of an FPGA is around 10 times higher than that of an ASIC, and the passive power consumption is even worse.

Since mobile embedded systems may also contain programmable microprocessors, it is useful to see how microprocessor activity consumes power. Tiwari *et al.* (1996) experimented on three processors that had varying architectures: CISC, RISC and DSP. The first observation they made was that the application execution time correlated strongly with its energy usage, such that a program that performed the same task faster than another program consumed also less energy. The second interesting observation was that instructions requiring memory accesses were generally more power consuming than those relying on register accesses.

The architectures handled in the work of Tiwari *et al.* (1996) originate from a time when system memory architectures generally consisted of a main memory and a cache. Nowadays, an alternative to caches is the use of scratchpad memories (Steinke *et al.* 2002). A scratchpad memory works like a compiler directed cache, and it has been reported to improve the energy-efficiency by around 20% when compared to cache based memory solutions. The use of a scratchpad memory evidently alleviates the energy consumption difference between memory addressing and register addressing operations.

Balfour *et al.* (2008) present a programmable processor architecture that is 23 times more energy-efficient than an embedded RISC processor. The solution is based on an exposed, compiled directed datapath. On the efficiency-flexibility scale this approach is more on the flexibility side than the one presented in this thesis. This is due to the fact that their architecture processes instructions, instead of relying on longer latency hardwired circuits, as our approach advocates.

3.2 Time-to-market and design cost

Besides energy-efficiency, embedded system design has also other goals. For the companies that produce the actual end-user devices, it is important that the design cost

is kept as low as possible and time-to-market demands are met.

Due to the high complexity and susceptibility to risk of designing new hardware (Berger 2002, Stabernack *et al.* 2007), it is advisable to implement as much as possible of the embedded system functionality in software. For the parts that necessarily need to be implemented in hardware, it is essential to keep the design simple. Especially for hardware, the design and verification times grow rapidly as the chip area increases.

Design re-usability has become a major objective in modern-day chip design (Seidel *et al.* 2005). It offers a solution to decreasing system design times as well as chip area. Hardware re-use can be done in two different ways: between systems and within the system. Hardware re-use between systems involves including previously fabricated task-specific components (IP-blocks) in the system (Vermeulen *et al.* 2000, Horstmannshoff & Meyr 1999). Another way of re-using hardware is to utilize one hardware component in several different applications within the same device (Oh & Ha 2002, Chen *et al.* 2007), which is common in *multi-mode* systems. A multi-mode system is a device that runs several different applications on the same platform by reconfiguring its functionality (Oh & Ha 2002). Kumar & Lach (2006) have implemented multi-mode resource sharing in fine granularity: they reconfigure the functionality of arithmetic components (adders and multipliers) and state machines according to the system operating mode.

The re-usability of a component is closely related to its generality: a component that implements a short, general purpose function is more likely to be useful in different systems and applications than a very specialized one. The previously introduced RVC standard (Lucarz *et al.* 2007) embodies this tendency: several different video decoding standards (applications) can be designed out of a set of common components that each are of relatively small size.

As the design cost of hardware is much higher than that of software (Berger 2002), researchers have developed several *high-level synthesis* methods to reduce the effort of hardware development (Arvind *et al.* 2004). In high-level synthesis the hardware component is described in a behavioral description (Wolf 2006) and then transformed into hardware through several automated steps, which is much a faster way than programming in a hardware description language like VHDL or Verilog. RVC also represents a form of high-level synthesis, since the application components are described in a high-level language called CAL (Eker & Janneck 2003) that can then be compiled into a hardware description by a separate tool (Janneck *et al.* 2008). As a drawback of high-level synthesis, it is necessary to mention that generally handwritten hardware descriptions produce more compact designs than outputs of high-level synthesis tools.

Another paradigm that alleviates the hardware design effort is the TCE toolsuite (Jääskeläinen *et al.* 2007) that is used to produce Transport-Triggered Architecture (TTA) processing units. TTA is a very interesting possibility for implementing dedicated co-processors for high performance MPSoCs, especially for multi-mode embedded systems. Processors created according to the transport-triggered architecture consist of a set of functional units (FUs) with an exposed, compiler directed interconnect (see Figure 9). This enables reconfiguring the TTA processors according to different application needs. Furthermore, the exposed interconnect enables efficient, compile time scheduling of the functional units. TTA processor design and verification times are also modest, since the separate FUs can be generated and tested independently of each other.

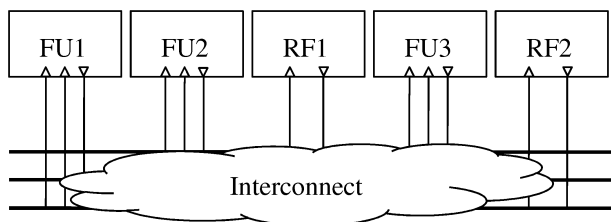


Fig 9. A TTA processor with two register files (RF) and three functional units (FU).

It is easy to add special function units (SFU) to the TTA interconnect, which produces a link between the topic of this thesis and the TTA paradigm. The SFUs can be interpreted as fine-grained hardwired PEs that are connected by a programmable interconnect. Section 7.1 discusses this issue in somewhat greater detail.

Interestingly, the methods of achieving good energy efficiency, fast time-to-market and low design cost overlap to some extent: architectures that consist of several fine-grained PEs provide the potential for parallel processing, as well as modest unit design times and verification times.

3.3 Multiprocessor synchronization issues

One of the conclusions of the previous section was that parallel processing is one of the key methodologies in designing energy-efficient embedded systems. Unfortunately, parallelism makes embedded system design considerably harder. Even simple sounding issues like the communication between processing elements need special attention to achieve high throughput and correctness in multi-PE solutions.

Wolf (2003) states that one of the key problems in architectures like the one shown in Figure 8, is the communication between the ASIC and the programmable processor: performance gains offered by the efficient IC can be nullified by communication and synchronization delays. Evidently, this problem can also be generalized to systems with more than two PEs.

As the central contribution of this work is closely related to multiprocessor synchronization and communication, we will now introduce traditional solutions to these issues and show their shortcomings.

3.3.1 Direct processing element interfacing

From the many different ways of organizing the interplay between PEs in multiprocessor systems, we shall start by looking at one of the most simple possibilities.

In a parallel processing system, a traditional way of utilizing parallelism is to select one of the PEs as a central processing unit (CPU), and assign the remaining PEs (we shall call them *slave units*) to follow the instructions of the CPU, as in the solution by Stabernack *et al.* (2007). By default, the slave units are in idle state and wait for instructions from the CPU, which acts as a *master*. Once the master sends a start signal to a slave unit, the slave unit begins to compute whatever it has been assigned to do. When the slave unit finishes, it sends a signal back to the master to inform it that the results are available.

Nedjah & de Macedo Mourelle (2007) describe two approaches to implementing this functionality. They name the first approach *busy-wait* (also known as *polling*), where the master starts the slave PE and goes into a polling loop, and constantly reads the *busy* status of the slave PE. Once the busy-state ends, the master knows that the results are ready and can proceed immediately. The drawback of this solution is evidently that the master cannot perform any useful computations while waiting, and it cannot even go into a power-saving mode. If the structure of the application running on the multiprocessor system permits, the master should perform useful computations while waiting for the slave unit to finish its work.

By changing the synchronization paradigm to an interrupt-based approach, the master can be enabled to work while the slave PE is processing. The *ready* signal line from the slave unit is connected to an external interrupt port of the master, and thus the signal sent by the slave makes the master interrupt its work and jump to an interrupt service routine (ISR). In the ISR, the master searches the interrupt source, reacts to

the signal and then continues its work. Nedjah & de Macedo Mourelle (2007) report that their design spends 180 clock cycles in the process that starts from reacting to the interrupt signal. This delay is clearly a downside of the interrupt approach when compared to busy-wait.

Nedjah & de Macedo Mourelle (2007) report that interrupt based interfacing has provided better results for the shared memory intensive type of applications, whereas the busy-wait mechanism provided better speedup with an application that had an emphasis on slave PE-internal computations. Hardware assistance (Ferri *et al.* 2008) improves the performance of the busy-wait approach, as does having a low power state in the master processor (Golubeva *et al.* 2007). On the other hand, the interrupt based approach is not advisable if the slave PEs are invoked often. Each slave PE invocation results in almost two hundred wasted clock cycles, as Nedjah & de Macedo Mourelle (2007) reported. Moreover, in embedded systems that have a cache, the cache contents are polluted because of the jump to the ISR (Silvén & Jyrkkä 2007).

Other synchronization mechanisms include *barriers*, *monitors* and the relatively new *transactional memories* (Herlihy & Moss 1993). In the work of Ferri *et al.* (2008), the energy efficiency of traditional synchronization methods are compared against transactional memories, and it is shown that transactional memories do provide an advantage over traditional approaches, on average.

In the two previous sections, we have discussed how efficient implementations of computationally intensive mobile applications would require numerous PEs that implement short latency functions. This also applies to the previously introduced, upcoming Reconfigurable Video Coding standard. As video decoders are often multi-processor systems with several task specific PEs (Schumacher *et al.* 2005, Wang *et al.* 2005, Liu *et al.* 2007), it is evident that an efficient implementation of RVC can also be based on several task specific PEs. At the same time, the reconfigurability in RVC requires that if video processing functions are implemented in hardware, they need to be fine-grained (*i.e.* short latency) to maintain support for various configurations.

Another less obvious problem also arises with the RVC example: in reality it is desirable to *pipeline* the PEs (Ling & Wang 2003, Wang *et al.* 2005) so that maximally many units are performing computations at any moment. None of the presented simple slave PE access schemes define a systematic way of pipelining the operation of several slave PEs. In the next subsection, we elaborate the communication coordination from two PEs to several PEs.

3.3.2 Offline scheduling of parallel PEs

At the end of the previous subsection, the analysis of simple synchronization schemes resulted in a problem: an application can require initiating repeated and simultaneous computations on several slave PEs that perform short latency tasks.

Methods that can solve problems like this require the use of *multiprocessor scheduling* algorithms. Scheduling means planning ahead the timing of several events so that the constraints set by the environment are not violated. Scheduling always involves some kind of optimization. A common optimization objective is *makespan* optimization that means aiming for the shortest possible overall schedule length in terms of processing time (Balarin *et al.* 1998).

From the energy efficiency perspective, the most efficient scheduling algorithms are *fully static*, which means that the schedule is computed at system design time, as Ling & Wang (2003) have done. The run-time execution of a static schedule only requires the timely triggering of the scheduled events. A good source of information for implementing a static scheduling algorithm for multi-PE systems is the work of Sriram & Bhattacharyya (2000).

The designer of a multiprocessing system, consisting of a master and several fine-grained slave PEs, must also pay attention to the behavior of the system at run-time. When the slave PEs use interrupts to signal to the master that the slave PE task has finished, the master can become overwhelmingly burdened with the interrupts (Gangwal *et al.* 2001). As interrupt service routines consume a couple of hundred master PE clock cycles (Rintaluoma *et al.* 2006, Nedjah & de Macedo Mourelle 2007) and HDTV decoding, for example, requires invoking several millions of tasks each second (see Section 2.1), we are speaking of a magnitude of a gigacycle that is lost in interrupt overheads.

As a remedy to this shortcoming, Silvén & Jyrkkä (2005) and Rintaluoma *et al.* (2006) suggest assuming that the slave unit latencies are deterministic or fixed. The master does not need a separate interrupt from the slave unit, when it is known beforehand when the results are available. Large scale integration (LSI) circuit implementations of a variety of video decoding functions, such as inverse discrete cosine transform (Rambaldi *et al.* 1998) and interpolation (Dang 2006) clearly have a fixed latency and are well-suited to the assumption of a deterministic execution time. Tasks with a great, unpredictable variance in their latency, such as context adaptive variable length decoding, where the worst case latency is almost twice the average latency (Alle *et al.* 2006), pose

a slight problem. If such tasks need to be executed amongst other statically scheduled tasks, their worst case latency must be always assumed (Lee & Ha 1989).

With the combination of a static schedule and static task latencies, it is possible to use a set of fine-grained slave units efficiently for applications that use the slave PEs in regular patterns. However, in Section 2.1 the description of MPEG-4 decoding revealed that the decoding procedure can vary from decoding zero to six blocks for each macroblock. With a fully static schedule, the worst case of six blocks must always be assumed, which leads to a degradation of throughput and energy-efficiency when the real number of blocks to decode is less than six. Cortés *et al.* (2005) have also detected that fully static schedules offer a low quality result in dynamic systems.

Run-time scheduling provides the flexibility required by applications that have data dependent workloads. These methods are discussed next.

3.3.3 Run-time scheduling

Run-time scheduling of processing elements is required when the usage of PEs cannot be resolved at design time and the PE usage patterns change considerably as the application executes. Run-time scheduling, like any run-time decision making, should be avoided in embedded systems whenever possible because it consumes more resources than static solutions.

Run-time scheduling requires additional computation time, hardware, or both. If scheduling is performed on the system CPU, the time used for scheduling takes its share from the processor worktime. Alternatively, a dedicated scheduling unit can be designed to run in parallel with the CPU. A dedicated scheduler PE consumes silicon area, whereas software scheduling on the CPU requires its share of computation time (Cho *et al.* 2007). Thus, both of the solutions affect the energy consumption of the system.

If it is necessary to implement run-time scheduling functionality into an embedded system, a maximally simple scheduling algorithm should be chosen. It requires careful studying of the scheduling problem to find out what scheduling decisions really are required and what is redundant. Scheduling is always some kind of optimization and in many cases the problem is NP-complete. This means that all but the smallest scheduling problems cannot be exhaustively solved if time constraints exist. Since the optimal solution is generally not reachable, *heuristics* are often used to acquire *good* solutions in reasonable time. As scheduling has already been studied for decades, there

are many different heuristics that can be chosen to solve the problem at hand. Choosing the heuristic means balancing the tradeoff between the complexity of the heuristic and the solution quality. Generally, algorithms that run for a longer time, produce a better quality result. For solving compile time scheduling problems, the algorithm execution time is usually meaningless, and the heuristic is chosen on the basis of solution quality it offers. In run-time scheduling the algorithm execution time cannot be dismissed, since it affects the energy consumption and throughput of the system.

Figure 10 shows a rough taxonomy of run-time multiprocessor scheduling algorithms. Starting from the root node, the scheduling algorithms are divided into dataflow oriented and control oriented methods, according to the division by Balarin *et al.* (1998). Control oriented scheduling methods are used in hard real-time embedded systems, where strict task execution deadlines exist. Control oriented algorithms can be divided into dozens of categories based on the requirements of the system and application. Here, we only mention the work of Ramamritham *et al.* (1990), since it has similar objectives to the problem presented in this thesis. However, we shall not further discuss control oriented scheduling, since the application examples of our work are clearly data oriented. Filling the requirements of hard real-time systems consumes a lot of run-time resources and should be avoided if not necessary (Sriram & Bhattacharyya 2000, Xu & Parnas 2000).

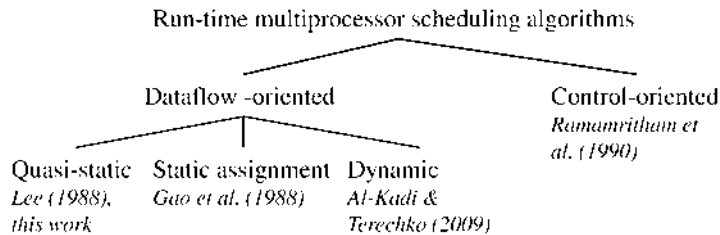


Fig 10. A taxonomy of run-time multiprocessor scheduling algorithms

The dataflow-oriented methods are further classified on the basis of their flexibility. In the work of Sriram & Bhattacharyya (2000), scheduling algorithms are divided into six categories based on their run-time overhead and flexibility. Since we are considering run-time scheduling in this section, we only consider those approaches that are flexible enough to adapt to dynamically changing program execution paths. The *dynamic* scheduling approach of dataflow models makes all scheduling decisions at run-time, and thus provides maximum flexibility, as does the hardware scheduler of Al-Kadi & Terechko (2009). However, for the scheduling of fine-grained hardwired PEs, the

dynamic approach offers even too much flexibility, since we assume that the tasks to be scheduled have been assigned to certain PEs at design time. By adding this constraint, the number of run-time decisions becomes smaller, and thus leads to the *static assignment* category of algorithms. According to Lee & Ha (1989), this leaves the decisions of *ordering* and *timing* of task executions to run-time, as it is done in the work of Gao *et al.* (1988).

In this thesis, algorithms of the static assignment category are briefly mentioned in Chapter 6, but mostly we shall discuss *quasi-static* solutions. Sriram & Bhattacharyya (2000) define quasi-static solutions as follows: "A quasi-static scheduling technique tackles data-dependent execution by localizing run-time decision making to specific types of data dependent constructs such as conditional, and data dependent iterations." In other words, quasi-static scheduling can take into account the data dependent paths of execution that are present in MPEG-4 macroblock decoding and 3G LTE baseband processing. On the other hand, tasks are assigned to processors already at design time, which reduces the number of run-time computations. Quasi-static scheduling was pioneered by Lee (1988) and developed further by Bhattacharya & Bhattacharyya (2000).

Cortés *et al.* (2005) have created an interesting mixture of quasi-static scheduling of control oriented systems. They also concentrate on the effort of deriving quasi-static multiprocessor schedules for achieving low run-time overhead, but the major difference with our work is that Cortés *et al.* (2005) actually work with hard real-time systems that have periodic tasks. According to the classification of Balarin *et al.* (1998) this puts their effort to the category of control centered scheduling. The work is significant, as quasi-static scheduling has traditionally been associated with data oriented processing. The authors also state, as do we, that a fully static schedule for multiprocessors is too pessimistic and it reduces the performance of the system. In their example application, which is related to vehicle cruise control with collision avoidance, they report that a quasi-static schedule produces a gain of 40% in the timeliness of tasks, when compared to a fully static schedule.

3.4 Conclusion

In this chapter, we have analyzed the requirements that computationally demanding applications pose on mobile embedded devices on the system level. As a tight energy budget is the most demanding constraint in a mobile device, all designer choices must

keep this in mind. Previous research indicated that parallel processing is essential for energy-efficiency, as is the use of heterogeneous (possibly hardwired) PEs. The requirements of flexibility, reasonable design times and low design cost require that the dedicated hardware is designed as fine-grained units.

An energy-efficient system that consists of several fine-grained PEs requires a scheduling algorithm for coordinating the mutual transactions of the PEs. Schedules that are determined at design time are not flexible enough for data dependent signal processing applications such as MPEG-4. On the other hand, energy-efficiency requires that as many scheduling decisions as possible are fixed at design time. These requirements lead to the use of quasi-static scheduling algorithms that provide flexibility with a modest overhead. In the next chapter we shall look at application modeling that is related to the quasi-static scheduling methods that are proposed in this thesis.

4 Representing and scheduling multiprocessing applications

Generally, applications and algorithms have not been designed for multiprocessor execution. For humans, it is more natural to express a set of operations in a sequential manner instead of considering which algorithm parts could run in parallel, and which not. Sequential programming languages such as *C* do not even provide a way for the programmer to express that computations could be executed in parallel.

If an already existing algorithm or application has been designed in a sequential fashion, it requires a considerable effort to parallelize it and generally a great deal of manual work is involved. For *C* and other sequential languages, researchers have designed analysis algorithms that try to expose the parallel nature of computations from already written sequential code (Saito *et al.* 2000). The success of these methods varies from case to case, but clearly the use of a sequential programming language is not a recommended starting point for programming parallel systems.

Jerraya *et al.* (2006) list several traditional parallel programming languages (*e.g.* Occam and MPI), but unfortunately all of the languages mentioned only take to account the software side of a parallel system. Jerraya *et al.* (2006) argue that enabling concurrent (rapid) design of hardware and software in an MPSoC system requires that the interface between hardware and software needs to be abstract in the programming model, which is evidently not possible if the programming model does not consider hardware at all. One of the more recent solutions to this is OpenCL (Munshi 2008) which has been developed because programming heterogeneous architectures with conventional programming models is difficult (Udupa *et al.* 2009).

The Reconfigurable Video Coding standard has been designed with multiprocessor systems in mind, right from the beginning. In cases like this, it is easy for the embedded system designer to use the potential of multiprocessing platforms. In this thesis, we assume from now on that the program to be executed on the multiprocessor system is already expressed in a way that exposes the parallelism in the program.

4.1 About Models of Computation

Related to programming languages, it is necessary to understand the concept of *Model of Computation* (MoC). The concept of MoC is understood in this thesis in the same manner as in the publication by Jantsch & Sander (2005): a MoC defines how computation takes place in a structure of concurrent processes. Models of Computation are related to programming languages, but can also be decoupled from each other (Lee 2001).

Jantsch & Sander (2005) divide MoCs into several subcategories, with the representation of *time* being the most distinguishing criterion in their publication. The authors divide the MoCs into *continuous-time*, *discrete-time*, *synchronous* and *untimed models*. Considering this thesis, the discrete-time model is used in the VHDL language that is used to describe the circuit presented in Chapter 8. The synchronous data-flow model presented in Section 4.1.1 belongs to the category of untimed models.

Models of Computation also differ from one another in their capability of representing program behaviour, which is also called expressiveness or expressive power (Wolf 2006). Expressiveness is the capability of supporting program control constructs such as *if-then-else*. In the MoCs underlying traditional sequential programming languages, it is given that these constructs are supported. However, in data-oriented MoCs the if-then-else functionality, which is needed for efficient modeling of MPEG-4 and 3G LTE baseband processing, is not mandatory since there are many traditional signal processing applications, such as sample rate conversion that do not necessarily need such control behaviour.

The expressiveness is not the only characteristic of a MoC – the other interesting feature is the analyzability of the model. Unfortunately, these two characteristics are generally tradeoffs. As the expressive power of a model increases, the complexity of verifying the correctness of the system is magnified (Varea *et al.* 2006). Thus, choosing the Model of Computation for describing a system means balancing between the analyzability and expressiveness of a model.

In the next section, we shall introduce the MoCs that are used in modeling the scheduling problems that are related to the content of this study.

4.1.1 Synchronous data flow

In this thesis, we often refer to the synchronous data flow (SDF) (Lee & Messerschmitt 1987) MoC or its extension called Parameterized SDF (Bhattacharyya & Bhattacharyya

2001). In SDF and PSDF the application or algorithm is expressed as a directed graph (DG) (Sriram & Bhattacharyya 2000). A DG consists of *vertices* and unidirectional *edges* that link the vertices together as a graph. A DG vertex represents a part of the application and an edge represents a data dependency between two application parts. The direction of the edge is set so that the edge is directed towards the vertex (application part) that is dependent on the one that the edge comes from.

SDF and PSDF are not the only MoCs that are expressed with DGs, and therefore we shall discuss the properties of (P)SDF more specifically to distinguish them from other MoCs. SDF and PSDF use a restricted type of graph, the SDF graph, to model applications. An SDF graph consisting of eight actors (name for vertices in SDF) is depicted in Figure 11. Data is represented as *tokens* that can contain any amount of real data. Tokens are placed on edges; a token on edge e has been produced by the actor from which e originates and it is ready to be consumed by the actor where e ends. In Figure 11 edges are named $e_{1...11}$ and tokens are symbolized by black dots.

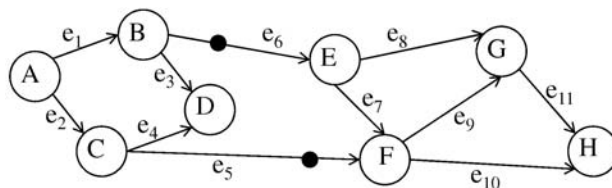


Fig 11. A synchronous dataflow graph. Token rates of ports are not shown; they are assumed to be 1, which makes the graph a homogeneous SDF graph.

In SDF, every actor can *fire* independent of other actors. However, the actor must have a sufficient number of tokens on its input edges before it can fire. In Figure 11, actor E is ready to fire, whereas F is not, since e_7 does not bear a token. In practice, firing an actor means executing a program function or method, and tokens represent input data. Thus, it is easy to understand that a function cannot be started before its input data (parameters) are available. Finally, after the firing, the actor produces a set of tokens to its output edges.

In pure SDF, the token consumption and production rates are fixed during the program execution. This has several implications. First of all, it allows the system designer to create a static schedule at system design time, where all the actors are fired in an optimal sequence and with minimal runtime overhead. On the other hand, this is also a restriction that disallows the modeling of dynamic programs that would require

changing the token rates at runtime. A special simplification of SDF is homogeneous SDF (HSDF), where all token rates are exactly *one*. General SDF graphs can be transformed as HSDF graphs (Sriram & Bhattacharyya 2000).

For the scheduling of SDF graphs, each actor must be assigned to a processor in the system. To enable static scheduling, the assignment must evidently be done at design time. A schedule for an SDF graph is computed by using a scheduling algorithm that is compatible with the SDF MoC. An example of such an algorithm is the work of Pino *et al.* (1995). A static SDF schedule defines on which processor each actor fires and when that happens. Schedules are generally periodic and overlapping, except for *blocked schedules* where all actors in the graph need to have fired exactly once before a new *iteration* starts (Lee & Messerschmitt 1987).

In real-world implementations, the application parts modeled as actors can be implemented in a software program that runs on a programmable processor or even as an ASIC. The edges are implemented as FIFO queues or with computationally lighter *static buffering*, explained in the work of Lee & Messerschmitt (1987). From the modeling point of view, static buffering and FIFOs work identically.

4.1.2 Parameterized synchronous data flow

PSDF (Bhattacharyya & Bhattacharyya 2001) extends the synchronous data flow MoC with support for modeling data-dependent applications, and provides the flexibility required by quasi-static scheduling. This is done by defining a consistent way of changing the token consumption and production rates (*configuration*) of actors *between* graph iterations. The token consumption and production rates of actors cannot be changed while the graph is executing.

A PSDF *subsystem* consists of a *body graph*, *init graph* and *subinit graph*. Figure 12 depicts an arbitrary PSDF subsystem, *SS*, that consists of three graphs: *SS.init*, *SS.subinit* and *SS.body*. The graph *SS.body* contains four actors *A*, *B*, *C* and *D*, of which *C* is hierarchical and contains another PSDF subsystem. The body graph models the functional behaviour of the system and is actually a plain SDF graph after the graph parameters have been configured. Both the init graph and subinit graph configure the body graph parameters, but have slightly different functions. The subinit graph is allowed to set the internal parameters of the body graph, but it is not allowed to change parameters that affect the body graph's interface to the outside. In *SS*, the edges e_{ia} , e_{ib} and e_o serve as an interface to the outside, and can thus be configured by *SS.init*. A

parameter that the subinit graph may modify is the token rate between two actors inside the body graph. The actors inside $SS.body$ are interconnected with edges e_1 , e_2 and e_3 , whose token rates can be set by the graph $SS.subinit$. The subinit graph can also have dataflow inputs (but no outputs) from the parent graph. (Bhattacharya & Bhattacharyya 1999)

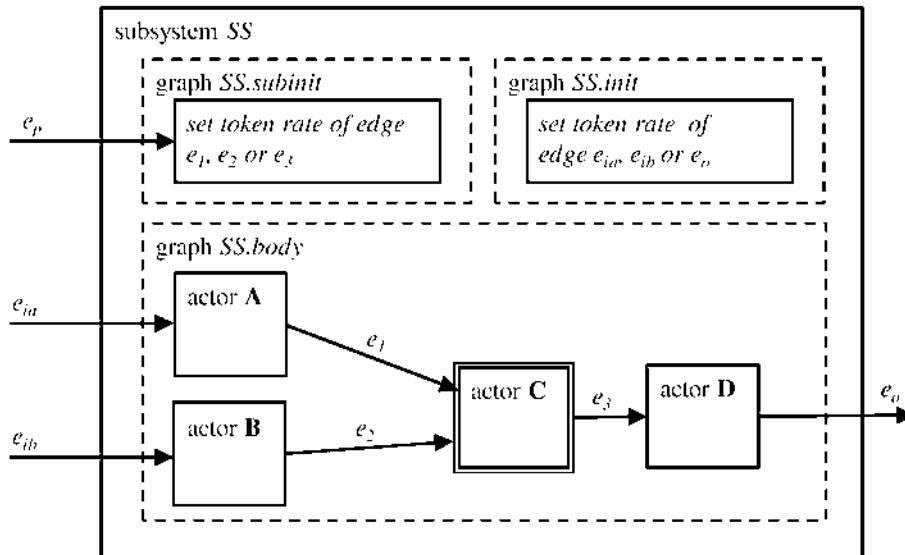


Fig 12. A PSDF subsystem with four actors: A, B, C and D.

The init graph on the other hand, is not allowed to have any dataflow input or output. The configuration power of the init graph is greater than that of the subinit graph, as it can also change the interface parameters of the body graph. The invocation frequency of the init and subinit graphs also differ. A subinit graph is invoked once for each subsystem invocation, but the init graph is only allowed to be invoked once for each invocation of the parent graph. (Bhattacharya & Bhattacharyya 1999)

PSDF enables the use of quasi-static scheduling, which is also mentioned in the work of Bhattacharya & Bhattacharyya (2001). A static multiprocessor schedule can be computed for each body graph configuration, and then can be sequenced at run-time as Lee (1988) has indicated in his work. In this thesis, we will discuss different techniques of sequencing consecutive quasi-static multiprocessor schedules with small run-time overhead. The rest of this chapter will focus on this topic.

4.2 Flow-shop scheduling

Flow-shop is a tool for efficient sequencing of multiprocessor computations. The origins of the flow-shop scheduling problem come from factory production lines where multiple production *machines* work in parallel and products move from one machine to another as they are assembled. As we now look at the formulation of the flow-shop problem, we shall remain faithful to the original flow-shop terminology and use the term *machine* when speaking about PEs. The application of flow-shop scheduling to video decoding was originally shown in the work of Boutellier *et al.* (2007a).

The flow-shop problem was initially formulated by Johnson in 1954 (Gupta & Stafford 2006) and has since been actively studied. The flow-shop scheduling problem involves the processing of N jobs on M machines. Each job consists of a set of *operations*, so that each operation has to be processed on exactly one machine. The processing times of operations are known beforehand and each job passes through the machines in a prescribed order, although it is possible for jobs to skip machines (Gupta & Stafford 2006). We assume that the optimization objective of the flow-shop problem is makespan minimization instead of other possible scheduling objectives. Energy minimization would also be an appropriate objective, but it is not considered here.

The permutation flow-shop (PFS) problem is a more restricted version of the general flow-shop problem. In permutation flow-shops, the job order for each machine is the same (French 1982), which means that a certain operation m must be performed on job $n - 1$ before it can be performed on job n (assuming we have to complete the jobs $1 \dots n$ in ascending order). A flow-shop schedule is completely specified by a permutation sequence $(1, 2, 3, \dots, n)$ that describes the job order, if we agree upon the method of *timetabling* (French 1982). Timetabling is the process of constructing actual start and end times for each operation and job based on the order of jobs and technological restrictions. Reflecting back to the scheduling algorithm taxonomy shown in Figure 10, run-time flow-shop scheduling with job ordering can be classified as a sort of *static assignment* scheduling.

Semi-active timetabling is a method to unambiguously derive schedules from ordered job sequences. In semi-active timetabling, each operation is started as soon as the previous operation on the same machine has finished *and* the previous operation of the same job has finished. Another possibility is to use *no-wait* timetabling. In no-wait timetabling each operation (n) within one job is started immediately after the previous operation ($n - 1$) of the same job has finished. Figures 13(a) and 13(b) show the

difference of no-wait timetabling and semi-active timetabling in Gantt-charts. The operations in the figures are named with the notation nm , which means that operation nm belongs to job n and is executed on machine m . The differences between no-wait timetabling and semi-active timetabling not only affect the operation start times, but may also affect schedule makespans

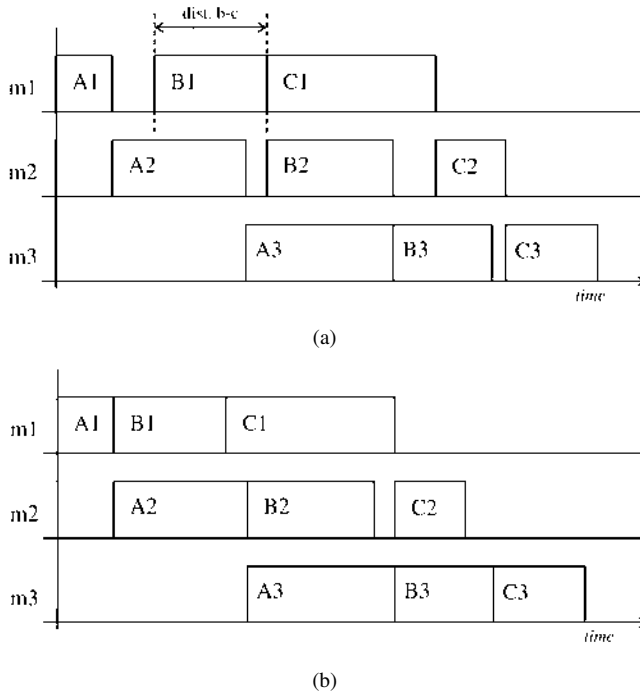


Fig 13. A Gantt chart of a no-wait (a) and a semi-active (b) schedule. (Boutellier *et al.* 2009a, modified by author with permission of Springer.)

Figure 14 shows a DG formulation of a permutation flow-shop problem after the job order has been fixed. Each row of actors represents one job, and each column represents a machine. One actor represents the operation of a job on a particular machine. Notice that in the figure two jobs skip some machines.

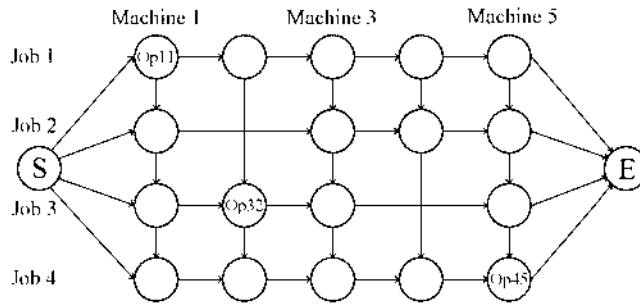


Fig 14. A directed graph showing a flow-shop problem. (Boutellier *et al.* 2009a, published by permission of Springer.)

It can be seen that the execution order of operations is very restricted. After operation (1,1) has been executed, only operations (1,2) and (2,1) can be executed. When the method of semi-active timetabling is used, operations (1,2) and (2,1) start at the same time, whereas no-wait timetabling requires (2,1) to be executed immediately after (1,1). In no-wait, (1,2) may only be executed when it is sure that (2,2), (3,2) and (4,2) can follow (1,2) instantly without waiting. Since the graph does not allow much freedom in schedule optimization, the remaining variable to be optimized is the order of jobs, as was stated in the problem formulation of permutation flow-shop scheduling. Changing the order of jobs alters the graph by interchanging the rows of actors.

The flow-shop jobs can be interpreted as parts of a static multiprocessor schedule. Thus, a flow-shop sequencing problem is a form of quasi-static multiprocessor scheduling, where static schedule parts are selected and combined at run-time, data-dependently. The low complexity and efficiency of semi-active timetabling and no-wait timetabling makes flow-shop scheduling interesting for practical run-time systems. Measurements that compare different flow-shop heuristics are presented in Chapter 6.

The ordinary flow-shop model assumes that once the computations of a job have finished on one machine, the computations continue on the next machine – it is not allowed to simultaneously perform computations for one job on several machines. In factory production lines this is understandable, but in the application of scheduling fine-grained tasks on parallel PEs, this restriction is impractical. In Section 4.4, extended permutation flow-shop scheduling is introduced. Extended permutation flow-shop scheduling is a more relaxed model where the computation of one job can simultaneously take place on several machines.

In general it can not be assumed that the number of different schedules required

by the quasi-static system would be so small that all the required schedules could be pre-computed and stored for run-time use. The more general case is that short schedule pieces can be assembled and stored at compile time and then practical run-time schedules are composed by sequencing and overlapping several of these pieces. Flow-shop scheduling is a method that can be used to compose a single schedule from several pieces in this way.

4.3 Granularity of modeling

Independent of the MoC, the granularity of modeling needs to be chosen in application design. The granularity affects the number of program details that are shown. For human viewers, exposing maximally many program details is not necessarily the best solution, since too many visible details make the important high-level structures disappear. Hierarchical modeling, which is also possible in (P)SDF, is a key method in modeling large applications without compromising the accuracy of modeling small details.

The granularity of modeling has also a strong influence on the technical functionality of an embedded system. This can be illustrated by inspecting Figure 15 that reveals the impact of granularity on the functionality of scheduling. In the left part of Figure 15, functions *a* and *b* have been modeled as actors that are interconnected with an edge. On the right side, the same functions *a* and *b* are modeled in higher detail, which reveals that *a* and *b* consist of eight iterations of the same computation.

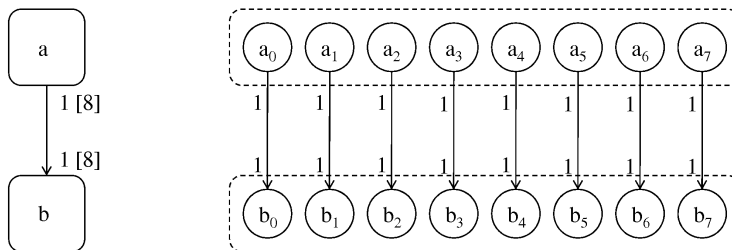


Fig 15. Different granularities of modeling.

Figure 16 shows the Gantt charts of parallel multiprocessor schedules of *a* and *b*. The Gantt chart reveals that when *a* and *b* are modeled in higher detail, they can be executed in parallel with a remarkably shorter schedule makespan. Another issue that is not so obvious is related to the real-world implementation of communication edges between actors. When the edges are implemented with FIFO buffers, the buffer space requirement

of the coarsely modeled solution is eight times higher than that of the high detail model. If the coarse-granularity system is implemented as shown in the model, a buffer for eight tokens needs to exist between a and b . For the fine-granularity system the required buffer size is only one.

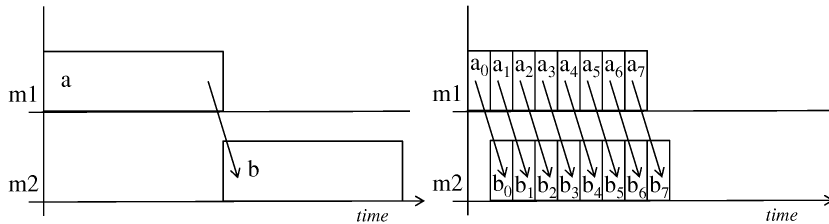


Fig 16. Gantt charts of schedules of systems from Figure 15.

The coarse-grain model schedule in Figure 16 can obviously be interpreted as a flow-shop job that has been scheduled on two machines. Similarly, the fine-grain model of the same figure can be interpreted as eight identical jobs that have been scheduled on two machines. The original assumption was that both the fine-grain model and the coarse-grain model depict the same functionality that has been expressed in two different ways. This means that a trade-off exists: either, the designer can model the function with less detailed information (two operations), but pay the cost of a suboptimal schedule and extra buffer space (Gangwal *et al.* 2001), or increase the amount of information (16 operations) and get a lower cost result. Increasing the amount of information is undesirable if run-time scheduling is considered: scheduling eight jobs instead of one increases the scheduling overhead considerably.

The conventional flow-shop model does not offer a way of profiting from the benefits of both granularities. In the next section, we shall extend the flow-shop model so that the advantages of both modeling granularities can be adopted.

4.4 Extended permutation flow-shop scheduling

Figure 13(a) shows the Gantt chart of a permutation flow-shop scheduling problem consisting of 3 jobs, 3 machines, and 3 operations per job. Each task within a job is executed on a separate machine and no-wait timetabling (French 1982) ensures that the next task within the same job starts immediately upon completion of the previous task in the same job.

We define a new concept that we name *inter-job distance*. The inter-job distance is the time offset between the start times of the first operation of job n and the first operation of job $n + 1$. An example of the inter-job distance is shown in Figure 13(a), where the inter-job distance is shown for jobs B and C. We shall also define another concept that we name *operation offset*. Operation offset of operation m of job n is the starting time of operation m subtracted by the starting time of first operation of job n .

In conventional flow-shop modeling with no-wait timetabling the operation offset of m is equal to the accumulated lengths of operations $1 \dots m - 1$. We shall now depart from the conventional flow-shop model and explicitly define an operation offset $o_{n,m}$ for each operation m of job n . If the inter-job distances between each job n and job p are also allowed to be defined arbitrarily, it is possible to extend flow-shop scheduling to support additional dependencies between operations, that are not possible with conventional flow-shop. For example in the situation of Figure 13(a), this extension would allow introducing a dependency that C1 is forced to start only after B2 has finished. More importantly, the extension allows several operations of the same job to be executed at the same time, which is very useful in modeling multiprocessing applications.

We shall call this version of the flow-shop model with the two additional features, the *extended permutation flow-shop* (EPFS), which was initially introduced in the work of Boutellier *et al.* (2008). Pragmatically, EPFS behaves like its parent, permutation flow-shop scheduling with no-wait timetabling, but the detailed definition of EPFS is shown in Figure 17. In the definition, the optimization objective of the problem is left open, as there are several alternatives for that.

Item 4 in Figure 17 requires some further discussion. It is possible to set the processing time of an operation ($t_{n,m}$) to zero, which has a special meaning: that operation is skipped. In the practical implementation of an EPFS scheduler, special attention has to be paid to the zero-length operation truly being ignored and not affecting the sequencing of jobs.

1. There is a fixed and known set J , consisting of N jobs: $J = \{1, 2, \dots, N\}$.
2. The operations of job $n \in J$ have to be processed on M machines $1, 2, \dots, M$ in that order, once and only once.
3. Each job $n \in J$ consists of M operations $O_{n,1}, O_{n,2}, \dots, O_{n,M}$. The processing of operation $O_{n,m}$, belonging to job n on machine m , takes $t_{n,m}$ integer time units and may not be interrupted.
4. All $t_{n,m} \geq 0$ and are fixed and known in advance.
5. The starting time of operation $O_{n,m}$, $m = 2, \dots, M$, $n \in J$, is the starting time of operation $O_{n,m-1} + o_{n,m}$, where $o_{n,m}$ is a positive or a negative integer.
6. Each *schedule* of jobs is a permutation $\pi = (\pi(1), \pi(2), \dots, \pi(N))$ of set J .
7. All machines are initially available and no machine may process more than one operation at a time.
8. The processing order of jobs is the same on each machine.

Fig 17. The definition of the EPFS scheduling problem.

In the previous section it was shown how different modeling granularities affect the operation scheduling. EPFS allows combining the advantages of coarse and fine-grained modeling, as depicted in Figure 18. One job can now be active on several machines simultaneously, which enables more general modeling of multiprocessor tasks. A more concrete explanation follows in the next subsection.

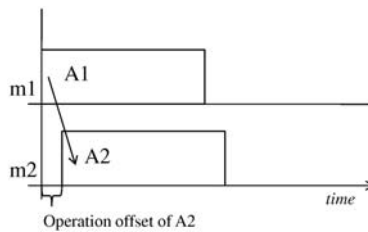


Fig 18. Operation offset.

4.5 Combining synchronous data flow and flow-shop

Two Models of Computation (SDF and PSDF) and two scheduling approaches (PFS and EPFS) have now been introduced in this chapter. In this section, we shall see what the roles of these approaches are in data-oriented application modeling and scheduling.

Starting from the scope of the whole application, such as MPEG-4 video decoding, it is evident that the application model needs to support if-then-else -like control structures to capture, for example, all the different macroblock decoding paths. PSDF is a suitable MoC for this, as it provides a way of expressing control structures.

Following the PSDF terminology, the application is modeled as a PSDF body graph and the changes in the program control are implemented through parameter changes via init and subinit graphs. In the Subsection 4.1.2, it was stated that after the system parameter values have been set, the body graph is actually a pure SDF graph that is statically schedulable. This means that the PSDF model of the application can also be viewed as a repository of SDF graphs and the parameters as an indexing system where one combination of parameters points to exactly one SDF graph. Following this view of the system, the repetitive iterations of the system can be seen as a chain of consecutive SDF graph invocations.

When we recall that at least one multiprocessor schedule can be formed for each SDF graph in the general case, we see that the execution sequence becomes a flow-shop scheduling problem, where each static SDF graph schedule becomes a flow-shop job or more generally, an EPFS job. Thus, a sequence of N SDF graph executions becomes an extended permutation flow-shop scheduling problem of N jobs, which is illustrated in Figure 19.

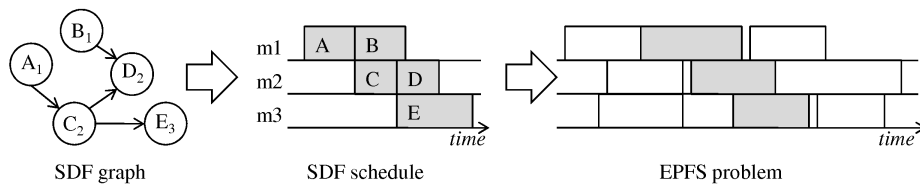


Fig 19. Formulating an SDF graph to an EPFS scheduling problem. (Boutellier *et al.* 2009c, published by permission of Springer.)

Ko & Bhattacharyya (2005) have also proposed the approach of pre-computing SDF graph schedules at system design time, and storing the schedules for efficient run-time use. However, their work does not describe the issue of combining the multiprocessor schedules at run-time. Das *et al.* (1997) proposed generating a set of alternative static schedules at compile-time and using them at run-time to achieve fault tolerance in running a distributed program on a cluster of networked workstations. The work of Cortés *et al.* (2005) also uses a set of compile-time generated schedules.

In the RVC formulation of the MPEG-4 Simple Profile decoder (Figure 6), static tasks such as IDCT can be mapped as static (Falk *et al.* 2008) subgraphs of the body graph of the whole PSDF system. Unfortunately, real applications contain also non-deterministic tasks, like Inverse Scan, that have unpredictable execution times. Since the EPFS formulation does not allow unpredictable execution times in jobs, the non-determinism must be hidden inside the task model (operation or SDF actor) and reveal only the worst-case execution time to the model. This approach causes a degradation to the throughput of the system, as the worst-case latency has to be assumed always. However, if the unpredictable task is implemented as an LSI circuit, the inefficiency caused by unpredictable execution times can be alleviated by a power saving scheme that is enabled as soon as the task has finished.

EPFS provides a way for sequencing quasi-static application schedules on parallel PEs. Chapter 7 will show practical measurements that show the efficiency of the scheduling algorithm implementation even when the PE tasks are fine-grained and have short execution times. Chapter 8 will describe a dedicated hardware scheduler that supports run-time scheduling of EPFS jobs.

4.6 Relation of EPFS to more general scheduling approaches

Flow-shop and EPFS are special cases in the large variety of multiprocessor scheduling methods that have been developed. To see how EPFS relates to the other existing scheduling methods, we first have to limit the scope of suitable scheduling methods to match the problem at hand.

The most obvious restriction that has to be imposed on the scheduling method, is that it (1) must support multiprocessor systems. This excludes uniprocessor scheduling algorithms from our study. Second, it is essential that (2) the processors in the system are allowed to be heterogeneous, because embedded MPSoCs (Wolf 2004) must also be considered. This restriction rules out popular *load balancing* methods for homogeneous PEs (Kumar *et al.* 2007).

Within the scheduling approaches that satisfy both requirements (1) and (2), a major dividing line is whether the set of tasks to be scheduled is assumed to be fixed and known, or not. If tasks become available to be scheduled sporadically, the scheduling algorithm must be of the control-oriented type that was discussed in Section 3.3.3. By

contrast, if (3) the set of tasks is known and fixed when the scheduler is invoked, it is essential that (4) the tasks are allowed to have dependencies, because this is needed when the application is modeled for the scheduler.

If we assume that the system also contains application specific hardwired PEs (accelerators), there must be a way of (5) disallowing task pre-emption in the scheduling algorithm, because implementing pre-emption of accelerators is costly (Kumar *et al.* 2008). Task pre-emption refers to the action of suspending the execution of a lower priority task, because a higher priority task must suddenly gain access to the PE.

Finally, if (6) the task execution times are assumed to be known and fixed, the algorithm by Bajaj & Agrawal (2004) can be applied. Bajaj & Agrawal (2004) have proposed a scheduling method that satisfies these requirements (1 ... 6), and even provides optimal results when certain conditions are satisfied. The approach of Bajaj & Agrawal (2004) uses directed, acyclic graphs as an application representation. Since the set of PEs is assumed to be heterogeneous, the Bajaj & Agrawal (2004) algorithm needs to know what the execution time of each task is on each PE.

The compile-time generated multiprocessor schedule for the application graph (using, for example, the approach of Bajaj & Agrawal (2004)), actually is an EPFS job when it is placed on a Gantt chart. This procedure is the same as that depicted in Figure 19.

As a summary, it can be stated that the quasi-static EPFS scheduling approach that is advocated in this thesis is designed for heterogeneous multiprocessor applications that are piecewise predictable. The application area that can, in some sense, be seen as the opposite of the scope discussed here, is the control-oriented scheduling of sporadically appearing tasks that possibly have non-constant (or unknown) execution times. For the piecewise predictable MPEG-4 SP video decoding and MIMO-OFDM baseband processing that have been taken as application examples in this thesis, the application of these control-oriented scheduling methods should be avoided (Sriram & Bhattacharyya 2000, Xu & Parnas 2000), since they consume a large amount of run-time resources.

In the next chapter, the modeling issues discussed thus far are concretized by showing how an RVC video decoder can be transformed into a PSDF model and further into EPFS jobs.

5 Quasi-static scheduling of an RVC decoder

In this chapter, a practical case is shown, where a concurrent model of an RVC video decoder is transformed into a set of SDF graphs, and a set of static schedules are generated for efficient run-time execution.

The effort to design the Reconfigurable Video Coding (RVC) standard (Lucarz *et al.* 2007) was motivated by the intent to describe existing video coding standards with a set of common atomic building blocks (*e.g.*, IDCT). Under RVC, existing video coding standards are described as specific configurations of these atomic blocks. This greatly simplifies the task of designing future multi-standard video decoding applications and devices by allowing software and hardware reuse across video standards. The RVC coding tools are specified in a dataflow/actor object-oriented language named CAL (Eker & Janneck 2003) that describes the atomic blocks in a modular way, and exposes parallelism between computations.

The CAL Model of Computation is *untimed* (Lee & Sangiovanni-Vincentelli 1998), which means that the order of events in a CAL specification reflects ordering induced by causality; there is no particular schedule. In practice, this means that various control mechanisms (Lucarz *et al.* 2008) need to ensure at run-time that events take place in the correct order. Such run-time decision making renders the specification flexible, but creates overhead and unpredictability.

As a remedy, we introduce a methodology to derive quasi-static execution schedules for a set of CAL actor networks, including the RVC MPEG-4 Simple Profile video decoder. In quasi-static scheduling, most of the scheduling effort takes place off-line, and only some infrequent data-dependent scheduling decisions are left to run-time. The off-line determined schedules are collected to a repository that is used by the run-time system, which selects entries from the repository and appends them to the ongoing program execution. This approach limits the number of run-time scheduling decisions and improves the efficiency of the system. The topics discussed in this chapter are also presented in Boutellier *et al.* (2009c).

5.1 Related work

Similar MoCs to the networked CAL actors have been proposed by Girault *et al.* (1999), Thiele *et al.* (1999) and Buck & Vaidyanathan (2000). These Models of Computation integrate finite state machines (FSM) with dataflow graphs (Bhattacharya & Bhattacharyya 2001). An option for scheduling networked CAL actors would be to transform the CAL network into one of these representations because CAL actors can be expressed as FSMs that contain variables. This makes the CAL FSMs into *extended finite state machines* (EFSM). Henniger & Neumann (1995) state that EFSMs can be transformed into regular FSMs, with the cost of a possible state-space explosion. Instead of transforming the networked CAL actor model into one of the networked FSM models, the Parameterized SDF (Bhattacharya & Bhattacharyya 2001) MoC was chosen for practical reasons: because PSDF is a dataflow MoC, existing dataflow tools and methods are more likely to be applicable to PSDF than the more exotic MoCs.

The CAL EFSMs contain run-time control mechanisms (Lucarz *et al.* 2008) that allow or deny possible state transitions. Actors are connected to each other with *dataflow edges* that are attached to the actor *ports*. Actors communicate by firing tokens along the edges of the dataflow graph.

The idea of this study is to reduce the run-time overhead of the targeted CAL networks by analyzing the network behaviour off-line and thus replace the run-time state transition control mechanisms by a set of static schedules. Our approach relies on the assumption that the CAL EFSMs have a limited state-space, which we also assume for the MPEG-4 SP decoder model. The static scheduling is made possible by transforming the untimed CAL model into a set of HSDF graphs that are statically scheduled at design time. The schedules are stored and invoked at run-time on demand, which allows some data dependence in the run-time control flow of the CAL networks. In the literature, this is called quasi-static scheduling. We express the resulting model with the Parameterized Synchronous Data Flow (Bhattacharya & Bhattacharyya 2001) notation.

Recently, also two other approaches have been proposed for developing a methodology for deriving efficient schedules for CAL actor networks. The approach of von Platen & Eker (2008) sketches a method to classify CAL actors to different dataflow classes for efficient scheduling. Gu *et al.* (2009) introduce a way of extracting statically schedulable regions from CAL networks. Both of these approaches, as well as the one presented in this study, try to minimize the number of run-time scheduling decisions.

The Open Dataflow environment (Bhattacharyya *et al.* 2008) supports the design, simulation and debugging of CAL models. Deriving software and/or hardware implementations from these CAL models is a non trivial task. However, several tools do already exist: a hardware code generator converts CAL actors to a hardware description (Janneck *et al.* 2008) and a software code generator converts CAL actors into their C/C++ implementation (Wipliez *et al.* 2008). The hardware and software code generators have the capability to compile networks of actors. Currently, the software code generator converts CAL actors to C language, and inserts automatically a dynamic scheduler written in SystemC in order to determine at run-time the order of execution of the actors.

5.2 The proposed approach

Figure 6 shows a high-level view of the RVC implementation of the MPEG-4 Simple Profile decoder. The first three steps of our approach, which are described in this section, describe a set of procedures that transform the CAL actor network into a set of statically schedulable HSDF graphs. Section 5.3 describes the PSDF model of the MPEG-4 SP decoder. Section 5.4 describes the off-line and on-line scheduling of the decoder model.

5.2.1 Preprocessing

The scheduling approach proposed in this thesis requires some preprocessing of the CAL actor network to acquire information that is necessary for the graph transformations. Figure 20 shows the abstract model of the assumed CAL network. The model has a dataflow input and output, as well as an input interface P .

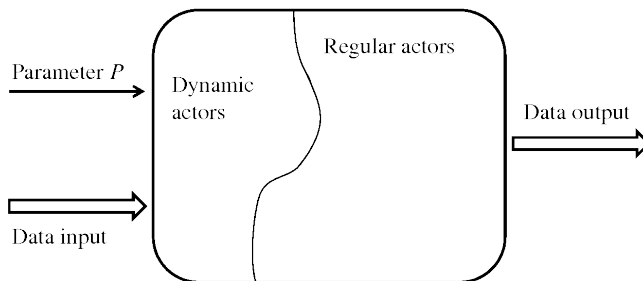


Fig 20. The high-level model of the system to be scheduled. (Boutellier *et al.* 2009c, published by permission of Springer.)

An input interface provides dataflow tokens from outside the considered system. The dataflow token values that arrive in the system through P change the configuration of the CAL actor network according to the token values. P needs to be clearly identified and defined before applying our approach to the network. P is a combination of a non-negative number of dataflow input interfaces. Every combination of token values that flows through P , must be known at system design time. The combinations of the values of tokens flowing through P are evaluated as values of P . For example, if P consists of two input interfaces a and b , both of which may carry tokens with a value '0' or '1', P can get four different values: '00', '01', '10' and '11'. Reflecting back to the PSDF MoC, P equals to the dataflow inputs of the subunit graph.

The identification of P happens manually at the moment. For an arbitrary actor network, P can be identified as follows: all changes in the network topology or actor token rates must happen as a function of P . On the other hand, P must not contain interfaces that do not cause network topology or token rate changes.

Our scheduling algorithm creates a static schedule for every possible token value combination that arrives through P . For example, if P can get only one value, only one schedule is created. On the other hand, if P is a combination of several input interfaces, the number of generated schedules can become considerable. In the MPEG-4 SP application example (Figure 6), the $BType$ interface is P . Based on the CAL code, the tokens arriving through $BType$ can get 4096 different values, but inspection of the whole decoder network reveals that only five different network topologies are produced; many values produce an identical actor network. Evidently, the $BType$ values that result in identical networks should be mapped to a single configuration. The number of different token value combinations transmitted through P is denoted with the notation $|P|$. Reflecting back to Section 4.5, P represents the parameter that at run-time points out a pre-computed schedule from the repository schedules.

On the *actor level*, the network must be analyzed to extract information about the token rates of actor ports *for each of the $|P|$ configurations*. For each port of all actors in the system, the P -value dependent token production and consumption rate must be known, or be specified as variable, which excludes the actor from quasi-static scheduling. Determining the token rates requires exploration of the state machines of the actors. The EFSMs are assumed to have an *initial state*, from which several state transition paths depart. It is also assumed that all of the departing state transition paths eventually lead back to the initial state. An example of a state transition path can be seen in Figure 21: $newVop \Rightarrow other \Rightarrow other$.

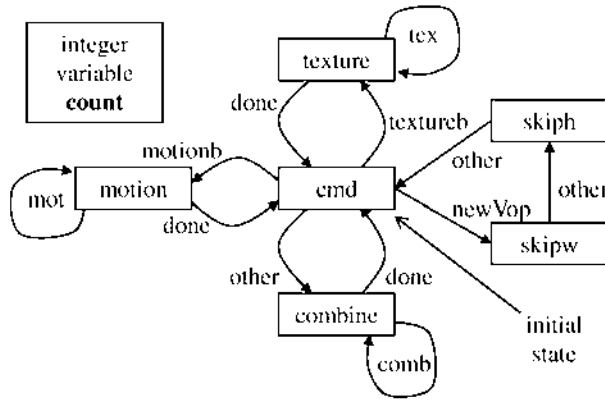


Fig 21. The EFSM of the *add* actor, classified as regular. (Boutellier *et al.* 2009c, published by permission of Springer.)

5.2.2 Actor classification

If the token consumption and production rate for all state transition paths of an actor only change as a function of P , the CAL actor is classified as *regular*. If any of the actor ports has a non-constant token rate with any value of P , the actor is *dynamic*. In Table 3 which presents an overview of our approach, the number of regular actors is denoted by K .

Table 3. Overview of our approach. (Boutellier *et al.* 2009c, published by permission of Springer.)

Step	Explanation
1	Classification of CAL actors: regular or dynamic
2	Unrolling of K actors: from K EFSMs to $K * P $ HSDF graph fragments
3	HSDF graph merging: merge $K * P $ HSDF fragments to $ P $ system-level graphs
4	Processor assignment: assign each HSDF actor to a PE
5	Off-line scheduling: compute a static schedule for $ P $ system-level graphs
6	Execution and run-time scheduling

Regular actors require some explanation: a regular actor has several operation modes that are switched by the parameter P values. In the EFSM representation of an actor, the value of P selects the state transition taken from the initial state. With a regular actor,

the token rate of each port may only vary as a function of P . As an example, we can take the *add* actor in Figure 21 and Figure 22. When $P = V_1$, the actor consumes 64 tokens from input port [TEX] and produces 64 tokens through the output port [VID]. When $P = V_2$, the actor consumes 64 tokens from port [MOT] and produces 64 through [VID], but does not consume any tokens from [TEX]. So, the token rates of ports for this actor do change, but only as a function of P .

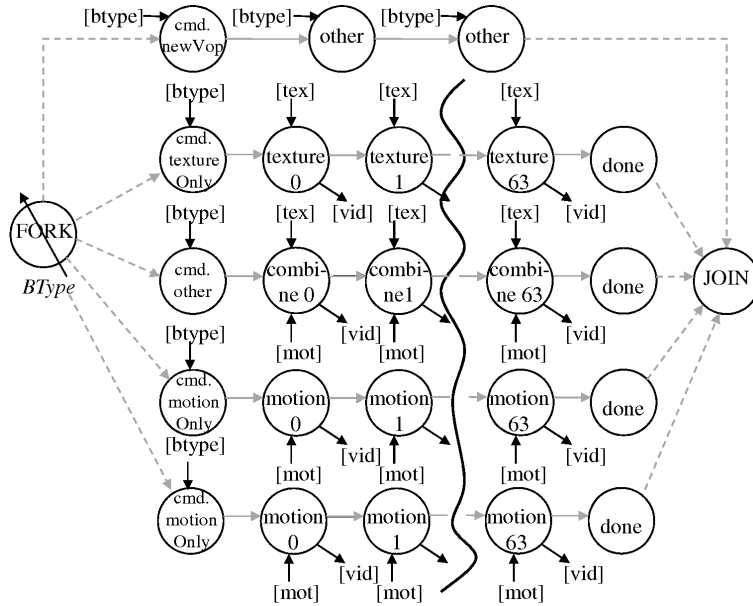


Fig 22. The five HSDF graph fragments acquired from unrolling *add*. (Boutellier *et al.* 2009c, published by permission of Springer.)

As another example, Figure 23 depicts the EFSM of the *inverse scan* actor that is classified as dynamic. The EFSM state *full* has two departing state transitions (and it is not the initial state), which makes the token rates of several state transitions unpredictable. Thus, the *inverse scan* actor is classified as dynamic. The token rates are not visible in the figure and not even directly in the CAL code. At the moment, the token rates are extracted from the CAL actors manually and it is unclear if this can be automated or not.

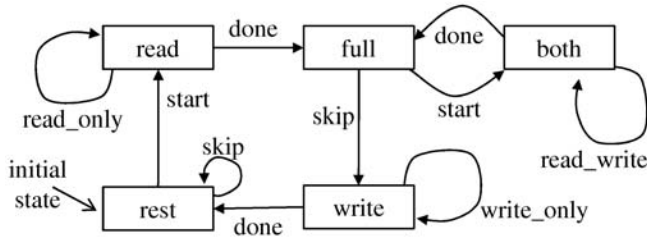


Fig 23. The EFSM of the dynamic *inverse scan* actor. (Boutellier *et al.* 2009c, published by permission of Springer.)

Furthermore, at the actor *network level*, our model assumes that the scheduled part of the CAL network matches the model in Figure 20. Verbally, the model is as follows: regular actors are not allowed to have dynamic actors connected to their output ports. If a CAL network does not match this network level requirement, there are two possibilities: 1) adjust the coverage of the scheduled network part or 2) reclassify individual actors so that the model requirements are met. Let us take an example of actor reclassification: if actor *A* has been initially classified as a regular actor, but the output ports of *A* are connected to a dynamic actor *B*, *A* can be reclassified as a dynamic actor to meet the requirements. This reclassification overrides the port token rates. In fact, both of these two options have the same effect: actors outside the scheduled network, as well as dynamic actors, are left out of the further steps described in this chapter.

Figure 6 depicts the result of our actor classification for the RVC implementation of the MPEG-4 Simple Profile decoder. Actors with thick outlines are regular actors and those with thin outlines are dynamic actors. The *DC prediction* actor is initially classified as a regular actor, but because one of its outputs is connected to the dynamic *Inverse scan* actor, *DC prediction* has been reclassified as dynamic. The same applies to *DC split*.

For this thesis the port token rates and the specification of *P* were determined manually. Automating the acquisition of this information is left beyond the scope of this study.

5.2.3 EFSM unrolling

Step 2 of our approach is automatic: the EFSM representations of *regular* CAL actors are unrolled into a collection of HSDF graphs, so that every state transition becomes

an HSDF actor, and every EFSM state becomes an HSDF edge. EFSMs are assumed to have an *initial state* that serves as the origin of unrolling. It is assumed that all state transition paths leaving from the initial state eventually return to the initial state. The number of state transitions (l) originating from the initial state is assumed to be $1 \leq l \leq |P|$. As an example, in Figure 21, $l = 4$. If l could be less than 1, it would mean that the actor has no actions and thus, no behaviour. If l is larger than $|P|$, it means that the actor is not compatible with the previously introduced assumption that actor token rates must happen as a function of P .

We call the HSDF graph that is produced from unrolling one state transition path an HSDF *graph fragment*. Every CAL actor is required to produce $|P|$ graph fragments, some of which may be empty graphs with no HSDF actors. There is no possibility of representing dynamic control structures in HSDF, but for intuitiveness, the graph fragments can be illustrated as being connected by a fork-actor. The unrolled version of the EFSM in Figure 21 is shown in Figure 22. The two last graph fragments are identical because the corresponding values of P lead to identical behaviour in the EFSM of the *add* actor. In reality these represent the predicted block decoding modes with a non-zero motion vector and a zero motion vector.

The HSDF actors within one graph fragment are interconnected by static *control flow edges*. Control flow edges do not transmit data, they just make sure that the dependencies expressed in CAL actors are also observed in scheduling. Nevertheless, when the HSDF graph is later scheduled, control edges behave in the same manner as dataflow edges. In the figures of this thesis, control edges have a gray color. Dynamic actors are omitted during the unrolling phase, and thus they do not contribute to the creation of HSDF graph fragments.

The state transition paths of EFSMs may contain iterative self loops such as *comb* in Figure 21. This poses no problem if the number of loop iterations is fixed. Variable numbers of iterations within one P value cannot be supported by the proposed approach. Another issue arises when the number of state transitions is fixed and large, which respectively results in a large HSDF graph. However, the large HSDF graphs only exist during off-line scheduling: in the run-time system the program code represented by the HSDF actor does not need to be replicated but can be replaced by repetitive calls to the same function.

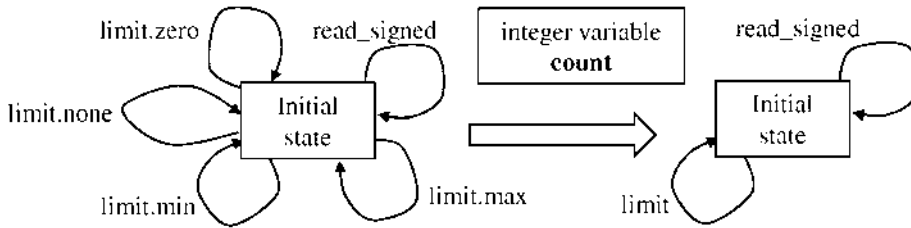


Fig 24. The original (left) and modified (right) FSM of the *clip* actor.

All of the actors in the MPEG-4 Simple Profile decoder do not map trivially into our model. One exception can be found within the hierarchical *IDCT2D* actor. One of the actors contained by *IDCT2D* is named *Clip* and it is responsible for saturating integer values. The problem with the implementation of this actor is that, depending on the value of the integer that has to be clipped, a different state transition is chosen. This is a feature that is not supported by our scheduling approach. In the work by Gu *et al.* (2009) the same problem has been noticed and corrected using a modified, predictable version of the same actor. We have chosen the same solution for this particular problem. The original and the modified version of the clip actor’s FSM is visible in Figure 24. With the modified clip actor, the state transition is chosen solely based on *BType*. The feedback loop that is seen in the CAL network of Figure 6, has a fixed and known number of iterations and thus poses no problem in graph unrolling.

5.2.4 Parameter-specific system-level graphs

The unrolled EFSM representations of the CAL actors consist of HSDF graph fragments that can be thought to be connected by a *fork* actor (see Figure 22). The HSDF actors in Figure 22 have the same data flow interfaces as the state transitions in the CAL code. In the third step of our approach, these data flow interfaces are automatically connected to the respective interfaces of graph fragments originating from other CAL actors in order to create *system-level* HSDF graphs. An example of joining graph fragments of two CAL actors is depicted in Figure 25. The gray vertices belong to the *interpolate* actor, and the white vertices belong to the *add* actor.

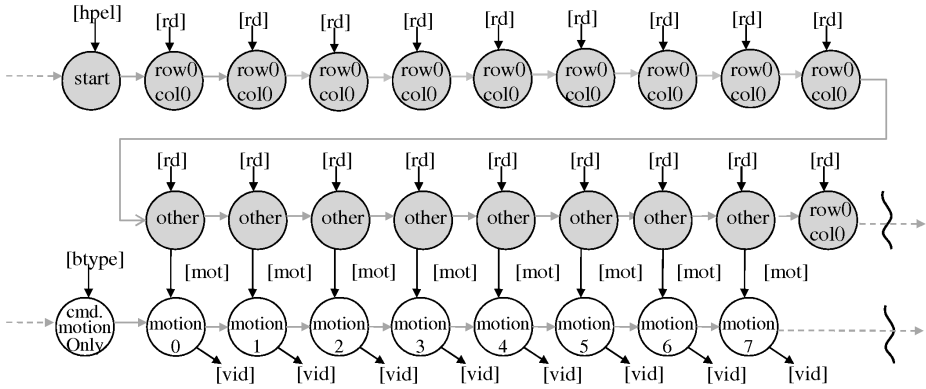


Fig 25. Joining the graph fragments of *interpolate* and *add* for one value of P . (Boutellier *et al.* 2009c, published by permission of Springer.)

Since there is one (possibly empty) HSDF graph fragment in every unrolled CAL actor for each value of P , the HSDF graph joining happens between fragments that represent the same value of P . For example, the HSDF graph fragment of *add* that represents P value l is joined with the HSDF graph fragment of the *interpolation* actor that represents P value l . Thus, for each value of parameter P , there will be a unique system-level graph. Figure 26 shows a system-level subgraph that contains all the actions and their dependencies.

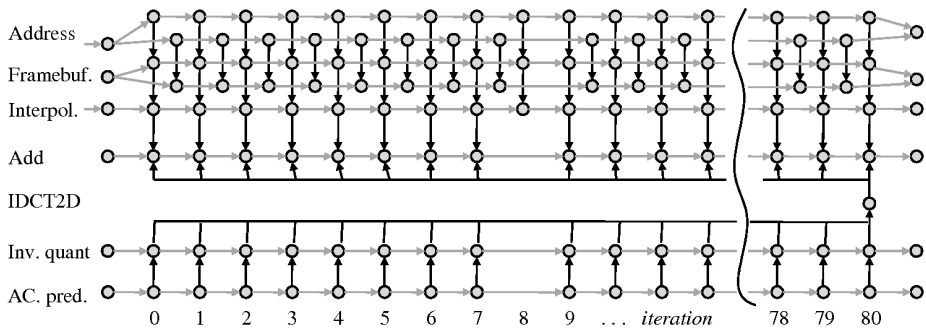


Fig 26. A system-level graph corresponding to the P value that enables motion compensation and texture decoding. (Boutellier *et al.* 2009c, published by permission of Springer.)

5.3 PSDF model of the system

Figure 27 shows a PSDF model of the MPEG-4 SP decoder that consists of a subunit graph, init graph and a body graph. The body graph contains seven actors, of which five are hierarchical and thus marked with a double rectangle. These actors are named U_1, U_2, U_3, U_4 and U_5 ; we shall denote the set of these five actors by U . The five actors of U contain the system-level HSDF graphs that were assembled as described in Subsection 5.2.4. We shall call the set of HSDF actors inside U by the name T . The non-hierarchical actor labeled D contains all the functionality that was classified in Subsection 5.2.1 as dynamic actors. Since the execution time of D is unpredictable, it is not modeled in greater detail here.

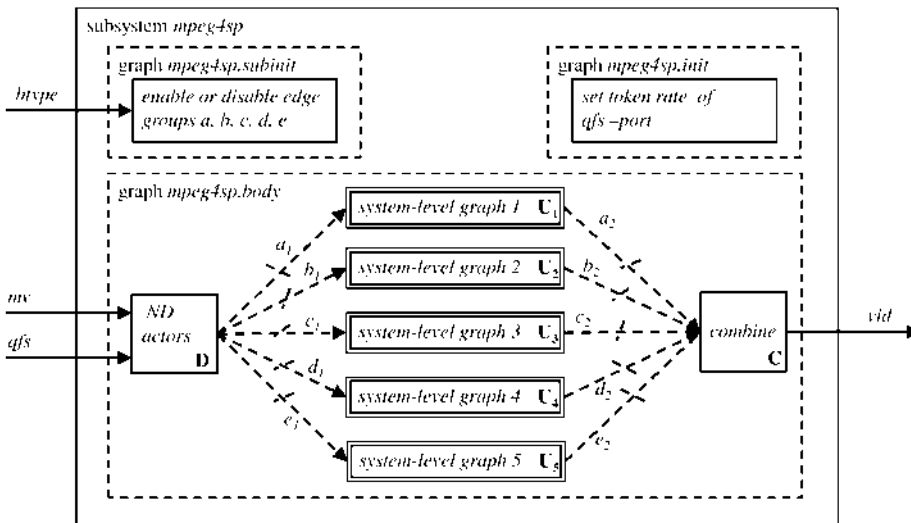


Fig 27. The PSDF model of the RVC MPEG-4 SP decoder system. (Boutellier et al. 2009c, published by permission of Springer.)

The actors U are connected to D by dashed edges. The dashed edges denote that the existence of these edges depends on the parameter values a, b, c, d and e that are toggled in the subunit graph. The parameterized existence of these edges is realized by changing the token production rate of the respective output port of D to zero when the edge is not needed. The parameterized edges a, b, c, d and e have been marked with a slash to denote that the edges actually represent a set of edges. The parameterized edges represent the same functionality on the system-level as the *fork* actor (see Figure 22) did

on the actor-level. The init graph could change the interface token rate of the port qfs , although it is fixed in MPEG-4 SP. Finally, the actor labeled C serves as a common output that gathers the data produced by the five optional actors U .

The work of Ko & Bhattacharyya (2005) presents a very similar modeling approach as the one described in this section. Their work is also based on PSDF semantics and the authors also propose that the schedules of static graph partitions are computed at design time and are stored for use at run-time.

5.4 Scheduling

In this section, we show a way to create low overhead (multiprocessor) schedules for systems such as the one shown in Figure 27. The scheduling consists of two parts: the off-line (design time) part and the run-time part. We assume that the run-time system consists of a set of PEs that is fixed at design time (*i.e.* we do not account for the possibility of a PE failing or the addition of extra PEs during run-time). Scheduling encompasses the regular CAL actors within the CAL network that is to be scheduled. If the CAL network that is to be scheduled is a part of a larger network, the outlying actors are not considered here.

5.4.1 Off-line scheduling

The first step of off-line scheduling is to assign each HSDF actor of T to one of the PEs in the system. Each actor is mapped to exactly one PE; each PE may be responsible for any number of actors. The majority of the actors of T originate from unrolled loops. This means that there are several calls to the same function. If the actor network is executed on a multiprocessing system, it is advisable to map these instances of the same functionality to the same PE to avoid replicating functionality. The actor instances of the same function are connected with control flow edges (see Subsection 5.2.3), which ensures the correct order of execution.

To produce fully defined off-line schedules, the latency of each actor in T must be fixed. If the state transition of a CAL actor has some variance in the latency, the HSDF actor must assume the worst-case latency for successful off-line scheduling. Fixed execution times are especially beneficial for generating multi-PE schedules, since it allows inter-PE communication issues to be resolved off-line.

The scheduling of the functionality inside D is not considered here since it requires a fully dynamic run-time scheduling approach, which is beyond the scope of this work. Here, the functionality of D can be assumed to be executed on a single PE in a sequential fashion, which does not require any special methods. In contrast, the graphs U are scheduled off-line using a fully static scheduling algorithm. We do not discuss any of these here, since there are plenty of off-line scheduling methods available. The book by Sriram & Bhattacharyya (2000) contains a good overview of the available methods.

5.4.2 On-line scheduling

On-line scheduling takes place when the system is actually running and computing. In the approach presented in this thesis, the run-time scheduling effort is actually limited to selecting a pre-computed schedule out of the ones generated and stored at the time of system design.

With regard to Figure 27, the suitable schedule is selected based on the token that comes from the dataflow input to the subunit graph. According to the view of the PSDF model, the schedule is selected by toggling the parameterized edges (a , b , c , d and e) and enabling *one* of the graphs in U .

At run-time, the system consecutively executes pre-computed schedules in an order which is unknown at system design time. This raises a question how to merge two consecutive multi-PE schedules together at run-time. It can be seen that the extended permutation flow shop (EPFS) model that was described in Section 4.4 fits exactly the problem at hand.

5.5 Experiments

The transformation and off-line scheduling steps described in Sections 5.2 and 5.4 have been implemented to a great extent in Java. In the earlier phases of the transformations, the EFSMs are represented with classes provided by the JGraphT package (Naveh & Sichi 2008) and during the later stages the HSDF graphs are represented with classes from the SDF4J package (Nezan *et al.* 2008). All of these steps are performed in the same OpenDF environment that the code generators (Wipliez *et al.* 2008, Janneck *et al.* 2008) use, which enables smooth interoperability.

On a workstation, the transformations of the RVC MPEG-4 SP decoder proposed in this chapter improved the frame rate of video decoding by 83%. This number was

acquired with the C language version of the network and by executing all the actors of Figure 6 in a quasi-statically scheduled manner. Including all the actors to quasi-static scheduling required an updated version of the Inverse Scan actor.

Our quasi-static scheduling approach is capable of producing multiprocessor schedules. However, verifying the functionality of the multiprocessor schedules requires a multiprocessing system that supports the execution of the RVC decoder. Establishing such a system is a demanding task that requires a considerable amount of work. However, from the run-time EPFS scheduling point of view, such a system is no different from that described in Chapter 8.

5.6 Discussion

The preprocessing steps (identifying P and token rates) discussed in Subsection 5.2.1 are performed completely manually at the moment. However, some work has been done to derive the token communication patterns automatically, but this functionality has not yet been tested in conjunction with our scheduling approach.

The strict requirements of the assumed system (See Subsection 5.2.1) allow quasi-static scheduling for systems that have dynamic functionality only before the regular functionality. Generally, there can be CAL actor networks that might have a dynamic part at the end of the network, or consist of several mixed regular and dynamic patches. Extending our approach to such general systems is a clear direction for future work.

6 Efficient run-time sequencing of multiprocessor schedules

In Chapter 4, the modeling of applications was discussed with the conclusion that static parameterized synchronous data-flow schedules can be interpreted as (extended) permutation flow-shop jobs. In this chapter, we shall discuss different ways of implementing pure flow-shop scheduling algorithms efficiently, and also study the relationship between the scheduling algorithm performance and scheduling algorithm overhead. Most of the topics discussed in this chapter are a part of the work of Boutellier *et al.* (2009a).

6.1 General flow-shop scheduling

Recently, many review papers have been published that cover flow-shop scheduling methods (Bagchi *et al.* 2006, Gupta & Stafford 2006, Kis & Pesch 2005) and there is one that focuses on the special case, the permutation flow-shop problem, which is NP-hard (Framinan *et al.* 2004). The permutation flow-shop scheduling problem can be solved as such, or by formulating it as an asymmetric traveling salesman (ATSP) problem (Bagchi *et al.* 2006). This formulation enables the possibility of using ATSP solving methods for scheduling, which is interesting since a branch-and-bound based exact algorithm has been proposed for solving restricted size problems. However, most of the methods of solving permutation flow-shop problems (and ATSP-problems) are heuristics that look for a good solution in a reasonable time. Since we are interested in scheduling of short-latency tasks, we shift our attention to efficient heuristics and do not pay much attention to solutions that could provide optimal solutions. However, for scientific purposes two different ATSP solving algorithms (Carpaneto *et al.* 1995a, Brest & Zerovnik 1998) were implemented in our experimental setup.

The procedure of permutation flow-shop scheduling consists essentially of the task of finding an order of jobs that minimizes the makespan of the whole schedule. Palmer (1965) proposed a simple heuristic for finding a good order of jobs for the flow-shop problem. The method can be put simply in one clause: *those jobs, whose first operations tend to be short, should be placed first in the execution order*. The shape of each job is determined by computing a *slope index*.

Bonney & Gundry (1976) formulate Palmer's slope index equation as a sum, which

we have modified slightly by subtracting the sum from a large integer I_{MAX} to ensure that only positive slope indices are acquired. The equation used to acquire the slope index I for job n is

$$I_n = I_{MAX} - \sum_{m=1}^M \frac{2m - M - 1}{2} t_{n,m}, \quad (1)$$

where m is the machine index (out of a total of M machines) and $t_{n,m}$ the time of job n on machine m . The order of the jobs in the schedule is then decided by sorting the jobs to an order of ascending slope indices.

As the overhead of scheduling is not of interest in normal (production-line scheduling) uses of flow-shop algorithms, the authors of the methods have only aimed at minimizing the makespan of the produced schedule. In scheduling of short-latency PEs, the overhead of scheduling is equally important as the produced makespan. We have found that scheduling jobs in the order in which they arrive at the scheduler produces good results for short-latency PEs with minimal schedule computation time overhead. Next, we will take a look at a few practical flow-shop scheduling algorithms, and discuss their efficient implementation.

6.1.1 No-wait timetabling without job ordering

This approach involves scheduling the operations in the order they arrive at the scheduling algorithm, and timetabling the operations according to the no-wait principle (Section 4.2). Figure 28 shows the essence of no-wait timetabling: the jobs are rigid operation sequences, which the scheduler has to slide as close to each other as possible without overlapping any two operations. In the situation of Figure 29(a), all jobs are as close to each other as possible, without overlapping anywhere or violating the no-wait condition.

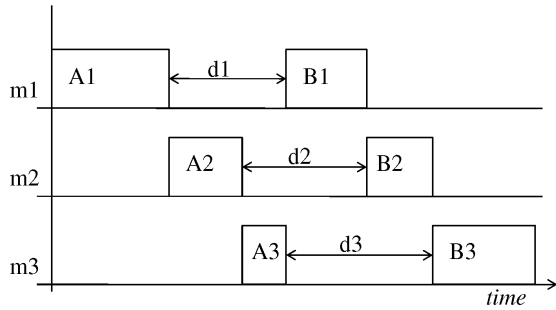
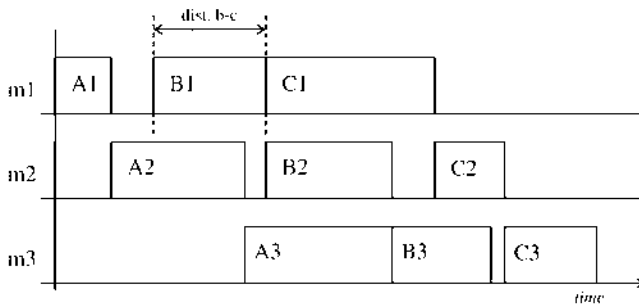
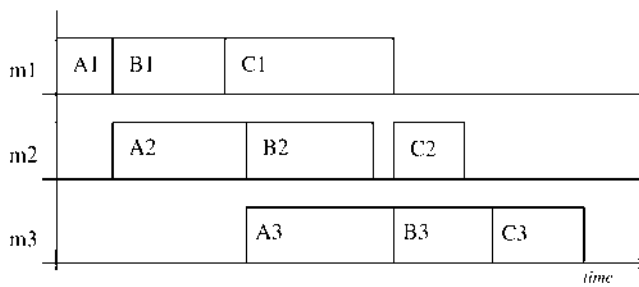


Fig 28. A Gantt chart of the design-time computation for no-wait inter-job distances.



(a)



(b)

Fig 29. A Gantt chart of a no-wait (a) and a semi-active (b) schedule. (Boutellier *et al.* 2009a, modified by author with permission of Springer.)

For a given job order, the optimal offset d_{opt} between two jobs is determined by computing the distance d_m between consecutive operations on each machine m and then taking the minimum of all $d_m, 1 \leq m \leq M$, where M is the number of machines in

the system.

$$d_{opt} = \sum_1^M t_{n,m} - \min(d_m), m = 1, \dots, M. \quad (2)$$

This computation is illustrated in Figure 28 for two arbitrary jobs: solving $t_{A,1} + t_{A,2} + t_{A,3} - \min(d_1, d_2, d_3)$ gives the optimal offset between job A and job B. If the number of jobs to schedule is considerable or the scheduler is invoked frequently, even this simple computation will produce some overhead to the run-time system.

Thus, an optimization possibility emerges when we assume that all different job types of the system are known *a priori*: it is possible to build a look-up table at the time of design that includes optimal inter-job distances for all job combinations. The size of this look-up table is obviously $O(N^2)$, where N is the number of job types. If this optimization is performed, the only task left for run-time is fetching the distance from the look-up table and performing the timetabling for each operation, which can interestingly be implemented without any conditional expressions (if machine skipping is not implemented) as the source code in Figure 30 shows.

```
void NoWait(unsigned int* jobs, unsigned int job_count)
{
    unsigned int jobtime = 0;          // cumulative job starting time
    unsigned int prev_jobtype = JOB_TYPE_COUNT; // <= special index to an
    unsigned int job_ind;              // array location holding
                                        // zero
    for(job_ind = 0; job_ind < job_count; job_ind++) // loop through jobs
    {
        unsigned int op;               // variable for op. loop
        unsigned int time = 0;         // cumul. op. start time
        unsigned int jobtype = jobs[job_ind]; // get job type by index

        jobtime += dist_table[prev_jobtype][jobtype]; // accumulate job
                                                        // starting time
        for(op = 0; op < PROC_COUNT; op++) // op. loop
        {
            // implement mach. skipping
            if(op_length[jobtype][op] != SKIP_MACHINE)
            {
                exec_list[op][job_ind] = ((jobtime+time) << 8) | jobtype;
                // write operation to dispatch list and increment time
                time += op_length[jobtype][op];
            }
        }

        prev_jobtype = jobtype; // record previous job type
    }
}
```

Fig 30. The source code for no-wait timetabling.

This flow-shop scheduling algorithm offers very low overhead but unfortunately produces longer schedule makespans than more complex algorithms. The first reason for longer makespans can be detected by comparing Figures 29(a) and 29(b): the no-wait condition hinders optimal overlapping of consecutive jobs.

The second reason is less obvious and it only emerges if machine skipping is allowed and the inter-job distances are pre-computed at design time. When the inter-job distances are pre-computed for each job type pair, it is not allowed for jobs to overtake each other on any machine. Overtaking could lead to overlapping computations, which cannot be allowed in any case. Thus, when machine skipping is allowed, some pre-computed inter-job distances will be suboptimal. The optimal solution can be computed trivially when inter-job distances are resolved at run-time.

No-wait timetabling also has a benefit: the buffers between machines (PEs) only need to store intermediate results for one job at a time, even when machine skipping is allowed (Wisner 1972). For semi-active timetabling the buffers may need to store intermediate results of several jobs. Consider Figure 31: operations A1, B1 and C1 finish before A2 finishes. This means that the outputs of B1 and C1 need to be stored somewhere until B2 and C2 start. This was already noticed in the work of Ramamoorthy & Li (1974). In the next sections, the no-order, no-wait approach will be described with the abbreviation *NoNw*.

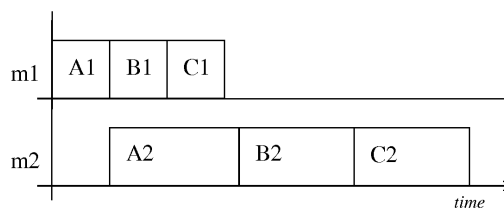


Fig 31. Semi-active timetabling requires intermediate buffers.

6.1.2 Semi-active timetabling without job ordering

This approach is basically the same as the previously described *NoNw*, except that the timetabling method is changed to semi-active. The design-time computation of inter-job distances cannot be done with this solution, since the operation start times within a job are not fixed. Thus, the number of run-time computations is necessarily higher. Figure 29(b) shows the same jobs as Figure 29(a), except that the schedule has been

constructed with semi-active timetabling. Operation B1 can now start immediately after A1, since the waiting time between operations within a job need not be zero anymore. As a consequence, the overall makespan improves somewhat over *NoNw*. The source code for implementing this algorithm can be seen in Figure 32. This algorithm will be named *NoSa* in future references.

```

void SemiActive(unsigned int* jobs, unsigned int job_count)
{
    unsigned int job_ind;           // loop variable
    unsigned int proc_in_use[PE_COUNT]; // array for keeping track of
    // available PE time slots
    memset(proc_in_use, 0, PE_COUNT*sizeof(unsigned int));

    for(job_ind = 0; job_ind < job_count; job_ind++)
    {
        unsigned int op;           // operation loop variable
        unsigned int time = 0;     // cumul. op. start time
        unsigned int jobtype = jobs[job_ind]; // get job type by index

        for(op = 0; op < PE_COUNT; op++)
        {
            // implement mach. skipping
            if(op_length[jobtype][op] != SKIP_MACHINE)
            {
                // make sure that operation
                // does not start until
                // PE is available
                if(time < proc_in_use[op])
                    time = proc_in_use[op];

                exec_list[op][job_ind] = (time << 8) | jobtype;
                time += op_length[jobtype][op];
                // write operation to dispatch list and increment time
                proc_in_use[op] = time;
            }
        }
    }
}

```

Fig 32. The source code for semi-active timetabling.

6.1.3 Palmer job ordering

The job ordering heuristic by Palmer (1965) is essentially a slope-index sorting operation. The amount of run-time processing can be minimized by computing the slope indices to a look-up table at compile time. Again, this is only possible if we assume knowledge of all job types at the time of design. When using the no-wait approach, timetabling can be optimized as in Subsection 6.1.1, by the $N * N$ look-up table. Machine skipping was implemented by using Palmer's slope formula as usual and by setting the execution times of skipped operations to zero. Later in this thesis, the algorithm combining Palmer's job ordering and semi-active timetabling is named *PaSa*.

6.1.4 Job ordering by ATSP solving

The formulation of the permutation flow-shop problem into an asymmetric traveling salesman problem has two phases: (1) determining the respective parts of *intercity distances* in the flow-shop problem, (2) the use of dummy jobs. Determining the intercity distances (Bagchi *et al.* 2006) is actually the same operation as determining the inter-job offsets that was described in Subsection 6.1.1. Similarly, the intercity distances can be precomputed at design time.

The use of dummy jobs can be done according to Wismer (1972), which results in two dummy jobs for each scheduling problem. Therefore a flow-shop problem with N jobs results in a $(N + 2) * (N + 2)$ ATSP matrix. For practical run-time scheduling, optimal ATSP solving is not feasible because of its high computational complexity. However, it can be used to evaluate the performance of permutation flow-shop heuristics.

Carpaneto *et al.* (1995a) have published a method of solving constrained size (up to 2000 vertices) ATSP problems. The implementation of this algorithm has been shown in Carpaneto *et al.* (1995b) and was included in our experimental setup through the formulation of Wismer. The output of the algorithm by Carpaneto *et al.* (1995b) is very valuable, because it generally solves small sized problems (as those of ours) optimally, and separately announces when the result is sub-optimal. Thus, their algorithm can be used as a reference for evaluating the performance of other approaches.

Brest & Zerovnik (1998) have proposed a heuristic named RAI (Randomized Arbitrary Insertion) to solve ATSP problems. Their algorithm was also included in our experimental setup through the formulation of Wismer. Although the algorithm by Brest & Zerovnik (1998) is not optimal, it produced almost equally good results to those with the algorithm by Carpaneto *et al.* (1995a). We measured the computation time of the algorithm of Brest & Zerovnik (1998), and it was evident that this algorithm is not feasible for run-time determination of job orders.

The formulation of Wismer (1972) is only intended for no-wait flow-shop problems, but we also performed an experiment, where the job order indicated by the RAI algorithm was used with semi-active timetabling. Bagchi *et al.* (2006) also indicate that there are other ATSP formulations that could be used to achieve approximate solutions to the semi-active case.

6.2 The performance of flow-shop algorithms

To evaluate the performance of the presented permutation flow-shop heuristics, the algorithms were implemented in *C* language and executed on an Altera Cyclone III FPGA that runs a NIOS II/f soft processor. The experiments conducted will first be introduced, and the numerical results will then be shown and analyzed.

Altogether there were five different scheduling method combinations that were tested on the NIOS soft processor. We were mainly interested in the trade-off between the average makespan of the produced schedules and the average time used in building the schedules. The scheduling procedure consists of two major operations: job ordering and timetabling. *No ordering*, Palmer's job ordering heuristic and the RAI heuristic (RaNw) were tested and timetabling was done with no-wait and semi-active approaches, giving a total of five algorithms.

In the first experiment, we used a generalized model of multi-stream MPEG-4 video decoding (similar to the work of Schumacher *et al.* (2005)): 20 random job types were defined, some of which could skip some of the six machines in the system. There were four streams with one to six random jobs each, which resulted in four to 24 jobs per scheduling problem. The experiment was conducted on various average operation lengths to see which scheduling factors are independent of the operation length. The standard deviation of operation lengths was about 50% of the average operation length. The scheduling process was iterated 10000 times with different job numbers and job types to get reliable average results. Table 4 shows the results for each algorithm. The following notation is used:

T_s average scheduling time,

L_s average sequential schedule makespan,

L_p average parallel schedule makespan,

L_{pX} average parallel schedule makespan for average operation length X ,

T_t sum of average scheduling time and average parallel schedule makespan,

L_o average scheduled operation length,

N_o average number of operations to be scheduled for each scheduler invocation,

K_p throughput compared to sequential execution.

The average number of operations that were scheduled with one scheduler invocation was 45. With this information, it was possible to define a *speedup factor* for each

scheduling algorithm. The speedup factor tells us what is the best achievable program speedup with the respective scheduling algorithm and parallel machines. The factor expresses the speedup compared to sequential execution of the same program on one machine. Thus, a speedup factor of 2.00 for scheduling algorithm A would mean that if A is used to schedule a parallel system, it will run twice as fast as a one machine solution. The speedup factor is dependent on the number of parallel machines in the system, as well as average lengths and numbers of operations in the scheduled jobs.

The same experiments were also carried out with a different set of jobs that did not skip any of the six machines. Again, 20 random job types were defined so that there were four streams with a maximum of six random jobs each, which resulted in a scheduling problem of up to 24 jobs. 10000 iterations were used also in this experiment. Without machine skipping the average number of operations per scheduler invocation was 84. Table 5 shows the results for this experiment.

Table 4. Execution times and makespan results with machine skipping. (Boutellier *et al.* 2009a, modified by author with permission of Springer.)

Algorithm	T_s	L_{p100}	L_{p500}	L_{p1000}	K_p
NoNw	2567	2333	11618	23222	1.93
PaNw	4600	2342	11479	22945	1.94
RaNw	$5.5 * 10^6$	1339	6669	13329	3.36
NoSa	3182	1723	8578	17146	2.61
PaSa	4862	1487	7401	14794	3.03
Seq.	0	4499	22418	44813	1.00

Table 5. Execution times and makespan results without machine skipping. (Boutellier *et al.* 2009a, modified by author with permission of Springer.)

Algorithm	T_s	L_{p100}	L_{p500}	L_{p1000}	K_p
NoNw	3712	3063	15230	30441	2.76
PaNw	5677	2876	14304	28591	2.94
RaNw	$5.5 * 10^6$	2404	11965	23920	3.52
NoSa	5057	2450	12191	24371	3.45
PaSa	6312	2300	11442	22875	3.68
Seq.	0	8446	42066	84103	1.00

Tables 4 and 5 reveal that Palmer's job ordering implemented by insertion sort considerably slows down the scheduling, but provides improved throughput. However, the effect of timetabling is more significant than the effect of job ordering. The RAI algorithm performs remarkably well when machine skipping is allowed, but does not offer equally remarkable results when machine skipping is disallowed. Unfortunately, the huge computational overhead of RAI makes it inappropriate for run-time scheduling of fine-grained tasks.

The NIOS II processor used had 2kB of data cache and 2kB of instruction cache and ran at 50MHz. The whole benchmark program was executed from a 32kB on-chip memory, of which 22kB were used for the program and data. We also conducted brief tests to see how the algorithms slow down when there is no on-chip memory available (except caches) and everything is on off-chip SDRAM: the algorithms slowed down at most 29%.

The aforementioned algorithm by Carpaneto *et al.* (1995b) was tested in the same experimental setup, but on a workstation. Surprisingly, the difference between the performance of RAI and the algorithm by Carpaneto *et al.* (1995b) was less than $\frac{1}{1000}$ in terms of makespan. The explanation behind this is that the algorithm by Carpaneto *et al.* (1995b) provides a rather poor solution when it cannot find the optimal job order. With our set of jobs it happened approximately for 1 scheduling problem out of 1000 that the algorithm by Carpaneto *et al.* (1995b) could not provide the optimal solution.

Furthermore, a brief experiment was conducted to analyze the makespan and scheduling time jitter of each algorithm. A set of four jobs was selected and all 24 permutations of those four jobs were scheduled with each algorithm. The results are visible in Table 6. Algorithms with Palmer or RAI job ordering heuristics naturally produce no jitter in schedule makespan, because the jobs are scheduled in the same order, independent of the order in which they arrive. When no job ordering is used, the makespan depends on the job arrival order. The scheduling time jitter for all algorithms except RAI was around 3%, and for RAI it was slightly larger: 8%. In this experiment, all variables except the order of jobs was fixed.

Table 6. Makespan jitter, scheduling time jitter, and speedup as a function of machine count.

Algorithm	L_p avg.	L_p jitter (%)	T_s avg.	T_s jitter (%)	$S_{M=4}$	$S_{M=6}$	$S_{M=10}$
NoNw	1193	11.1	1203	3.3	1.30	1.47	1.77
PaNw	1061	0.0	1643	2.4	1.16	1.38	1.84
RaNw	1060	0.0	260000	8.1	0.01	0.01	0.02
NoSa	1138	9.1	1443	2.4	1.45	1.69	1.97
PaSa	1054	0.0	1817	2.8	1.42	1.69	2.07

Table 6 also shows the effect of the number of machines in the system to be scheduled. In this experiment, the number of jobs to be scheduled ranged between 2 and 12, and the number of machines was 4, 6 or 10. For each machine count, a different set of jobs was defined, but in all job sets 42% of the machines were skipped. The average length of operations was 500 for all job sets. The results show that Palmer's scheduling heuristic provides better results when the number of machines increases. Heuristics using no job ordering suit better those systems that have few machines. This phenomenon can be explained by the complexity of timetabling and job ordering: in the algorithms presented here, the complexity of job ordering increases as the number of jobs grows, but is not affected by the number of machines. On the other hand, the complexity of timetabling grows both as a function of the machine count and the number of jobs.

To help the analysis of the results presented in the previous section, a couple of equations were derived. With the help of these equations, it was possible to draw informative graphs that reveal interesting facts about the scheduling algorithms and the problem at hand. It was decided that the performance of each scheduling algorithm could be analyzed best by comparing it to the performance of sequential uniprocessor execution. For this, we shall define the speedup factor S :

$$S = \frac{L_s}{T_t}. \quad (3)$$

The total time used by parallel execution can also be expressed by

$$T_t = L_p + T_s \quad (4)$$

and the length of the parallel schedule can be defined as follows:

$$L_p = \frac{L_s}{K_p}. \quad (5)$$

K_p is a scheduling algorithm-specific value that depends on the number of operations scheduled and number of machines in the system. However, it is not dependent on the average operation length L_o . Now the speedup coefficient can be rewritten as

$$S = \frac{L_s}{T_s + \frac{L_s}{K_p}}. \quad (6)$$

Statistically, it also holds that

$$L_s = L_o N_o. \quad (7)$$

Finally, we can formulate the equation for S in its final form:

$$S = \frac{1}{\frac{T_s}{L_o N_o} + \frac{1}{K_p}}. \quad (8)$$

With the help of this equation, we can draw two different figures for each scheduling algorithm. As it can be seen from Tables 4 and 5, T_s and K_p are constants when the number of processors (machines) and the set of jobs remain unchanged. The first figure is created from the data set shown in Table 4.

The curves in Figure 34 have also been computed with the help of equation 8. This figure displays the speedup factor as a function of the number of tasks to schedule for $L_o = 500$. As K_p is dependent on the number of tasks (operations) scheduled, the speedup factor had to be computed discretely on the basis of a set of measured K_p values. It can be seen that run-time scheduled parallel execution starts to provide a benefit over uniprocessor execution already with less than 10 tasks. Again, these results only apply for $L_o = 500$, a system with six PEs and the set of jobs that was used in this experiment.

Figure 33 shows the speedup provided by the presented scheduling algorithms, as a function of average operation length. This result applies to the randomly generated job set that does machine skipping and maps the task to six PEs. The curves in Figure 33 were computed with equation 8 from the Table 4 data set. Figure 33 tells that parallel processing with run-time scheduling starts to provide a benefit over sequential uniprocessor execution when the average operation length is more than 150 clock cycles. Also, it can be seen that the best speedup is provided by the NoSa algorithm when the average operation length is less than 700. For longer operations, the more complex *PaSa* algorithm provides better results. This is due to the fact that *PaSa* offers a better throughput, but requires a longer time to compute.

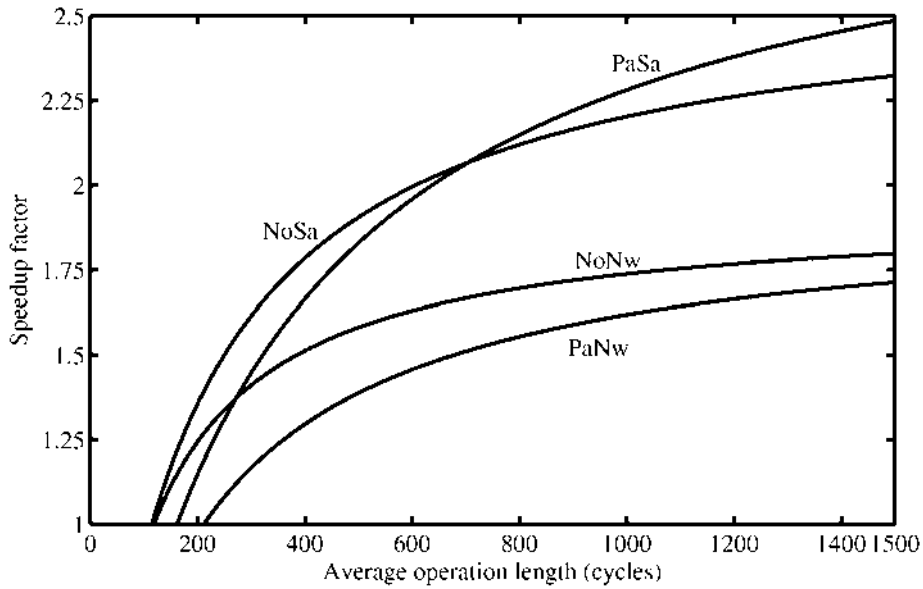


Fig 33. The speedup provided by run-time scheduled 6 parallel machines. (Boutellier *et al.* 2009a, modified by author with permission of Springer.)

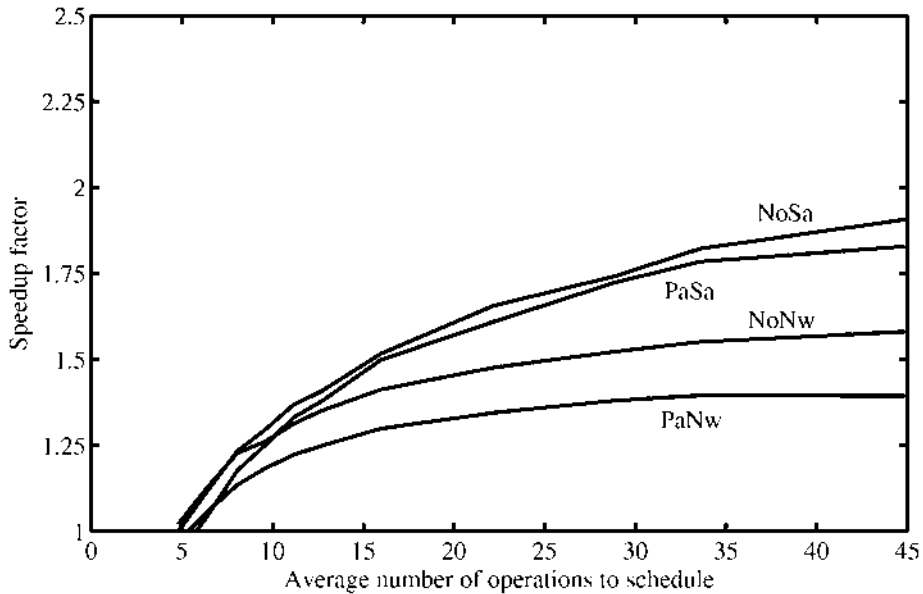


Fig 34. Speedup provided by run-time scheduled 6 parallel machines, as a function of task count. (Boutellier *et al.* 2009a, modified by author with permission of Springer.)

Similar results were also computed with the data of Table 5. To save space, these results are only explained briefly without figures. The data set of Table 5 differs from the data set of Table 4 only by the different set of jobs, which do not do machine skipping. As a result, the scheduling algorithms had more operations to schedule per job. That increased the scheduling time, but on the other hand provided better speedup factors by better PE utilization. These consequences can also be observed directly from Table 5 by looking at the columns T_s and K_p .

6.3 Discussion

The analysis shows that with our experimental setup, run-time scheduling starts to be beneficial when the scheduled tasks are longer than 150 clock cycles. Unfortunately, some of the functions that were mentioned in Section 2.1.1 are shorter than this, if they are implemented as LSI circuits (Rambaldi *et al.* 1998). Therefore, the C-code implementations of the scheduling algorithms presented do not make the use of fine-grained PEs feasible.

Optimizing the results by re-writing the algorithms in assembly could improve the results somewhat. Instead of doing this, we have directly implemented a dedicated LSI circuit for flow-shop scheduling, which is described in Chapter 8. The LSI circuit schedules a job (independent of the number of PEs) in 3 clock cycles using the *NoNw* algorithm. Before describing the hardware solution, however, the next chapter explains the influence of memory architectures on scheduling.

7 The influence of memory architectures on flow-shop scheduling

Up to this point, the system architecture interconnecting the PEs has not been discussed and the approaches presented have been rather theoretical. In practice, the architecture of the multiprocessor system dramatically affects the scheduling of PEs. In this chapter we shall investigate the influence of the memory architecture on scheduling.

Difficulties emerge with memories in multiprocessor systems when several PEs have access to the same memory resource. When this is the case, the memory accesses need to be synchronized in some manner (Graunke & Thakkar 1990) to avoid conflicts. Synchronization makes sure that consecutive reads and writes to the memory are performed in the correct order to avoid reading stale data. Furthermore, if the shared memory allows multiple simultaneous accesses (Patel *et al.* 2004), the synchronization algorithm must ensure that the number of PEs reading the memory does not exceed the maximum number of allowed simultaneous accesses.

Figure 35 shows a part of a system architecture that involves a set of pipelined PEs. PEs that are interconnected as in this figure, can be scheduled directly by the methods that have been discussed in Chapter 6, without attention to memory accesses. An alternative solution would be the use of multi-port memories discussed in Section 7.2.

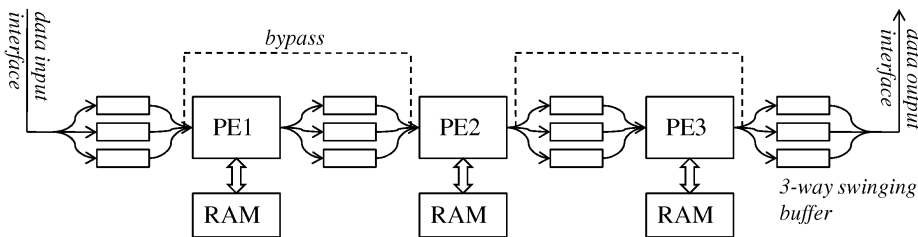


Fig 35. An accelerator pipeline with multi-way swinging buffers.

Each PE has a dedicated RAM that contains the data and program memories required by that PE. This arrangement makes sure that the program code and variables can be accessed by each PE without resource sharing problems. The PEs exchange data by buffer memories that are placed between the PEs. Because of machine skipping (See Section 4.2), each buffer memory has equally many buffer elements as there are PEs. In

principle, machine skipping can allow all PEs to access the same buffer simultaneously. Figure 35 also contains two arcs drawn with dashed lines. These arcs depict *bypasses* that allow implementing machine skipping.

Unfortunately, this memory architecture that is called *swinging buffers*, occupies more silicon area than the same amount of buffer space organized as monolithic buffers, which is undesirable. In this chapter, we shall replace the swinging buffers with larger memory structures, and see how the scheduling algorithm must be changed to cope with the new memory architecture. We shall also explore the use of a single shared memory as a communication buffer between several PEs. Before going into these topics, the next section explains how the flow-shop model restricts the choice of memory interconnects.

7.1 Generalizing the pipelined interconnect

The flow-shop model offers a constrained way of expressing scheduling problems. Previously, we have introduced the theoretical view of flow-shop scheduling, and now we shall inspect the implications of the model for the memory architecture. Figure 36 depicts a multiprocessing system with three PEs and four monolithic communication buffers organized into an interconnection matrix. Each PE and memory element has separate read and write ports.

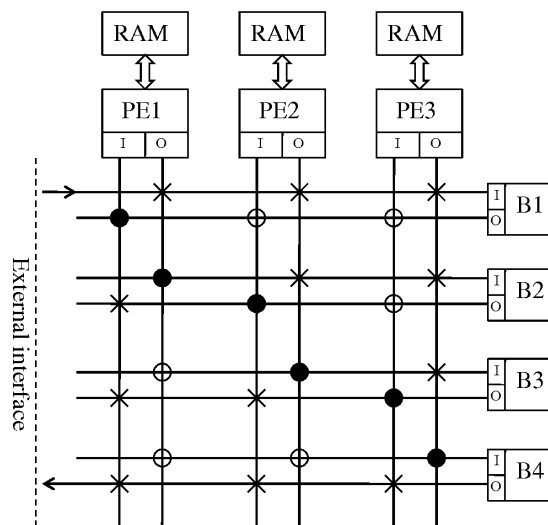


Fig 36. An accelerator pipeline as an interconnection network.

The bus interconnects have three kinds of markings: solid black circles, transparent circles and Xs that categorize the connections into three groups. The set of connections that are represented with solid circles are the group that establishes a pipelined interconnect between the PEs and memory elements. If all the other connections are discarded and only the solid circles are taken, the system interconnect becomes a pipeline without bypasses.

The second group of markings are depicted as transparent circles. Enabling these connections yields a pipelined processor-memory architecture with all bypasses enabled. This means that the output of each memory element i can be redirected to any of the PEs in the pipeline after i . Similarly, the output of any PE m can be redirected to any of the memory elements after m .

The last group of connections, labeled with Xs, represents connections that would produce a *feedback* if the memories and PEs are organized into a pipeline. Enabling all of these connections would result in a fully connected interconnection network, *i.e.*, a *crossbar* that has exponentially growing complexity.

If we go back to the theory of the flow-shop problem, we remember that one of the restrictions in the flow-shop model is that all jobs need to advance from one machine to the next in *the same order*. This means that feedbacks are not allowed in the flow-shop model. On the other hand, forward bypassing poses no problem to flow-shop theory, as it is interpreted as *machine skipping*.

The disallowance of feedbacks means that the system designer needs to place the PEs and memory elements into an order that has to remain fixed when the system is performing computations. If the PEs are labeled $P_1 \dots P_M$, ($1 \leq m \leq M$) and the memory elements $E_1 \dots E_{M+1}$ ($1 \leq i \leq M + 1$), the data under processing has to flow from PE m to memory element i ($i > m$) or respectively from any memory element i to one of the processing elements m ($m \geq i$). If the flow-shop scheduled system is multitasking and several of the running applications share some PEs over a period of time, this PE order applies to all of the running applications: each application must use the PEs in the same order.

Although this limitation seems quite restrictive, there is a major benefit that this limitation produces and it also explains the efficiency of flow-shop scheduling: since all jobs use the machines (PEs) in the same order, the shape of the jobs naturally tends to produce high PE utilization over time. If we inspect Figure 29(b), it is easy to notice that no matter how jobs A, B and C are ordered on the time axes, they will always overlap in time so that parallelism follows naturally.

If the flow-shop limitation of disallowing feedbacks is abandoned, the system has to be modeled with a more general *job-shop* (French 1982) model that allows changing the order of machines between jobs. This is illustrated in Figure 37 that depicts an arbitrary job-shop problem with three jobs. Jobs A and C use the machines in the same order, whereas job B uses them in the opposite order. On the left side of Figure 37 the job-shop problem is solved in the way the flow-shop schedules are generated: by sliding the jobs as close to each other as possible. However, it can be seen that this approach produced low machine utilization and a long makespan. It is evident that job-shop systems cannot be scheduled as easily as flow-shop systems. The right side of the same figure shows the correct approach in job-shop scheduling that involves interlacing of jobs and this reduces the makespan of the solution considerably. However, the number of computations required for computing efficient schedules for interlaced jobs is much greater than that of flow-shop sequencing.

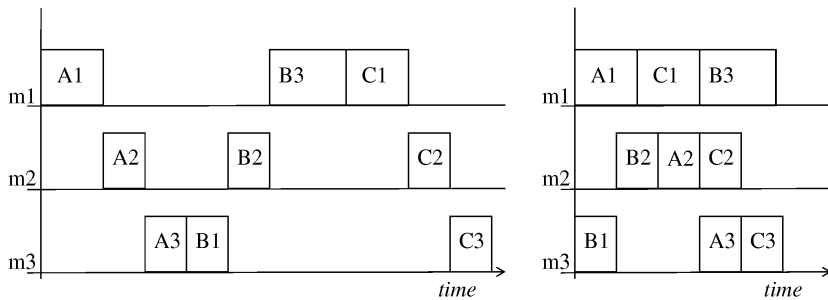


Fig 37. An arbitrary job-shop schedule.

For applications that would require some kind of feedbacks between memories and PEs, there are two workarounds that still keep the system as a flow-shop. The first possibility is to replicate some of the PEs. Needless to say, this solution is costly as it means placing an extra PE and buffer into the pipeline, but on the other hand, both of the replicated PE instances are also available simultaneously.

The second solution is to perform reconfiguration of the interconnection matrix between the scheduler calls, so that the order of the PEs in the execution pipeline is changed. However, this solution cannot simultaneously support jobs that use the machines in different orders.

Now we have seen how the flow-shop model limits the architecture of a multiprocessing system, as well as how it naturally leads to high PE utilization. The properties of

the flow-shop model fit well the description of desirable embedded system properties as formulated by Wolf (2006): "[Embedded systems] should be programmable enough to make the design flexible and long-lived, but need not provide unnecessary flexibility that would detract from meeting system requirements".

In the next section, we shall look at the design space of the memory solutions that we are going to explore.

7.2 The memory architecture design space

A multi-port memory allows more than one simultaneous access to the same memory element. Most usually, the number of ports is two, and rarely more than four. Additional memory ports come with a cost: address and data pins, word-lines and the bit-lines are duplicated, which adds to the silicon area and power consumption of the memory element. (Patel *et al.* 2004)

Although the multiprocessor memory architecture can be designed in numerous ways, we will mostly¹ concentrate on two specific memory architectures. The first approach is based on a single memory element that is shared by all M PEs through k memory ports and a specialized flow-shop scheduling algorithm. The second approach places $M+1$ memory elements around the M PEs and strictly avoids overlapping memory accesses through the special attention of the scheduling algorithm.

These two approaches can be considered as extreme points of a two-dimensional design space that is depicted in Figure 38. The horizontal axis in the figure depicts the number of memory units that are available to the PEs and the vertical axis the number of ports in each memory. The *shared memory* approach that is introduced here, occupies the leftmost column of the discrete design space, although we only explore the solutions between one and three memory ports. The second solution, *parallel memories*, maps to the low edge of the rightmost column of the figure. We shall explore the solutions with one and two ported buffers. On the top-right edge of the figure, the swinging buffer solution is depicted: it has M memory ports in the form of separate memory elements, for each of the $M + 1$ buffers.

¹Section 7.7 discusses arbitrary pipelines

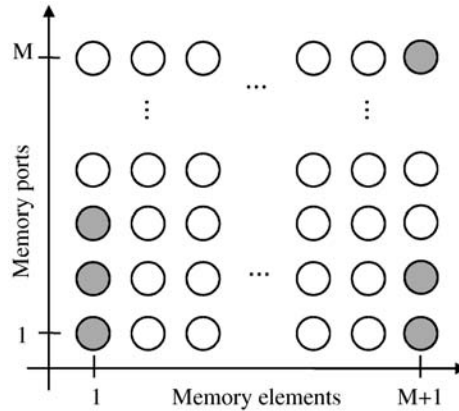


Fig 38. A design space of memory solutions.

Application specific solutions with $1 < \text{memory elements} < M+1$ are briefly covered in Section 7.7. In these solutions, it is not unambiguous which memory serves as an input for PE m and which memory serves as an output. In the next section, we shall look at the solutions that are located on the leftmost edge in the design space: a shared memory with various numbers of ports.

7.3 Flow-shop scheduling with a single shared memory

Figure 39 shows a shared memory connected to three PEs over a bus. A shared memory is a commonly used means of inter-processor communication for multiprocessor systems (Nedjah & de Macedo Mourelle 2007, Wolf 2003). Regarding the issues discussed in this thesis, there are two facts that make a single shared memory an attractive architecture. The first benefit is that a shared memory has minimal hardware overhead with regard to address and data pins, word-lines and the bit-lines (Patel *et al.* 2004). The second benefit is that a shared memory used by several PEs theoretically allows the system to process data without copying it from one memory location to another. However, regarding the second benefit, we have to remember that for true parallel processing each PE needs to have access to an input and output storage that is not simultaneously accessed by other PEs. In our experiments, all PEs always had separate input and output storages. Also, accesses to a large monolithic memory consume more energy than accesses to a small memory block (Patel *et al.* 2004, Benini *et al.* 2002, Steinke *et al.* 2002).

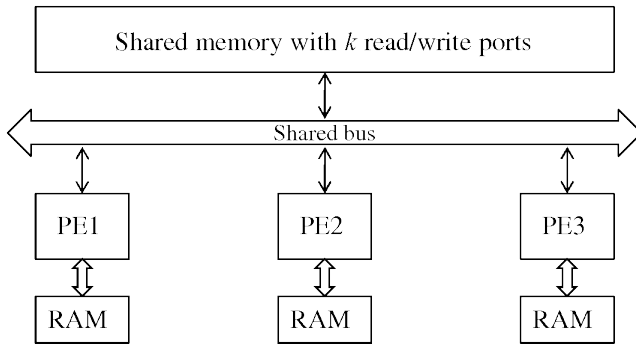


Fig 39. A shared memory architecture.

In pure flow-shop scheduling that does not consider memory architectures, each task (operation) consists of one component in the Gantt chart that depicts a schedule. Now that the memory accesses have to be considered in the scheduling, the task model must be updated. We divide each task into three parts: *memory block read*, *processing* and *memory block write*. Memory block read/write means reading/writing all the data that the PE requires for processing one data set. It can be, for example, 64 bytes of an 8x8 IDCT operation. For now, we shall assume that these three components of a task can be separated from each other, but they must always be performed in this order. Machine skipping is allowed in our model, unless it is explicitly denied. The new task model is depicted in Figure 40.

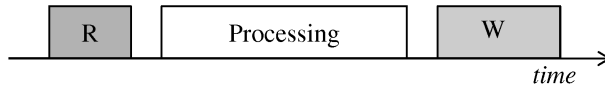


Fig 40. The three-part task model. R is a buffer read and W is a buffer write.

Cortés *et al.* (2005) have also considered inter-processor communication issues in their quasi-static hard real-time scheduling solution. They have regarded the inter-PE communication buses as PEs, and communication activities as tasks. In our three-part task model, the communication tasks are mapped to the same PE time axes as the processing (see Figure 40), whereas Cortes has separate axes for communication buses and tasks.

When a single shared memory is used, the distribution of the memory access timeslots becomes a central issue. Since all PEs access the same memory, both during their read and write phases, there is easily a shortage on access slots. This means that the

distribution of access time slots must be done carefully. At the same time, it is evident that as the complexity of the access time sharing algorithm rises, it also means that the scheduling overhead increases.

For our experiments, the scheduling algorithm for a shared memory architecture was derived from the *NoSa* algorithm that was shown in Subsection 6.1.2. The algorithm was modified to the new three part task model and a *quantized reservation vector* was added for keeping track of granted memory accesses.

Since the overhead of run-time scheduling has to be considered, the complexity of the shared memory scheduling algorithm has to be kept low. Thus, durations of tasks to be scheduled were *quantized* so that the quantization factor was set to be equal to the length of the longest memory access (read or write). This way, the length of all memory accesses became 1, and the lengths of actual processing tasks became small integers (our assumption was that memory accesses are short compared to the actual processing).

When a memory read or memory write has been scheduled, the access time is recorded to one of the elements of the reservation vector. Respectively, before scheduling a memory access, the reservation vector is browsed to find a free time slot for the access. This is done by reading the vector from the earliest possible moment when the memory access can happen: that is after the previous operation of the same job has written its result in the memory. If the first possible time instant for the memory read is not available, the next possible time instant is checked, until an available slot is found.

The vector requires memory space. At a minimum, the vector requires one bit for each memory port and each time instant. In the scheduling of fine-grained tasks, the makespan of a complete schedule might be more than a thousand clock cycles.

Variable length memory accesses were also tried, but a brief experiment showed that the scheduling algorithm execution time became 4 times longer (and the average schedule makespans improved 15%), which made this option impractical².

Figure 41 shows a schedule that has been constructed with the help of a reservation vector. Again, there are six PEs in the system and a set of random jobs are to be scheduled on the PEs. Starting from job 0, the scheduling algorithm greedily schedules each operation with the memory block reads and writes. In the example in Figure 41, the reads and writes are immediately next to the processing operation, to which they are related. Generally, this need not be the case: reads and writes may be arbitrarily far from the processing operation. With the help of the reservation vector, the number of

²The memory access lengths were here between 4 and 20, and the processing task lengths were between 60 and 100.

simultaneous shared memory accesses is kept within the number of available memory ports, which is two in this example. On several occasions, as at time instant 14, memory accesses have been delayed to keep the number of simultaneous accesses two or less.

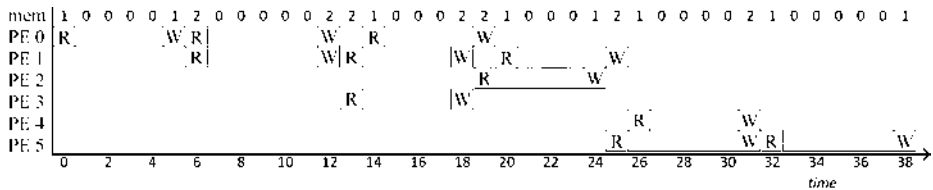


Fig 41. The quantized reservation vector (topmost row) and the corresponding schedule.

The problem of scheduling a set of hardware accelerators interconnected by a shared memory is similar to the dynamic scheduling of a programmable processor’s functional units. The famous algorithm by Tomasulo (1967) performs the scheduling of CPU functional units in a fully dynamic fashion at run-time. Using the algorithm by Tomasulo (1967) to pre-compute quasi-static schedules at design time is possible, but produces no advantage, since a reservation vector or a timestamp is needed as well.

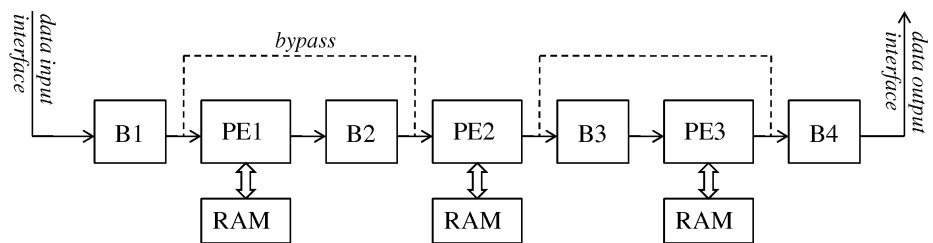


Fig 42. An accelerator pipeline with parallel memories.

7.4 Flow-shop scheduling with parallel memories

In this section, the solution from the rightmost edge of the memory architecture design space (Figure 38), parallel memories, is discussed. We shall assume that if the system has M PEs, there will be $M + 1$ buffer memories. The scheduling algorithms for the parallel memory architecture were also derived from the *NoSa* approach and the previously introduced three part task model is used. This kind of a pipeline architecture is also used in the publication of McKeown *et al.* (2008), where the pipeline bypasses are

reconfigured into different modes depending on the application requirements.

Parallel memory architecture scheduling, like the shared memory model, must keep track of buffer access times. For parallel memories, however, the tracking scheme can be simplified by just assigning a timestamp variable to each port of each buffer element. When a memory access to buffer i is requested by a PE, the timestamp variables of buffer i ports are read, and the new memory access is allowed just after the previous timestamp. Then, the timestamp variable of buffer i is updated to hold the ending time of the latest granted memory access. Note that the timestamp variables only exist within the scheduling algorithm: they are not used anymore when the schedule is executed. The downside of this solution is that gaps will appear in the memory access schedule. There is no way to keep track of these gaps, since the timestamp variable only keeps track of the last assigned memory access time.

When the granted access time slots are only tracked by single buffer and port-wise timestamp variables, the task scheduling order has an impact on the resulting schedule makespan. Two different task scheduling orders were examined: the first one, *regular*, scheduled the tasks of one job after another, in ascending order of PEs. This is shown in Figure 43 on the left.

Job 1	Job 2	Job 3	...						
1	6	11	16	21	26	31	...	PE 1	
2	7	12	17	22	27	32	...	PE 2	
3	8	13	18	23	28	33	...	PE 3	
4	9	14	19	24	29	34	...	PE 4	
5	10	15	20	25	30	35	...	PE 5	

Job 1	Job 2	Job 3	...						
1	3	6	10	15	20	25	...		
2	5	9	14	19	24	29	...		
4	8	13	18	23	28	33	...		
7	12	17	22	27	32	37	...		
11	16	21	26	31	36	41	...		

Fig 43. Task scheduling orders: regular (left) and diagonal (right).

The second task scheduling order, *diagonal*, was more complicated and it is shown in Figure 43 on the right. The numbers express the order in which the operations are added to the schedule. In this approach, the job-machine matrix is browsed diagonally. The reasoning behind this solution can be explained by the nature of pipeline schedules, such as the one shown in Figure 29(a): tasks that are assigned to PE m tend to start later than tasks assigned to PE $m - 1$. Thus, the diagonal scheduling order is likely to consecutively schedule tasks that have similar starting times in the resulting schedule.

This in turn is beneficial when buffer usage is only tracked with one variable per buffer.

7.5 Experiments

We have now described a new task model where each task has been divided into three parts. The new task model has been used to fit the flow-shop scheduling model to shared memory and parallel memory architectures. As the different memory architectures have their own characteristics in terms of hardware, they also differ in scheduler complexity. In this section, we shall evaluate the scheduling methods that are designed for scheduling jobs on shared memory and parallel memory architectures.

The experiments only point out the complexity and produced makespans of the scheduling procedures; they do not evaluate the total cost (silicon area overhead, maximum clock frequency, energy consumption) of the solutions.

The performance of the algorithms was compared to the *NoSa* flow-shop scheduling heuristic on a swinging buffer architecture that is depicted in Figure 35. From the scheduling algorithm point of view, this situation is equal to *NoSa* scheduling discussed in Chapter 6. Practically, the *NoSa* algorithm scheduled exactly the same jobs as the other algorithms in these experiments, but merged the buffer accesses as a part of the processing task.

The experiments were performed on an Altera Cyclone III FPGA running a Nios II/f soft processor. The algorithms were programmed in C language and all the program code and data resided in an on-chip memory.

The results are an average of 10000 schedule computations, and for each iteration a random set of 20 jobs was created. For the results in Table 7, a random number of one to six jobs was scheduled on six PEs. Similarly, for Table 8 a random number of four to 24 jobs was scheduled again on six PEs. In both experiments, every job had an operation for each PE with a probability of 68%. This gives an average of about four operations per job.

Table 7. Comparison of scheduling algorithms with an average of 3.5 jobs. The numbers in parentheses present execution time of the scheduler in clock cycles. The numbers outside the parentheses show the schedule makespan. The number of memory ports is k and m shows the number of memories in the system.

Mem. arch.	k	$k \cdot m$	Large	Small	None	Long access
Sequential	-	-	1603 (0)	1603 (0)	1603 (0)	1603 (0)
Swing.buff.	1	42	803 (896)	796 (931)	788 (839)	795 (931)
Par. mem.	1	7	837 (1670)	832 (1676)	827 (1591)	839 (1681)
Par. mem.	2	14	807 (2111)	801 (2054)	793 (1955)	801 (2053)
Sh. mem.	1	1	1003 (2659)	942 (2586)	849 (2331)	1104 (2601)
Sh. mem.	2	2	935 (2348)	889 (2303)	804 (2433)	976 (2303)
Sh. mem.	3	3	923 (2366)	877 (2299)	792 (2208)	957 (2295)

Table 8. Comparison of scheduling algorithms with an average of 14 jobs. The numbers in parentheses present execution time of the scheduler in clock cycles. The numbers outside the parentheses show the schedule makespan. The number of memory ports is k and m shows the number of memories in the system.

Mem. arch.	k	$k \cdot m$	Large	Small	None	Long access
Sequential	-	-	6421 (0)	6422 (0)	6422 (0)	6422 (0)
Swing.buff.	1	42	1957 (2774)	1936 (2752)	1914 (2723)	1934 (2756)
Par. mem.	1	7	2321 (6032)	2297 (6060)	2278 (6434)	2341 (6062)
Par. mem.	2	14	2046 (7876)	2022 (7886)	2000 (7893)	2034 (7887)
Sh. mem.	1	1	2844 (7783)	2603 (7895)	2324 (7513)	3470 (8111)
Sh. mem.	2	2	2333 (7451)	2211 (7538)	1997 (7159)	2464 (7624)
Sh. mem.	3	3	2266 (6866)	2153 (6916)	1946 (6948)	2360 (6955)

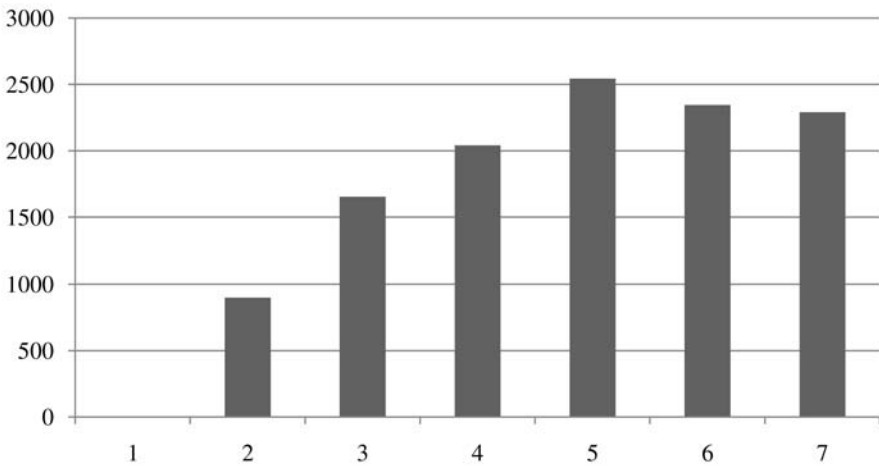
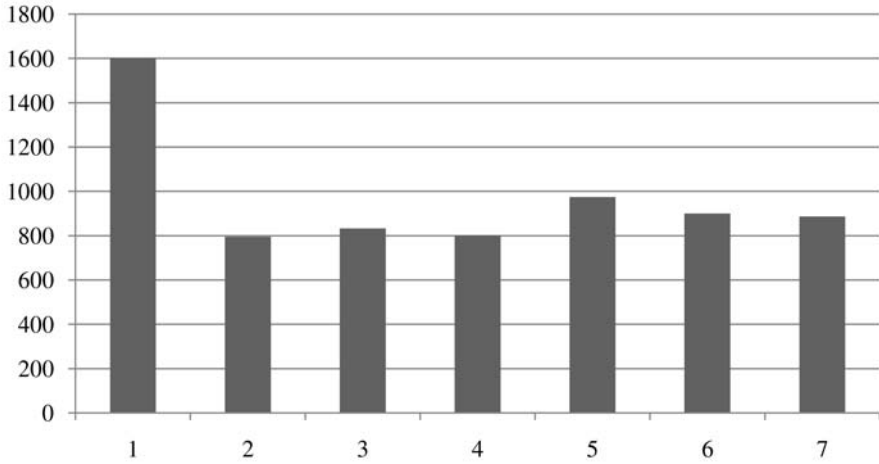
The experiment with one to six jobs reflects a situation where the multiprocessing application is not very predictable and can provide only a small set of jobs at a time to be executed in parallel. The experiment with four to 24 jobs gives directions how the algorithms would perform when the PE pipeline would be continuously in use.

Both Tables 7 and 8 have four columns of results; each column represents an experiment with somewhat different parameters. In each table cell, the number outside the parentheses gives the resulting average schedule makespan (in cycles) and the number in the parentheses the average number of CPU cycles used to compute the

schedule. For the experiments of the columns *Large*, *Small* and *None*, the average length of tasks was 80 cycles and the average length of buffer accesses was 16 cycles. These numbers are realistic, if we consider, for example, that they would represent IDCT acceleration, as described in (Rambaldi *et al.* 1998). Also, the memory transfer length of 16 cycles is realistic, since the 8x8 IDCT operation has 64 samples, which would mean transferring four samples each clock cycle.

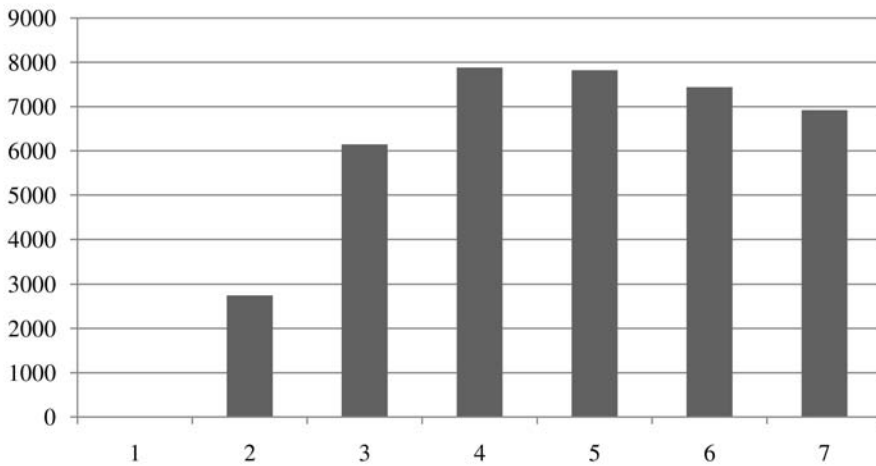
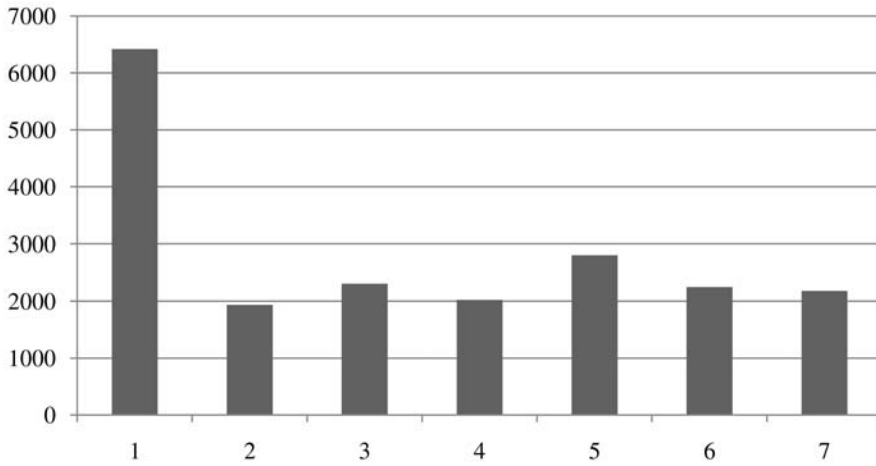
The table column titled *None* assumes that there is no deviation from these latencies of 80 and 16 cycles from task to task. The remaining random variables are the number of tasks per job and the number of jobs. The column named *Small* allows a task length variance of +/- 12.5% for both processing tasks and buffer transactions. Respectively, the column named *Large* allows a length variance of +/- 25%, which sets the range of buffer accesses to [12, 20] and processing to [60, 100].

Finally, the column *Long access* changes the ratio between the average lengths of buffer accesses and processing tasks. In this column, buffer accesses range between [21, 27] and processing tasks between [56, 72]. Notice, that the sum of the average lengths of buffer read (24), processing (64) and buffer write (24) still remains at 112, just as in the three other columns. So, all the results within one table are comparable. Figures 44 and 45 present the results of Tables 7 and 8 in the form of bar graphs.



1. Sequential execution
2. *NoSa* with swinging buffer architecture
3. Parallel 1-port memory architecture, timestamp variables
4. Parallel 2-port memory architecture, timestamp variables
5. Shared 1-port memory architecture, reservation vector
6. Shared 2-port memory architecture, reservation vector
7. Shared 3-port memory architecture, reservation vector

Fig 44. Makespan(top) and overhead(bottom) for 3.5 jobs on average. Units are CPU cycles.



1. Sequential execution
2. *NoSa* with swinging buffer architecture
3. Parallel 1-port memory architecture, timestamp variables
4. Parallel 2-port memory architecture, timestamp variables
5. Shared 1-port memory architecture, reservation vector
6. Shared 2-port memory architecture, reservation vector
7. Shared 3-port memory architecture, reservation vector

Fig 45. Makespan(top) and overhead(bottom) for 14 jobs on average. Units are CPU cycles.

Table 9 shows the memory utilization rates for the four different task lengths. The *NoSa* algorithm was used to schedule 14 jobs on average on a swinging buffer architecture and the number of simultaneous memory accesses was recorded. For example, row 2, column 2 shows that on average for 48% of the schedule makespans there is no memory access activity in the system that has the *Large* task length variance.

Table 9. Memory utilization rates for 14 jobs.

Simultaneous accesses	Large	Small	None	Long access
0	48%	59%	71%	60%
1	24%	14%	4%	5%
2	18%	11%	4%	6%
3	8%	10%	8%	11%
4	2%	6%	13%	18%

7.5.1 *Parallel memory architecture*

The parallel memory architecture was evaluated with three scheduling algorithms. The *regular* algorithm kept track of the granted buffer accesses with a simple bufferwise and portwise timestamp variable. This algorithm was evaluated with one-ported and two-ported buffers. The *diagonal* algorithm also used timestamp variables, but with an alternative task scheduling order that is depicted in Figure 43 on the righthand side. Finally, the quantized reservation vector approach that was originally designed for the shared memory architecture was also fitted to the parallel memory architecture by having a separate buffer for each memory element.

In terms of execution time overhead, no efficient software implementation was found for the *diagonal* scheduling algorithm, and thus the results were left out from the comparison table. The diagonal task scheduling order was found to be slightly beneficial with larger task sets: the makespan of the produced schedules became on average 9% shorter than with the regular task scheduling order when there were 14 jobs to schedule, on average. For small task sets of 3.5 jobs on average, the diagonal task scheduling order provided no advantage over the regular task scheduling order. The diagonal task scheduling order was only evaluated with one-port buffers.

The use of quantized buffer-wise reservation vectors with the parallel memory architecture turned out to be poor solution. The reservation vectors consumed a lot of

memory space ($number_of_memories * number_of_ports * worst_case_makespan$ bits) and the browsing of the vectors at scheduling time created a huge overhead. When compared to the timestamp variable-based algorithm, the reservation vector solution introduced 75% more computation time. With small task sets (3.5 jobs on average), the produced makespan was 7% worse than with the timestamp-based algorithm and 5% better on large task sets of 14 jobs on average. As the solution clearly is uninteresting, the results were left out of the comparison table.

Finally, the best parallel memory scheduling solution was the *regular* algorithm that keeps track of buffer accesses by timestamp variables. However, with one-port buffers the algorithm still created 80% more computational overhead than the same algorithm as a traditional flow-shop variant (Section 6.1.2). The effect of buffer sharing made the makespans 5% to 20% longer than those of the traditional flow-shop solution. The scheduling algorithm for the two-port solution had a 120% larger computational overhead than the traditional flow-shop approach, but produced equally short makespans as traditional flow-shop. For the parallel memory architecture, it is evident that the *regular* heuristic performs best: although the produced makespans are up to 20% longer than with *NoSa* on the swinging buffers solution, the *regular* algorithm has clearly the least overhead of the presented algorithms for parallel memories.

7.5.2 Shared memory architecture

The same experiments that were conducted for parallel memory architectures were also conducted on a single shared k -port memory. The experimental settings were kept the same, so the results are directly comparable. The experiments were mainly conducted for the quantized reservation table based scheduling algorithm, but brief tests were also conducted to see if a single timestamp variable would be enough to keep track of allocated memory access times, as it was done with the parallel memory architecture. Unfortunately, this approach worked very poorly with a shared memory. Thus, the results were left out of the comparison.

The shared memory algorithm that used a quantized reservation table produced moderate results. The reservation table algorithm was tested with a one, two and three-port shared memory. The results show the obvious fact that having only one read/write port in a shared memory decreases the throughput considerably. On the other hand, in most cases three ports do not bring much advantage over two read/write ports, and it is commonly known that having three (or more) port memories is not an

attractive solution in terms of hardware cost. The fact that a three-port memory did not increase the throughput significantly tells us that the scheduled tasks are not very memory-intensive.

The main problem with the quantized reservation vector solution is that the quantized memory accesses and processing task times differ from the real durations, which creates *invisible* gaps to the resulting schedule. The gaps are invisible in the sense that they do not appear in the schedule building phase, as in Figure 41, but exist in reality when the schedule is dispatched. This can be exemplified by considering that we have a PE m that has a static latency of 49 clock cycles and the schedule quantization factor is 16. When the latency of m is quantized with 16, the result has to be rounded up so that it becomes 4. Because of this, the scheduler has to treat m as if it would have a latency of 64. This creates a gap of 15 clock cycles that is invisible in the schedule building phase.

The multiprocessor memory access scheduling of a shared memory is problematic. Access time slots to the few available memory ports have to be shared carefully, which complicates the scheduling algorithm. Furthermore, the approach of quantizing the memory access and PE processing times for lowering the scheduling overhead creates gaps in the resulting schedule.

7.5.3 Experiments with varying numbers of PEs

Finally, experiments were also conducted with varying numbers of processing elements. To make the results comparable with the results presented in Chapter 6, the speedups S were computed for the results. The average operation lengths were also scaled to 500 clock cycles to match Table 6. Thus, the only difference between the results of Table 6 and Table 10 presented here is the set of jobs.

Table 10. The speedup provided by the algorithms on different numbers of machines (M). The average task length is 500 in the results of this table. The number of memory ports is k , and m shows the number of memories in the system. L is the schedule makespan.

Algorithm	Mem. arch.	k	$k*m$	$S_{M=4}$	$S_{M=6}$	$S_{M=10}$	$L_{M=6}$
Sequential	-	1	1	1.00	1.00	1.00	56000
<i>RaNw</i>	Swing.buff.	1	$M(M+1)$	0.00	0.01	0.01	14854
<i>NoSa</i>	Swing.buff.	1	$M(M+1)$	2.01	2.49	3.10	17073
<i>NoSa</i>	Par. mem.	1	$M+1$	1.48	1.75	2.12	20217
<i>NoSa</i>	Par. mem.	2	$2(M+1)$	1.45	1.69	1.98	17834
<i>NoSa</i>	Sh. mem.	1	1	1.26	1.40	1.46	24778
<i>NoSa</i>	Sh. mem.	2	2	1.39	1.60	1.86	20363
<i>NoSa</i>	Sh. mem.	3	3	1.33	1.69	1.97	19788

The results show how the three-part memory model reduces the efficiency of run-time scheduling. All scheduling happening on parallel or shared memories causes so many computations that the speedup provided by parallel execution falls clearly below 2.0, and this happened when the average operation length is 500, which is not really fine-grained anymore. Thus, we have to conclude that the three part task model that has also been used by other authors (Seidel *et al.* 2005, Cortés *et al.* 2005), is too burdening for run-time scheduling. Motivated by this observation, an alternative task model was developed, which is fundamentally different and has a minimal computation time. This approach is described in the next section.

7.6 A simplified memory conscious solution

Based on the observed computational overhead of the three part task model, we now introduce a simplified task model, where the buffer read and buffer write phases are integrated as a part of the task (flow-shop operation). A benefit of this approach is that an operation can have an access right to the read and write buffers simultaneously.

The key motivation for this approach was again to produce run-time generated schedules with minimal computational effort. As was identified previously, the use of separate buffer read and buffer write operations caused 80% of extra effort for the scheduling procedure (See Figure 44). Based on this notion, we formulate a way of including the buffer read and write operations *as a part* of the actual data processing

operation, which unifies read-process-write operations as one.

Again, the efficient *NoSa* scheduling algorithm (See Section 6.1.2) was taken as a basis. First, the algorithm was modified by augmenting every operation in each job with the information *when read buffer access ends* and *when write buffer access starts*. Illustrations of such operations are shown in Figure 46. On the left, the operation has non-overlapping buffer read and buffer write parts, along with a processing part that does not access any buffer. On the right, the buffer accesses actually overlap, just as can happen in real-world computations. It will be assumed that each operation starts with a buffer read access which continues for a fixed time. Similarly, it is assumed that each operation ends with a write buffer access that starts at some fixed point of time.

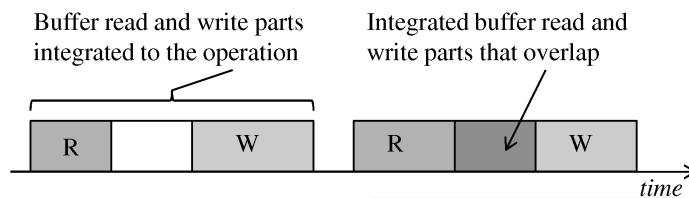


Fig 46. The simplified task model.

7.6.1 Abandoning bypasses

The machine skipping functionality is another source of computational overhead of a scheduler. When parallel memory architecture is used, machine skipping means that a PE reading from buffer i can, in principle, write to any of the buffers after i . Until now this functionality has been implemented with bypass networks.

The same machine skipping functionality can also be implemented by another procedure that we will call *copy-through*. In *copy-through*, the data is not routed by a separate bypass path around a non-wanted PE, but the PE is instructed to copy the data from the read buffer to the write buffer without modifications. This causes the same effect as machine skipping, but eliminates the hardware of bypass networks and simplifies the scheduling algorithm. *Copy-through* makes the system more regular as machine skipping is not allowed and bypass paths are not required around PEs. On the other hand, the PEs become somewhat more complex as they have to implement the *copy-through* mode in addition to their normal functionality.

Combining the simplified task model and machine skipping by *copy-through* produces a considerably simpler memory-conscious scheduling algorithm. When

scheduling an operation, the algorithm now only needs to make sure that the buffer write of the present operation m of job n does not overlap with the buffer read of the operation $m + 1$ of the previous job $n - 1$, just as it is shown in Figure 47. The notations r_i and w_i refer to a read or write to buffer i .

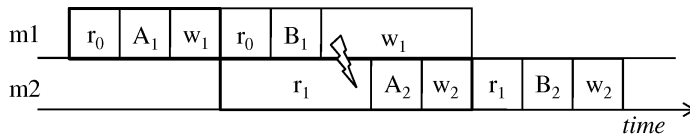


Fig 47. Potentially conflicting buffer accesses with no-wait / no machine skipping

This setting allows simplifying the scheduling problem even further. By abandoning the semi-active timetabling approach and by replacing it with no-wait timetabling, it is possible to go back to the extremely efficient scheduling heuristic introduced in Section 6.1.1, while still using parallel memories. In Section 6.1.1, it was explained how the inter-job distances can be pre-computed at system design time when no-wait timetabling is used. There, memory architectures were not considered and the pre-computed inter-job distances involved only the avoidance of overlapping operations on the same PE.

Now it is possible to add the information on buffer read ending times and buffer write starting times to each operation. At the time of system design, the minimum inter-job distances are again computed. The information about buffer access times is only required at the time of system design and does not need to be included to the run-time system. As a result, the complexity of the run-time system remains *exactly* the same as in the algorithm shown in Section 6.1.1: no additional run-time decision making is necessary.

7.6.2 Experiments

The performance improvement offered by copy-through machine skipping and the simplified task model was evaluated by comparing the improved algorithms to conventional flow-shop *NoSa*, as we have done in the experiments thus far. The improvements were made to both semi-active and no-wait algorithms that had been adapted to parallel memory architectures.

In these experiments, the length of tasks was between 8 and 32 clock cycles, which included a buffer read and a buffer write that each had a length from 8 to 16 clock cycles.

Thus, one task could simultaneously read from one buffer and write to another.

The results can be seen in Tables 11 and 12. The first and last algorithm in both tables are actually identical. When semi-active scheduling is used with 2-port parallel memories without machine skipping, the algorithm actually reduces back to ordinary semi-active timetabling (with swinging buffers), although the practical system implementation is different. Another noteworthy phenomenon is that no-wait timetabling has identical computation times for both 1-port buffers and 2-port buffers. This is caused by the fact that the schedules have been computed partially off-line in both cases, which again leads to identical algorithms.

Table 11. Results for an average of 3.5 jobs. The number of memory ports is k and m shows the number of memories in the system. In this experiment $M=6$.

Algorithm	Mem.	k	$k*m$	Makespan	Comp. time
<i>NoSa</i>	swinging buffers	1	42	193	745
<i>NoNw</i>	parallel memories	1	7	210	614
<i>NoNw</i>	parallel memories	2	14	203	614
<i>NoSa</i>	parallel memories	1	7	203	1122
<i>NoSa</i>	parallel memories	2	14	193	738

Table 12. Results for an average of 14 jobs. The number of memory ports is k and m shows the number of memories in the system. In this experiment $M=6$.

Algorithm	Mem. arch.	k	$k*m$	Makespan	Comp. time
<i>NoSa</i>	swinging buffers	1	42	452	2751
<i>NoNw</i>	parallel memories	1	7	590	2371
<i>NoNw</i>	parallel memories	2	14	553	2367
<i>NoSa</i>	parallel memories	1	7	532	4200
<i>NoSa</i>	parallel memories	2	14	452	2798

The no-wait timetabling approach that allows the use of parallel memories, has even a lower complexity than the *NoSa* approach that does not consider buffers at all. On the other hand, no-wait timetabling suffers again from slightly longer schedule makespans than the semi-active timetabling approach.

7.7 Application specific memory architectures

Up to now we have discussed memory architectures that are highly regular. The shared memory architecture used a single memory as an interconnect between all PEs and the parallel memory architecture placed exactly $M + 1$ buffers around M PEs.

As an example of an application specific solution, we can take a look at the interconnect in Figure 48. The depicted solution consists of five PEs (CPU, COP1, COP2, COP3, COP4) and five parallel memories (1,2,3,4,5). The apparent irregularity of the architecture vanishes if it is re-formulated as an interconnection matrix as depicted in Figure 49.

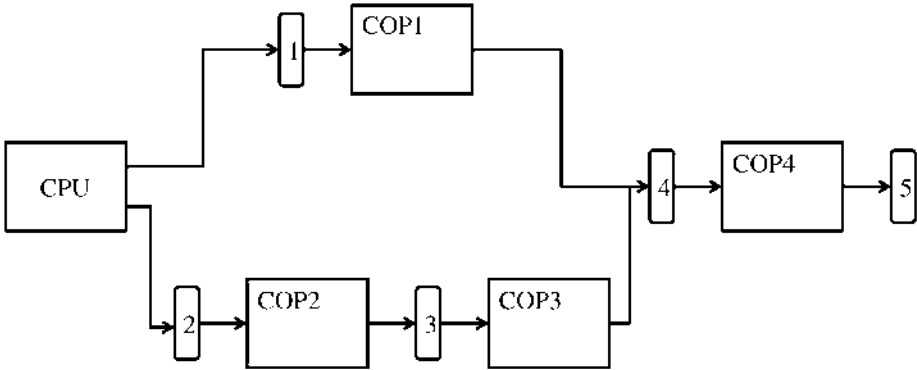


Fig 48. An application specific memory architecture

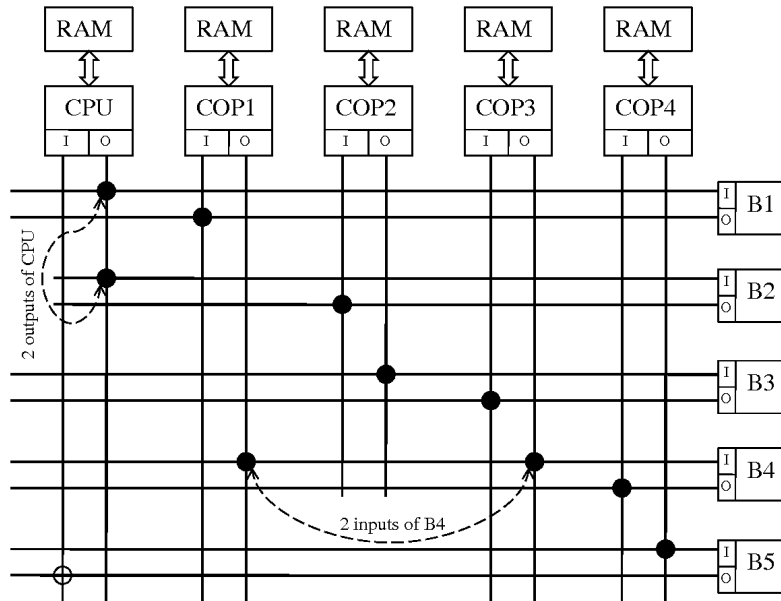


Fig 49. The interconnection network of an application specific pipeline with a parallel section.

The figure reveals that in fact, an application specific interconnect such as that depicted in Figure 48, is actually just a fixed bypass. This observation enables consistent modeling of application specific PE interconnects. Evidently, it is not possible to use the *copy-through* simplification when the datapath has parallel sections.

However, traditional flow-shop scheduling is not well suited to application specific interconnects. If we look at Figure 48, it is evident that for shorter schedule makespans, it is desirable to make COP1 and COP2 start simultaneously and compute in parallel, as Figure 50 illustrates. In the traditional flow-shop model it is not possible to create a job that has two tasks that are active at the same time. Fortunately, this is not a problem for the extended permutation flow-shop model.

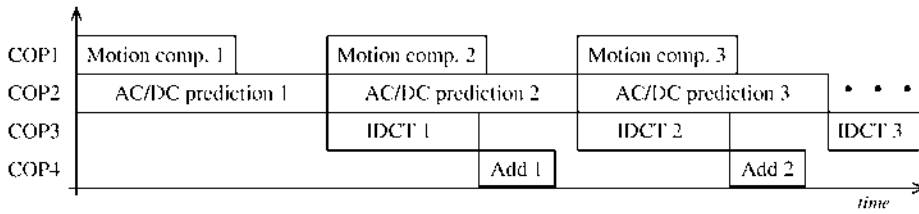


Fig 50. An EPFS schedule with jobs that have simultaneous operations.

7.7.1 EPFS and memory architectures

In Section 4.4, the extended permutation flow-shop (EPFS) model was introduced, and it was shown how the model enables constructing flow-shop like jobs out of static multiprocessor SDF graph schedules. The EPFS model also enables the use of application specific memory architectures. In this subsection, the memory related issues of EPFS are discussed. Since the three part task model presented in Section 7.3 was observed to be inefficient, the memory conscious formulation of EPFS is based on the simplified task model presented in Section 7.6.

Figure 51 shows a general EPFS situation where two jobs are to be executed as close to each other as possible. In traditional flow-shop, PE $m + 1$ would always start *after* PE m , which is not the case in EPFS, where both PEs may run simultaneously. In the EPFS model the buffer write of job 1 (white) operation A might conflict with the buffer read of job 2 (gray) operation B , because those buffer accesses are mapped to the same buffer element when the parallel memory architecture with $M + 1$ buffers (See Figure 42) is used. Generally, when scheduling operation m of job $n + 1$, it is mandatory to check that buffer accesses do not conflict with operations $m - 1$ and $m + 1$ of job n . Machine skipping is allowed in the EPFS model as well, in which case the buffer access arbitration needs to be done carefully, especially for application specific memory architectures: several PEs may have access to one memory element.

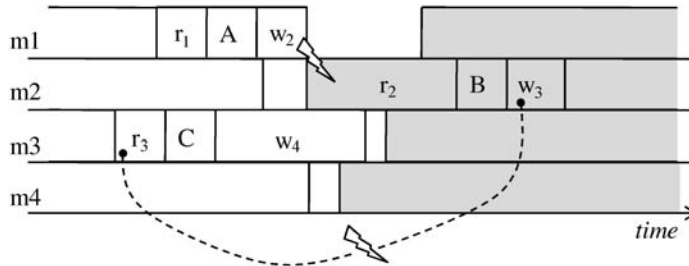


Fig 51. Potentially conflicting buffer accesses in extended permutation flow-shop

The EPFS model offers great flexibility in the range of memory architectures it supports. However, the successful use of the EPFS model in practical scheduling relies heavily on the design-time software that computes the quasi-static schedules needed by EPFS. The EPFS run-time system just considers the design-time determined job offsets and task offsets, and schedules the jobs according to those.

Since the EPFS model is an extension of the *NoNw* algorithm (Section 6.1.1), the computational overhead of implementing run-time EPFS scheduling is identical to that of no-wait flow-shop scheduling. In this section, we shall not present separate experiments for memory conscious EPFS scheduling, or application specific memory architectures. This is due to the fact that an EPFS compatible scheduler LSI circuit has been constructed and will be experimented in Section 8.4.2 in a real multi-PE system.

7.8 Summary

We have presented numerous scheduling solutions in this chapter, and they are summarized in Table 13. The solutions are grouped according to the memory model that they support. For all the presented memory architectures, except for the shared memory, an efficient scheduling solution has been found. By efficient, we mean that the scheduling overhead is not greater than that of the *NoSa* for swinging buffers that has been used as a reference.

Table 13. The presented scheduling algorithms.

Algorithm	Memory model	Machine skipping	Section
<i>NoNw</i>	Swinging buffers	Yes	6.1.1
<i>NoSa</i>	Swinging buffers	Yes	6.1.2
<i>PaNw</i>	Swinging buffers	Yes	6.1.3
<i>PaSa</i>	Swinging buffers	Yes	6.1.3
<i>RaNw</i>	Swinging buffers	Yes	6.1.4
<i>NoSa</i>	Parallel memories	Yes	7.4
<i>NoSa</i>	Parallel memories	copy-through	7.6
<i>NoNw</i>	Parallel memories	copy-through	7.6
<i>NoSa</i> , Q. res. table	Shared memory	Yes	7.3
EPFS	App. specific	Yes	7.7.1

Selection of the scheduling heuristic and memory architecture affects both the hardware cost and the performance cost of the system. Hardware cost means physically more silicon surface, whereas performance cost refers to longer scheduling times or makespans that decrease the throughput. The performance cost of different algorithms on a single-core instruction processor has been presented in Chapter 6 and this chapter. Tiwari *et al.* (1996) state that the energy usage and running times of programs on general purpose processors track each other closely, which evidently can also be applied here, since the NIOS II is a simple RISC processor.

Measurements on the hardware cost of different memory architectures have not been performed and it is not possible to deduce anything about the complexity or latency of LSI circuit implementations of different scheduling algorithms, based on the presented results. Furthermore, the maximum attainable clock frequency of different memory architectures (due to number of ports) has not been discussed. An extensive study of these effects is a direction for future research.

Selecting a single scheduling algorithm out of the ones presented here is not straightforward. There are three questions that have to be answered: 1) Which memory architectures are available for the multiprocessing system? It is possible that in some embedded systems the designer is forced to work with a single shared memory. 2) Which is more important: short schedule makespans or small buffer sizes? 3) How long are the average task execution times?

Answering question 1 may rule out a great number of alternatives already, since

for application specific and shared memory architectures we have only presented one solution. Question 2 essentially makes the selection between no-wait timetabling and semi-active timetabling. Semi-active timetabling provides much better makespans than no-wait timetabling, but may require considerable worst-case buffer sizes. Finally, the task execution times also make a small difference. For tasks longer than 700 clock cycles, it is worthwhile doing the job ordering of Palmer (1965) together with semi-active timetabling.

In the next chapter, we shall take a look at task dispatching, which is an extremely important issue when fine-grained PEs are used in an MPSoC system. After dispatching, a no-wait flow-shop and EPFS compatible LSI circuit is presented.

8 Hardware support for scheduling and dispatching

8.1 Dispatching applications on multiprocessors

Multiprocessor execution of an application requires a system that takes care of dispatching tasks according to a generated schedule. In conventional multiprocessor systems, dispatching of tasks is not a critical issue since the task latencies are so long that the dispatching accuracy and overhead is negligible. For fine-grained multiprocessing, also the dispatching needs to be done carefully to ensure an acceptable overhead, and correct system behaviour.

For analyzing the performance that a software dispatcher can provide, a software implementation of a multiprocessor dispatcher was implemented on an Altera Cyclone III FPGA. The architecture of the system is shown in Figure 52. The CPU contains a parallel I/O (PIO) port that is connected to the dispatch signal lines leading to each of the PEs that are orchestrated by the CPU. A pulse in the dispatch signal line causes the PE to start computing immediately. Irrelevant details have been omitted from the figure.

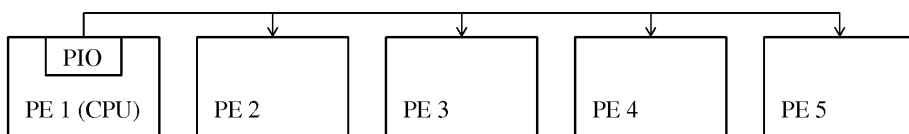


Fig 52. Dispatching slave PEs with CPU software.

From the CPU software point of view, the PIO port is seen as a memory mapped device that can be written to with a single command. Each signal line is connected to a separate bit in the PIO port, which makes it possible to dispatch all four slave PEs during one clock cycle.

The dispatching software was written as if the CPU would be dedicated to dispatching slave PEs. The tasks to be dispatched were written in a vector of 32-bit integers. Each of the cells in the vector contained the dispatching time in the upper 24 bits, and the PEs to be dispatched in the lower 8 bits. The dispatching loop is shown in Figure 53.

```

data = dispatchlist[0];
start = data >> 8;
procs = data & 0xFF;

for(timec = 0; timec < TIME_END; timec++)
{
    // start time interval measurement here
    if(start == timec)
    {
        IOWR(PIO_BASE, 0, procs);
        IOWR(PIO_BASE, 0, 0);

        data = dispatchlist[++index];
        start = data >> 8;
        procs = data & 0xFF;
    }
    // end time interval measurement here
}

```

Fig 53. The pseudocode of the dispatching algorithm.

In the general case, the dedicated dispatcher loops around the *if* statement and checks constantly if the timestamp of the next dispatching moment has been achieved yet. At some point, the *if* statement becomes true. Then, as a first task, the parallel output pins are activated, and immediately reset to zero. The *procs*-variable has been loaded previously and it includes values of '1' for those bits that represent a PE to be started. When the dispatching has been completed, the dispatch vector index is incremented and a new vector value is fetched from the memory. The fetched data word is then stored into separate variables that hold the timestamp of the next operation and the processors to be started.

During the experiment, the program and the data resided on a memory that was located on the FPGA. The CPU was a Nios II/e, which is a minimal RISC processor without caches or branch prediction. The compiler was directed to produce maximally optimized code (-O3 setting). More advanced kinds of the NIOS II CPU (the */s* and */f* variants) and different compilation options were tried, but the previously listed configuration produced the best performance. The measured time interval took 13 clock cycles when the *if* statement was false, and respectively 23 clock cycles when the statement was true.

This dispatching algorithm has been designed with minimum dispatching overhead in mind. The difficulty with this implementation lies in the mapping of scheduled tasks to the dispatching data vector. The presented organization implies that the tasks to be dispatched are sorted in groups according to their starting time. The approach is similar to the first phase of bucket sort (Corwin & Logar 2004) or computation of a histogram (Shapiro & Stockman 2001): a set of bins is established and each bin is responsible for a time interval. The bin width is chosen based on the worst-case dispatching time, such as 23 in the example above (three first bins would be 0-22, 23-45, 46-68). Then, all the tasks residing in the same bin would be designated to start simultaneously.

This experiment has presented a system that is heavily optimized for dispatching accuracy, neglecting the overall system performance. Despite this emphasis on optimization, the dispatching jitter is still considerable when fine-grained tasks are considered. This leads to a decreased throughput with fine-grained tasks that have latencies shorter than 100 clock cycles. From these observations, we have developed a dedicated hardware dispatcher that cycle accurately dispatches tasks without introducing computational overhead.

8.2 LSI circuit for scheduling and dispatching

Software based scheduling and dispatching becomes problematic when the task latencies are less than 100 clock cycles: the overhead of run-time scheduling nullifies the advantage provided by parallel processing (Section 6.2), and a software dispatcher provides mediocre accuracy requiring a dedicated processor.

In this section, we will describe a hardware unit that is connected to the CPU and relieves the CPU from the cumbersome tasks of scheduling and dispatching. For brevity, we will name the unit CMU as an abbreviation for Co-processor Management Unit. The main ideas of this chapter have also been described in Boutellier *et al.* (2009b).

Designing dedicated scheduling hardware involves a tradeoff between the introduced silicon area overhead and the circuit speed. A low latency scheduling co-processor will generally have a larger area overhead than a slower implementation. In the CMU solution that is presented in this chapter, we decided to optimize the design for speed. Designing an energy efficient LSI circuit for scheduling is definitely an issue for future work. At this stage, it was more important to make sure that the long scheduler execution times could be mitigated by a hardware solution, because the software scheduler running times were still too long for really fine-grained tasks.

In the CMU, the possible job types that can be scheduled have been fixed and embedded into the CMU internal memories at the time of system design. Thus, the CPU can command the CMU to schedule a job for execution by providing the identifier of that job to the CMU. If the job types are unknown at the time of system design, the CPU should transmit the starting and ending times of jobs to the CMU at the point of scheduling.

The CMU takes three clock cycles to schedule a job, is ready to dispatch the scheduled jobs without any delay, and dispatches the scheduled operations without any support from the CPU. The CMU is capable of doing PFS scheduling with no-wait timetabling (with swinging, parallel or application specific buffer architectures) and supports the EPFS model. The implemented version of the CMU is based on the look-up table optimized no-wait timetabling algorithm that was described in Section 6.1.1. This solution minimizes the latency of the job scheduling procedure, but limits the number of job types that the CMU can support.

8.2.1 Related work

Proposing scheduling and dispatching in hardware is not a novel idea. A fair amount of research has been carried out in designing hardware schedulers. Martins *et al.* (2005) divide the prior work to the categories of task scheduling and network traffic scheduling. We will only consider the prior work related to task scheduling here, because network traffic schedulers do not consider dependencies between tasks (Martins *et al.* 2005), which makes the problem quite different.

A well-known, but somewhat aged solution in hardware-assisted scheduling was developed at the University of Mälardalen under the names FASTHARD, FASTCHART and RTU (Real Time Unit). The two first mentioned projects were prototypes that eventually lead to the real-time service providing co-processor RTU (Stärner *et al.* 1996). RTU supports static and priority based scheduling. Around the same time, the Spring project produced the SSCoP scheduling co-processor (Burlison *et al.* 1999) that supports online as well as static scheduling with different policies. It assumes that tasks are non-preemptable and have precedence constraints. The tasks scheduled by SSCoP are guaranteed to meet their deadlines.

Both the RTU and SSCoP are related to our work in the sense that they target multiprocessor systems, but they are more complex and slower than our design. More recently, three research works have been published that target uniprocessor systems.

The co-processor designed by Hildebrandt *et al.* (1999), supports one pre-emptive on-line scheduling algorithm. A similar solution has been designed by Sáez *et al.* (1999), which describes a hardware implementation of a uniprocessor slack stealing algorithm. Kuacharoen *et al.* (2003) recently published a hardware scheduler design that implements three uniprocessor on-line scheduling policies which can be changed even at run-time.

Another branch of research related to the CMU is the work of Jung *et al.* (2002). They describe an approach for synthesizing a hardware controller for systems synthesized from SDF graphs. The purpose of the CMU is similar, except that the CMU is designed for piecewise deterministic systems, whereas the work of Jung *et al.* (2002) does not support quasi-static scheduling. In the solutions that we have presented, a single unit (CPU or CMU) orchestrates the functionality of other PEs in the system. Jung *et al.* (2002) name this approach as the *centralized controller* approach. According to Jung *et al.* (2002) the alternative to this methodology is a distributed controller, where the synchronization responsibility has been shared among separate units that are attached to each PE. Jung *et al.* (2002) state that their own system falls into the category of centralized control, which was their design choice because they claim that the distributed controllers have a larger area overhead.

Kumar *et al.* (2007) present architectural scheduling support, named *Carbon*, for chip multiprocessor (CMP) systems. Their work is similar to ours in the way that they have also observed that the overhead of the scheduling effort restricts the granularity of parallel tasks that can be scheduled. They present an extension to the CMP system by adding a set of hardware task queues and extensions to the instruction set architecture. Their experiments are conducted with task lengths between 329 to 13876 clock cycles, which is on the upper limit for presenting fine-grained tasks. Their scheduling approach is fully dynamic load balancing. The authors have evaluated their approach on a simulated CMP system where it is assumed that the added instructions for interfacing the scheduling hardware take five clock cycles. This is of the same magnitude as the latency of our system, with the exception that our scheduling hardware schedules a *set* of interdependent tasks (a job) in three clock cycles.

Al-Kadi & Terechko (2009) proposed a hardware task scheduler for fine-grained tasks. Their scheduler takes 15 cycles for scheduling and synchronization of tasks. Their solution is designed for homogeneous multiprocessing systems, and it also selects at run-time, to which processor the tasks are mapped. The solution by Al-Kadi & Terechko (2009) has several features more than the CMU, but on the other hand, is also far more

complex. The authors show results that slightly outperform the solution of Kumar *et al.* (2007).

Seidel *et al.* (2005) have proposed a computing engine for multi-standard video applications, and their approach has many features in common with our work. In their system, they use several Batch Control Units (BCU) that start different tasks on digital signal processors according to a schedule, and also send instructions to direct memory access controllers. In the categorization of Jung *et al.* (2002) the BCU solution falls into the category of distributed controllers. In the same fashion as our in our work, Seidel *et al.* (2005) assume that the execution times of DSPs are fixed (or worst-case) and pre-emption is not allowed. Furthermore, the authors say that the scheduling model used is *job shop*, which is a generalization of the flow-shop model that we have used. The work of Seidel *et al.* (2005) does not show if their system can produce dynamically changing (*i.e.* quasi-static) schedules. The example application that the authors use is moving JPEG encoding, which is implementable with a static schedule. Seidel *et al.* (2005) model data transfers to and from PEs as separate DMA tasks. This makes their approach similar to the three part task model that we have shown in Section 7.3, as well as the bus modeling used by Cortés *et al.* (2005).

Table 14 shows a summary of some of the designs mentioned here. Many of the designs (including ours) are in fact scalable and the shown values represent only one configuration. Moreover, the specified sizes of the designs are all given in different units. Since there is no reliable way of converting these units, the values are left as given in the references. The size of our scheduler is specified in *logical cells*, which is a measure used by Altera's design tool Quartus II. The size of the RAM is announced separately and here the example size is 512 bytes, which provides a maximum of 32 tasks for each of the four processors.

Table 14. Comparison of scheduling co-processors. (*) refers to an arbitrary number. The LCs in the proposed solution refer to Logic Cells, a measure used for the FPGAs of Altera (Altera 2008). (Boutellier *et al.* 2009b, modified by author with permission of IEEE.)

Implementation	Policy	PEs	Tasks	Announced size
Hildebrandt <i>et al.</i> (1999)	ELLF	1	32	3.2k conf. log. blocks
Sáez <i>et al.</i> (1999)	Slack steal.	1	n/a	n/a
Kuacharoen <i>et al.</i> (2003)	Many	1	16	1115 st. cells
Burleson <i>et al.</i> (1999)	Many	32	32	84372 transistors
Stärner <i>et al.</i> (1996)	Many	3	64	25000 gates
Al-Kadi & Terechko (2009)	Load bal.	4	N/A	0.048mm ² at 45nm
Jung <i>et al.</i> (2002)	Static	5	N/A	12 FPGA cells
Kumar <i>et al.</i> (2007)	Task steal.	<65	(*)	N/A
Seidel <i>et al.</i> (2005)	Static	(*)	N/A	N/A
Proposed	(E)PFS	4	128	485 LCs + $\frac{1}{2}$ kB RAM

The greatest difference between the related work and our own, is that only few (Seidel *et al.* 2005, Jung *et al.* 2002, Burleson *et al.* 1999) hardware schedulers support heterogeneous processing. Of these three solutions that consider heterogeneous PEs, the approaches by (Seidel *et al.* 2005) and (Jung *et al.* 2002) only support compile time generated schedules. Finally, the solution by Burleson *et al.* (1999) is designed for challenging hard real-time problems and is considerably slower than the CMU proposed by us.

8.3 Implementation environment

For implementing the CMU, the Altera Cyclone III FPGA was selected as a testbed. A set of Altera NIOS II/s soft processors (Altera 2008) were synthesized on the FPGA along with the CMU. One of the NIOS II processors acted as the main CPU while the others worked as slave PEs under the control of the CMU. The interface between the CPU and CMU was implemented with Altera’s Avalon bus. The bus delays have a negligible impact on the overall performance, because the CPU only communicates

a few bytes to the CMU when jobs are to be scheduled by the CMU. The CMU was connected to the slave PEs with nine signal lines, eight of which are used to send data and one provides a wake-up signal.

8.3.1 Detailed CMU functionality

To reduce the amount of communication between the CPU and the CMU, as well as speed up the computation of schedules, a fair amount of data has been stored in read-only look-up tables inside the CMU. One of these LUTs contains all jobs that the system may want to have scheduled, and the other is an inter-job distance table (see Section 6.1.1) that contains the time offsets between all job types. The distance table provides a great reduction in the co-processor latency but, unfortunately, sometimes provides a sub-optimal offset when machine skipping is allowed. The size of this inter-job distance LUT is quadratic in the number of known job types; however each LUT entry only contains a single integer value. In this FPGA implementation, the LUTs are fixed. However, there would be no obstacle in implementing the LUTs with RAM memories, which would enable changing the supported job types at runtime.

Figure 54 shows a block diagram of the CMU's scheduling functionality. The CMU has been implemented to support four slave PEs in this example. The data words coming from the CPU contain the job type (expressed as an identification number) to be scheduled, and eight bits of application specific data. The routing of the application specific data is not shown in Figure 54 for simplicity. The arrows originating from the interface block represent the routing of the job identification number. At the top of the figure, the job *id* is spread over four LUTs that contain the operation wise data for each job *id*. Below, the register named *PJ* is used to store temporarily a job index, so that the CMU is always aware of the present job *id*, as well as the previous one. By using the previous and current job *ids*, the *Job distance LUT* is accessed. This LUT provides the inter-job offset that has been determined at the time of design for all combinations of two consecutive jobs.

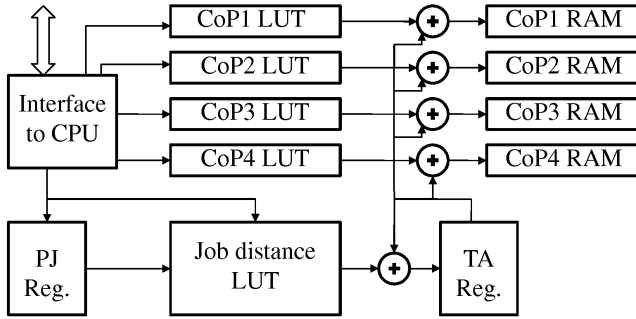


Fig 54. The scheduler part of our CMU. *PJ Reg.* contains the previous job index, *TA Reg.* contains the accumulated job time, *CoPx LUT* contains operation specific offsets. (Boutellier *et al.* 2009b, published by permission of IEEE.)

The *Job distance LUT* outputs the number of clock cycles that express how many cycles the present job has to be offset from the previous one. This number is added to the cycle offset of each operation in four separate adders. Finally, the offset clock cycles are accumulated and stored into a register named *TA*. The scheduling results are stored PE-wise in separate data vectors named *CoPx RAM*.

The dispatcher part of the CMU is shown in Figure 55. Once the CPU has sent all the jobs to the co-processor, the *CoPx RAMs* contain a list of operations that have to be executed on each PE, sorted in order of ascending starting times. Once the CPU instructs the CMU to start dispatching, a time counter is started in the dispatching unit, and each processor specific comparator register is loaded with the first value of the respective *CoPx RAM*. The time counter increments synchronously with the system clock and during each clock cycle the comparators compare the current operation timestamp to the time counter value, in parallel. When a match occurs in a comparator, the output lines of the CMU are activated and the application specific data is transmitted to the slave PE. The parallelism of the dispatch unit allows several operations to be started simultaneously.

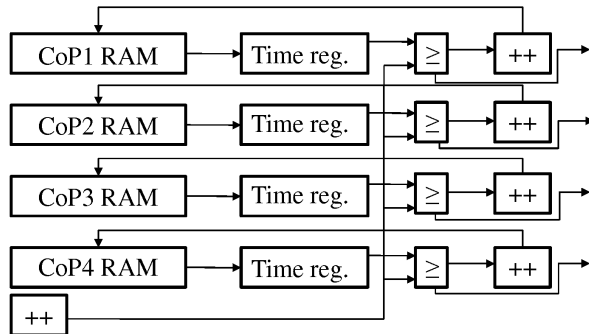


Fig 55. The dispatcher part of our CMU. *CoPx RAM* contains the scheduled operations that await dispatching. \geq is a comparator and ++ signifies a counter. (Boutellier *et al.* 2009b, published by permission of IEEE.)

The CMU also supports the machine skipping functionality of flow-shop scheduling. To allow this, a special *empty operation* has been implemented, which does not cause an activation of the slave PE. Due to the dispatcher unit design, these *empty operations* need to have a minimum length of three clock cycles, which is the read latency of the *CoPx RAM*. Practically, this means that in some cases two consequent operations are scheduled three clock cycles apart instead of in immediate succession.

The CMU design is scalable in the number of processors it supports, as well as the number of jobs. The results in Table 14 and in the following subsections apply only to one specific implementation that supports four processors, 13 job types and up to 32 scheduled jobs (which equals 128 operations/tasks). With these parameters, the CMU was synthesized, placed and routed with a UMC 180nm CMOS process and Faraday standard cell library, which resulted in 3300 gates, ignoring the 512 byte *CoPx RAM*. The maximum operating frequency of the LSI circuit is 526 MHz.

8.4 System level experiments

We performed two experiments with the CMU. In the first experiment, the CMU was inserted into an embedded system consisting of a CPU and three slave PEs. The CPU was used to perform MPEG-4 SP video decoding assisted by the slave PEs that were dedicated to perform certain demanding video decoding functions. The CMU was controlled by the CPU, and orchestrated the functionality of the slave PEs. The purpose of this experiment was to show that the CMU is flexible enough to be used with real

applications. Furthermore, the throughput of quasi-static scheduling offered by the CMU is compared to the throughput of a pessimistic static schedule, similar to the work of Cortés *et al.* (2005).

In the second experiment, the CMU was used in a more theoretical setting to demonstrate the performance issues related to the use of parallel, fine-grained PEs. The CMU was connected to a CPU and four slave PEs that emulated fine-grained MPEG-4 SP hardware accelerators. Although the system did not process real data, it demonstrates the benefit of run-time scheduling of fine-grained hardware accelerators over static scheduling.

8.4.1 Parallel MPEG-4 decoding

Our first experiment that involved the CMU was realized on an Altera Cyclone III FPGA. Four NIOS II soft processors were instantiated on the FPGA in a configuration that is depicted in Figure 56. One of the Nios II processors acted as the CPU and the other ones as dedicated slave processors. The processors were organized as a pipeline interconnected by 1kB on-chip buffer memories (numbers 1, 2, 3 in Figure 56). The last PE in the pipeline wrote the results directly to the frame buffer that was located on an external SRAM. The first slave processor (labeled COP1) was responsible for dequantization of image data, whereas the second processor was responsible for 8x8 pixel IDCT (COP2). The last co-processor (COP3) was used for adding together predicted image data and texture decoding data, followed by color value clipping. COP1, COP3 and the CPU were NIOS II/s soft processors with a 512 byte instruction cache. The IDCT processor was a more powerful NIOS II/f processor with 2kB of data cache and 2kB of instruction cache. The IDCT processor was given more computational performance because the IDCT operation had the longest execution time and was the bottleneck in the pipeline. The mapping of tasks to PEs in this experiment is an example; an alternative form of task mapping can be seen in the work of Schumacher *et al.* (2005).

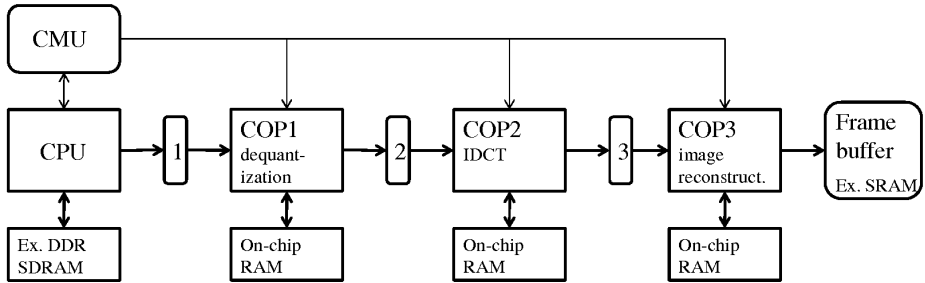


Fig 56. The MPEG-4 SP decoding system. (Boutellier *et al.* 2009b, modified by author with permission of IEEE.)

The CMU was connected to each of the slave processors through a dedicated signal line that provided a wake-up signal and application specific data whenever a slave processor was required to start processing. At system design time, quasi-static schedules were computed for the system by the 2-port parallel memory *NoNw* model that was described in Subsection 7.6.1.

Because the dedicated slave processors were performing the computations by running software code, the task latencies were rather long. The processor performing dequantization took 2290 or 2450 clock cycles for processing 64 pixels, the IDCT processor took 4510 cycles and the clipping processor 1470 or 2070 cycles. Dequantization and clipping were mode dependent: intra and inter-blocks require different numbers of cycles.

The pipeline buffers (1, 2 and 3) were initially designed as single-port memories. However, when the off-line schedule was generated, it was obvious that single-port memories would provide a reduced throughput. The design of the dequantization, IDCT and clipping algorithms was such that they needed access to read and write buffers throughout their whole execution time. Due to the constantly reserved buffers, the job overlap (pipelining effect) declined. This is depicted in Figure 57. The gray edges in an operation represent access to pipeline buffer 2 and the black edge represents access to pipeline buffer 3. To improve the throughput, the buffers were modified to have two ports that allowed simultaneous reading and writing of data.

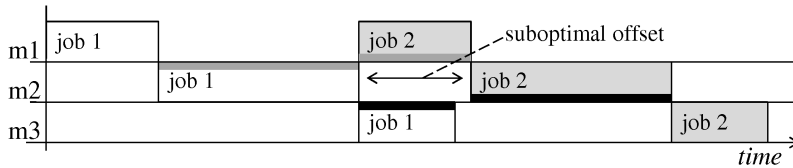


Fig 57. Suboptimal offset caused by shared memory accesses.

The resulting system was used to decode MPEG-4 SP video. A short clip of 45 frames of the well-known *foreman* video sequence and a low 176x144 resolution was chosen for decoding. We measured the number of CPU clock cycles that was required to perform the computation of the *parallelized decoder part*. This part encompassed the dequantization, IDCT and image reconstruction actions. For a comparison, it was determined how many clock cycles the same part would take with a fully static run-time schedule; the results are shown in Table 15. The run-time scheduled decoding of this part takes 67% less time (in CPU clock cycles) to complete the same task, including the communication between the CPU software and the CMU. This can be explained by the fact that in MPEG-4 macroblocks, the number of blocks that contain texture (residual) data can be anything between zero and six. Fully static scheduling naturally requires assuming the worst-case situation of all blocks containing residual data, which is only rarely the case in reality. Run-time scheduling can adapt to the changing number of blocks to decode.

Table 15. Duration of statically and dynamically scheduled MPEG-4 Simple Profile decoding efforts. (Boutellier *et al.* 2009b, published by permission of IEEE.)

Experiment 1, fully static schedule	141 Mcycles
Experiment 1, quasi-static schedule	47 Mcycles
Experiment 2, fully static schedule	1.13 Mcycles
Experiment 2, quasi-static schedule	0.78 Mcycles

The average utilization of PEs during the parallel program parts was 43%, which is not very good. Balancing the pipeline by making the tasks equally long, would not help too much either, since that would make the utilization 49%, which is not much better. The best way to improve the utilization would be to make sure that the pipeline is constantly in use. This would require three changes: 1) the sequential program part should run in parallel with the tasks that are scheduled in these experiments. 2) The latency of the sequential part should be maximally as long as that of the parallel part. 3) The CMU

should be re-designed so that it would continuously schedule and dispatch jobs.

Although this experiment is rather simplistic, it clearly shows that the CMU can be used in practical applications and that it can improve the system throughput. The PEs that were scheduled by the CMU are not fine-grained as their execution time is around a few thousand clock cycles. In the experiment that is described in the next subsection, the CMU is used in conjunction with PEs that truly have fine-grained execution times.

8.4.2 *Orchestration of fine-grained PEs*

In our second experiment, the CMU was placed on an Altera Cyclone III FPGA along with five NIOS II/s soft processors. One of the soft processors acted as a CPU, and the four other processors worked as slave units, emulating fine-grained accelerators. The interconnection of the devices is depicted in Figure 58. The CMU was connected to the CPU through a dedicated I/O port. A signal line was connected from the CMU to each of the co-processors.

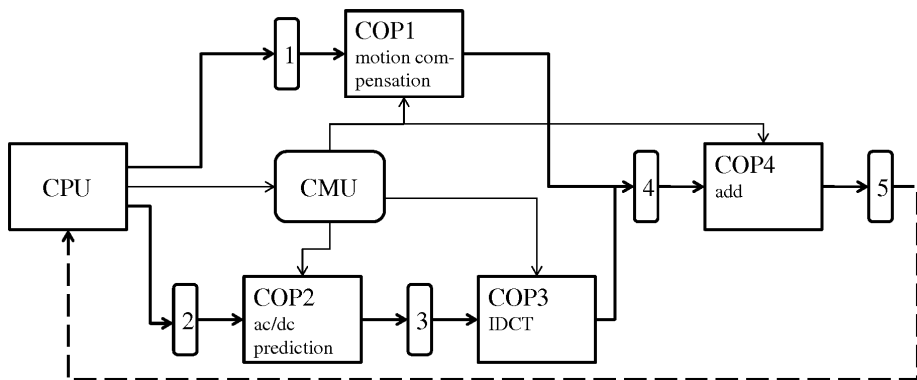


Fig 58. The system with fine-grained PEs. The CPU and COPs have dedicated RAMs that are not drawn. (Boutellier *et al.* 2009b, modified by author with permission of IEEE.)

To show the potential of quasi-static scheduling of fine-grained hardware accelerators, the system was adapted to model MPEG-4 video decoding. Except for COP4, the other slave PEs mimicked the functionality of real-world fine-grained hardware accelerators. COP1 was responsible for motion compensation that takes 85 clock cycles in the TTA implementation realized by Reinikka (2009). COP2 mimicked the AC/DC prediction

accelerator designed by Lin *et al.* (2004) that takes 136 clock cycles to complete. Finally, the IDCT operation was modeled according to the accelerator designed by Rambaldi *et al.* (1998) that takes 80 clock cycles to complete. The parallel motion compensation and texture decoding (AC/DC prediction and IDCT) streams are combined by COP4 that simply adds the respective results.

In our experiment, the four hardware accelerators were simulated by the NIOS II/s soft processors. Since the computation speed of the soft processors was not enough for computing the real operations at the desired latencies, the soft processors were programmed to move around data according to the commands of the CMU, without modification.

When the CMU wake-up signal starts one of the slave COPs 1, 2 or 3, they just read a data value from the input buffer, wait in an idle loop and write the data to the output buffer after a fixed time. The time from reading the input buffer to finishing the buffer write takes 85, 136 and 80 clock cycles, respectively.

COP4 was required to perform some computations. When the CMU initializes COP4, it reads the input data from the input buffer, and depending on the required decoding mode, either outputs the data produced by COP1, or the data produced by COP3, or outputs a combined value by adding the values together. Since these different operation modes take a different number of clock cycles to finish, COP4 has a latency of 32 clock cycles for outputting the combined result, and 24 cycles otherwise. The quasi-static schedules for this system were computed by the EPFS model that was described in Section 7.7.

The slave PEs were invoked in a pattern that was taken from real MPEG-4 SP video decoding. The software XViD video codec (Xvid Solutions 2008) was modified (Boutellier *et al.* 2007c) so that the usage of the functions modeled by COP1, COP2 and COP3 was recorded in a data file. This recorded data file was used in the FPGA system to provide a processing element invocation pattern.

The signals coming from the CMU were observed by Altera's SignalTap logic analyzer. The observations confirmed that the signals triggered by the CMU are clock-cycle accurate. On a larger scale, the reliability of the CMU was verified by computing checksums of the data that had been orchestrated through the datapath shown in Figure 58. The experiments confirmed that the CMU is suitable for orchestrating the functionality of fine-grained PEs.

The improvement in throughput, compared to a fully static schedule, shown in Table 15 is smaller than in Experiment 1. The reason behind this is that motion

compensation is performed virtually for every block in the stream. Thus, the schedule makespans do not vary as dramatically as in Experiment 1.

8.5 Discussion

In Section 4.4 it was stated that static SDF graph schedules can be modeled as EPFS jobs. Generally this implies that the PEs may communicate with each other at run-time after an EPFS job has been dispatched. For this to succeed, it means that the communication must happen either *a)* completely synchronously between the PEs or *b)* the PEs must be mutually responsible for the success of this communication, as there is no external mechanism to direct the interprocessor communication.

The role of the EPFS model and the CMU in this scheme, is merely to make sure that each PE contributing to the EPFS job is dispatched *cycle-accurately* with regard to the other PEs. With the experiment of Section 8.4.2, we have verified that the cycle-accurate dispatching of PEs works in practice. Thus, the responsibility for inter-processor communication between PEs is a matter of the PE designers.

In principle, the CMU could be modified to support different classes of applications with different scheduling requirements. For applications that continuously provide data to be processed, the CMU should be modified to continuously schedule and dispatch jobs, without a separate dispatching command.

Second, the job data LUTs could be expunged from the CMU to allow it to process jobs that cannot be determined at the time of design. This would remove look-up tables from the CMU, but would also introduce more arithmetic functionality. Thus, the critical path of the CMU would become longer, which would reduce the maximum achievable clock frequency. On the other hand, for slower RISC processors the present version of the CMU is unnecessarily fast.

Finally, the CMU could be adapted to work more efficiently with variable-latency tasks. Theoretically this would mean departing from the flow-shop and EPFS models, and this is beyond the scope of this thesis.

9 Conclusion

Designing energy-efficient hardware for applications such as MPEG-4 video decoding or MIMO-OFDM baseband processing is challenging because these applications require high throughput, as well as flexibility to avoid pessimistic allocation of processing resources. Hardwired accelerator circuits provide the best energy-efficiency, but are inflexible by nature. Furthermore, designing an application specific circuit is expensive and takes a great deal of time. A solution that is both energy-efficient and flexible, is to design fine-grained application specific processing elements. Fine-grained processing elements can be highly optimized circuits with a flexible interface that allows them to be used independent of other processing elements in the system.

Fine-grained application specific processing elements can be designed to implement general purpose functions that can be used in several applications and their small size makes the design and verification times reasonable. Prior to this work, it was unclear whether it is possible to dynamically use a set of heterogeneous low latency processing elements without introducing a tremendous orchestration and synchronization overhead. The units cannot be scheduled statically, since this would lead to a pessimistic schedule and decreased efficiency.

Fully dynamic run-time scheduling is not feasible either, since run-time scheduling is a highly power consuming task. Since the presented applications of MPEG-4 decoding and MIMO-OFDM baseband processing consist of short piecewise deterministic parts, the processing element invocation schedules can also be constructed piecewise. In literature this approach is called quasi-static scheduling.

Quasi-static scheduling captures the dynamic behaviour needed by the application but does not waste resources on unnecessary flexibility. Flow-shop provides a scheduling approach that is compatible with the requirements of quasi-static multiprocessor applications. In this thesis, several flow-shop scheduling heuristics have been proposed and compared to each other.

As the flow-shop model cannot support parts of schedules that have more than one processing element active at a time, an improved version of flow-shop scheduling was presented in this thesis. This model was called extended permutation flow-shop scheduling and it can be used to model parts of applications that use several processing elements simultaneously.

Moreover, both the extended permutation flow-shop model and the traditional flow-shop model have been extended to support various memory architectures. The covered memory architectures include the use of shared memory and application specific datapaths.

The experiments based on software schedules showed that a general purpose RISC processor is not efficient enough for computing schedules for fine-grained processing elements. Similarly, another experiment showed that the dispatching of fine-grained tasks also poses problems for a general purpose processor.

Motivated by these observations, a dedicated integrated circuit was designed for scheduling and dispatching of fine-grained PEs. The circuit supports both flow-shop scheduling and extended permutation flow-shop scheduling and has an area overhead of 3300 gate equivalents (excluding a 512B RAM that is also required) when it supports four processing elements and 13 job types. The circuit can schedule a quasi-static part of an application in three clock cycles and dispatch the schedule independently. The scheduling algorithm also supports parallel and application specific memory architectures.

With the help of this circuit, the run-time scheduling of fine-grained PEs can be implemented with negligible scheduling and dispatching overhead. This was shown through an experiment where four PEs implemented tasks that had latencies between 24 and 136 clock cycles. In another experiment, we connected the scheduler/dispatcher circuit to another multiprocessing system that decoded MPEG-4 video. With the experimental setup, the throughput of the parallelized application part improved 67% when compared to a static schedule.

9.1 Future work

Using the flow-shop scheduling heuristics to perform run-time scheduling of TTA processor functional units is a direction for future research. The interconnect between registers and functional units could be scheduled at run-time. At the moment, the TTA processors use compile-time scheduled VLIW-style code to program the data transports.

It would also be interesting to design alternative versions of the scheduling co-processor. One alternative would be to design the co-processor without internal LUTs; that would place more work on the CPU - co-processor interface, since the amount of transmitted data would increase considerably. Another, orthogonal, option would be to design the co-processor for continuous scheduling and dispatching. This would

make the co-processor better suited for applications that continuously supply data to be processed.

Finally, a design space exploration framework could be designed for optimizing the functionality of a piecewise predictable application on a heterogeneous multiprocessor system: after creating an initial solution, the system's performance would be evaluated and fed back to the exploration tool that would refine the task to PE mapping and the partitioning of jobs.

References

- Agerwala T & Chatterjee S (2005) Computer architecture: Challenges and opportunities for the next decade. *IEEE Micro* 25(3): 58–69.
- Ahn JH, Dally WJ, Khailany B, Kapasi UJ & Das A (2004) Evaluating the Imagine stream architecture. *Proc. International Symposium on Computer Architecture, München, Germany*, 14–25.
- Al-Kadi G & Terechko AS (2009) A hardware task scheduler for embedded video processing. *Proc. High Performance Embedded Architectures and Compilers, Paphos, Cyprus*, 140–152.
- Alle M, Biswas J & Nandy SK (2006) High performance VLSI architecture design for H.264 CAVLC decoder. *Proc. International Conference on Application-specific Systems, Architectures and Processors, Steamboat Springs, CO*, 317–322.
- Altera (2008) Altera corp. Nios II processor reference handbook: www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf.
- Arvind, Nikhil R, Rosenband D & Dave N (2004) High-level synthesis: An essential ingredient for designing complex ASICs. *Proc. IEEE/ACM International Conference on Computer Aided Design, San Jose, CA*, 775–782.
- Atienza D, Angiolini F, Murali S, Pullini A, Benini L & De Micheli G (2008) Network-on-chip design and synthesis outlook. *Integration, the VLSI Journal* 41(3): 340–359.
- Bagchi TP, Gupta JND & Sriskandarajah C (2006) A review of TSP based approaches for flowshop scheduling. *European Journal of Operational Research* 169(3): 816–854.
- Bajaj R & Agrawal DP (2004) Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems* 15(2): 107–118.
- Balarin F, Lavagno L, Murthy P & Sangiovanni-Vincentelli A (1998) Scheduling for embedded real-time systems. *IEEE Design and Test of Computers* 15(1): 71–82.
- Balfour J, Dally WJ, Black-Schaeffer D, Parikh V & Park J (2008) An energy-efficient processor architecture for embedded systems. *Computer Architecture Letters* 7(1): 29–32.
- Benini L, Macchiarulo L, Macii A & Poncino M (2002) Layout-driven memory synthesis for embedded systems-on-chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 10(2): 96–105.

- Benoit P, Torres L, Sassatelli G, Robert M & Cambon G (2005) Automatic task scheduling / loop unrolling using dedicated RTR controllers in coarse grain reconfigurable architectures. Proc. 19th IEEE International Parallel and Distributed Processing Symposium, Denver, CO, 148a–148a.
- Berger A (2002) *Embedded Systems Design: An Introduction to Processes, Tools and Techniques*. CMP Books.
- Bhattacharya B & Bhattacharyya S (2000) Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. Proc. International Workshop on Rapid System Prototyping, Paris, France, 84–89.
- Bhattacharya B & Bhattacharyya S (2001) Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing* 49(10): 2408–2421.
- Bhattacharya B & Bhattacharyya SS (1999) Parameterized modeling and scheduling of dataflow graphs. Technical Report UMIACS-TR-99-73, Institute for Advanced Computer Studies, University of Maryland at College Park.
- Bhattacharyya SS, Brebner G, Janneck JW, Eker J, von Platen C, Mattavelli M & Raullet M (2008) OpenDF - A dataflow toolset for reconfigurable hardware and multicore systems. Technical Report BTH-00422, Dept. of Systems and Software Engineering, Blekinge Institute of Technology.
- Bonney MC & Gundry SW (1976) Solutions to the constrained flowshop sequencing problem. *Operations Research Quarterly* 24: 869–883.
- Booch G (2007) *Object-Oriented Analysis and Design with Applications*, 3rd ed. Addison-Wesley Professional, Upper Saddle River, NJ.
- Borkar S (1999) Design challenges of technology scaling. *IEEE Micro* 19(4): 23–29.
- Boutellier J, Bhattacharyya SS & Silvén O (2007a) Low-overhead run-time scheduling for fine-grained acceleration of signal processing systems. Proc. IEEE Workshop on Signal Processing Systems, Shanghai, China, 457–462.
- Boutellier J, Bhattacharyya SS & Silvén O (2009a) A low-overhead scheduling methodology for fine-grained acceleration of signal processing systems. *Journal of Signal Processing Systems* (to appear).
- Boutellier J, Cevrero A, Brisk P & Ienne P (2009b) Architectural support for the orchestration of fine-grained multiprocessing for portable streaming applications. Proc. IEEE Workshop on Signal Processing Systems, Tampere, Finland, (to appear).
- Boutellier J, Jääskeläinen P & Silvén O (2007b) Run-time scheduled hardware acceleration of MPEG-4 video decoding. Proc. International Symposium on System-on-Chip, Tampere, Finland, 1–4.

- Boutellier J, Lucarz C, Lafond S, Gomez VM & Mattavelli M (2009c) Quasi-static scheduling of CAL actor networks for Reconfigurable Video Coding. *Journal of Signal Processing Systems* (to appear).
- Boutellier J, Sadhanala V, Lucarz C, Brisk P & Mattavelli M (2008) Scheduling of dataflow models within the Reconfigurable Video Coding framework. *Proc. IEEE Workshop on Signal Processing Systems*, Washington, DC, 182–187.
- Boutellier J, Silvén O & Erdelyi T (2007c) Restructuring a software based MPEG-4 video decoder for short latency hardware acceleration. *Proc. SPIE Electronic Imaging 2007*, San Jose, CA, 6507(08).
- Brest J & Zerovnik J (1998) An approximation algorithm for the asymmetric traveling salesman problem. *Ricerca Operativa* 28: 59–67.
- Buck J & Vaidyanathan R (2000) Heterogeneous modeling and simulation of embedded systems in El Greco. *Proc. Eighth International Workshop on Hardware/Software Codesign*, San Diego, CA, 142–146.
- Burleson W, Ko J, Niehaus D, Ramamritham K, Stankovic J, Wallace G & Weems C (1999) The Spring scheduling co-processor: A scheduling accelerator. *IEEE Transactions on VLSI Systems* 7(1): 38–48.
- Carpaneto G, Amico MD & Toth P (1995a) Exact solution of large-scale, asymmetric traveling salesman problems. *ACM Transactions on Mathematical Software* 21: 394–409.
- Carpaneto G, Dell'Amico M & Toth P (1995b) Algorithm 750: CDT: a subroutine for the exact solution of large-scale, asymmetric traveling salesman problems. *ACM Transactions on Mathematical Software* 21(4): 410–415.
- Chen YS, Shih CS & Kuo TW (2007) Dynamic task scheduling and processing element allocation for multi-function SoCs. *Proc. Real Time and Embedded Technology and Applications Symposium*, Bellevue, WA, 81–90.
- Chien SL, Chen CY, Chao WM, Huang YW & Chen LG (2003) Analysis and hardware architecture for global motion estimation in MPEG-4 Advanced Simple profile. *Proc. International Symposium on Circuits and Systems*, Bangkok, Thailand, 2: 720–723.
- Cho Y, Zergainoh NE, Yoo S, Jerraya AA & Choi K (2007) Scheduling with accurate communication delay model and scheduler implementation for multiprocessor system-on-chip. *Design Automation for Embedded Systems* 11(2-3): 167–191.
- Cortés LA, Eles P & Peng Z (2005) Quasi-static scheduling for multiprocessor real-time systems with hard and soft tasks. *Proc. International Workshop on Real-Time Computing Systems and Applications*, IEEE Computer Society, Hong Kong, China,

422–428.

- Corwin E & Logar A (2004) Sorting in linear time - variations on the bucket sort. *Journal of Computing Sciences in Colleges* 20(1): 197–202.
- Dally WJ, Balfour J, Black-Shaffer D, Chen J, Harting RC, Parikh V, Park J & Sheffield D (2008) Efficient embedded computing. *Computer* 41(7): 27–32.
- Dang P (2006) An efficient VLSI architecture for H.264 subpixel interpolation coprocessor. *Proc. International Conference on Consumer Electronics, Las Vegas, NV*, 87–88.
- Das D, Dasgupta P & Das P (1997) A new method for transparent fault tolerance of distributed programs on a network of workstations using alternative schedules. *Proc. 3rd International Conference on Algorithms and Architectures for Parallel Processing, Melbourne, Australia*, 479–486.
- Denolf K, Chirila-Rus A, Schumacher P, Turney R, Vissers K, Verkest D & Corporaal H (2007) A systematic approach to design low-power video codec cores. *EURASIP Journal on Embedded Systems* 2007(1): 42–42.
- Eker J & Janneck J (2003) CAL language report. Technical Report Tech. Memo UCB/ERL M03/48, UC Berkeley.
- Falk J, Keinert J, Haubelt C, Teich J & Bhattacharyya SS (2008) A generalized static data flow clustering algorithm for MPSOC scheduling of multimedia applications. *Proc. ACM International Conference on Embedded software, ACM, Atlanta, GA*, 189–198.
- Fan L, Ma S & Wu F (2004) Overview of AVS video standard. *Proc. IEEE International Conference on Multimedia and Expo, Taipei, Taiwan*, 1: 423–426.
- Ferri C, Viescas A, Moreshet T, Bahar RI & Herlihy M (2008) Energy efficient synchronization techniques for embedded architectures. *Proc. ACM Great Lakes Symposium on VLSI, ACM, Orlando, FL*, 435–440.
- Framinan JM, Gupta JND & Leisten R (2004) A review and classification of heuristics for permutation flow-shop scheduling with makespan objective. *Journal of The Operational Research Society* 55(12): 1243–1255.
- French S (1982) *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*. Ellis Horwood Limited, Chichester, UK.
- Gangwal OP, Nieuwland A & Lippens P (2001) A scalable and flexible data synchronization scheme for embedded HW-SW shared-memory systems. *Proc. International Symposium on Systems Synthesis, Montréal, Canada*, 1–6.
- Gao GR, Tio R & Hum HHJ (1988) Design of an efficient dataflow architecture without

- dataflow. Proc. International Conference on Fifth-Generation Computers, Tokyo, Japan, 861–868.
- Girault A, Lee B & Lee EA (1999) Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18: 742–760.
- Golubeva O, Loghi M & Poncino M (2007) On the energy efficiency of synchronization primitives for shared-memory single-chip multiprocessors. Proc. Great Lakes Symposium on VLSI, Stresa-Lago Maggiore, Italy, 489–492.
- Graunke G & Thakkar S (1990) Synchronization algorithms for shared-memory multiprocessors. *Computer* 23(6): 60–69.
- Gu R, Janneck JW, Raulet M & Bhattacharyya SS (2009) Exploiting statically schedulable regions in dataflow programs. Proc. IEEE International Conference on Acoustics, Speech and Signal Processing, Taipei, Taiwan, 565–568.
- Gupta JND & Stafford EF (2006) Flowshop scheduling research after five decades. *European Journal of Operational Research* 169(3): 699–711.
- Henniger O & Neumann P (1995) Test case generation based on formal specifications in Estelle. Proc. IEEE International Workshop on Factory Communication Systems, Leysin, Switzerland, 135–141.
- Herlihy M & Moss JEB (1993) Transactional memory: Architectural support for lock-free data structures. Proc. International Symposium on Computer Architecture, ACM, San Diego, CA, 289–300.
- Hildebrandt J, Golasowski F & Timmermann D (1999) Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems. Proc. Euromicro Conference on Real-Time Systems, York, United Kingdom, 208–215.
- Horowitz M, Alon E, Patil D, Naffziger S, Kumar R & Bernstein K (2005) Scaling, power, and the future of CMOS. Proc. IEEE International Electron Devices Meeting, Washington, DC, 7–15.
- Horstmannshoff J & Meyr H (1999) Optimized system synthesis of complex RT level building blocks from multirate dataflow graphs. Proc. International Symposium on System Synthesis, San Jose, CA, 38–43.
- Huang TY, Jian GA, Chu JC, Su CL & Guo JI (2008) Joint algorithm/code-level optimization of H.264 video decoder for mobile multimedia applications. Proc. IEEE International Conference on Acoustics, Speech and Signal Processing, Las Vegas, NV, 2189–2192.
- ITU-T (2005) H.263: Video coding for low bit rate communication, ITU-T recommen-

ation E 27414.

- Jain M, Balakrishnan M & Kumar A (2001) ASIP design methodologies: Survey and issues. Proc. International Conference on VLSI Design, Bangalore, India, 76–81.
- Janneck J, Miller I, Parlour D, Roquier G, Wipliez M & Raulet M (2008) Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. Proc. IEEE Workshop on Signal Processing Systems, Washington, DC, 287–292.
- Jantsch A & Sander I (2005) Models of computation and languages for embedded system design. IEE Proceedings - Computers and Digital Techniques 152(2): 114–129.
- Jerraya AA, Bouchhima A & Pétrot F (2006) Programming models and HW-SW interfaces abstraction for multi-processor SoC. Proc. DAC '06: Proceedings of the 43rd annual Design Automation Conference, ACM, San Francisco, CA, 280–285.
- Jääskeläinen P, Guzma V, Cilio A & Takala J (2007) Codesign toolset for application-specific instruction-set processors. Proc. SPIE Vol. 6507: Multimedia on Mobile Devices, San Jose, CA, 65070X.
- Jung H, Lee K & Ha S (2002) Efficient hardware controller synthesis for synchronous dataflow graph in system level design. IEEE Transactions on Very Large Scale Integration Systems 10(4): 423–428.
- Kawakami K, Kuroda M, Kawaguchi H & Yoshimoto M (2007) Power and memory bandwidth reduction of an H.264/AVC HDTV decoder LSI with elastic pipeline architecture. Proc. Asia and South Pacific Design Automation Conference, Yokohama, Japan, 292–297.
- Kaxiras S, Narlikar G, Berenbaum AD & Hu Z (2001) Comparing power consumption of an SMT and a CMP DSP for mobile phone workloads. Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, ACM, Atlanta, GA, 211–220.
- Khailany B, Williams T, Lin J, Long E, Rygh M, Tovey D & Dally W (2008) A programmable 512 GOPS stream processor for signal, image, and video processing. IEEE Journal of Solid-State Circuits 43(1): 202–213.
- Kim N, Austin T, Baauw D, Mudge T, Flautner K, Hu J, Irwin M, Kandemir M & Narayanan V (2003) Leakage current: Moore's law meets static power. Computer 36(12): 68–75.
- Kis T & Pesch E (2005) A review of exact solution methods for the non-preemptive multiprocessor flowshop problem. European Journal of Operational Research 164(3): 592–608.
- Ko DI & Bhattacharyya S (2005) Dynamic configuration of dataflow graph topology for

- DSP system design. Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing, Philadelphia, PA, 5: v/69–v/72 Vol. 5.
- Kuacharoen P, Shalan M & Mooney V (2003) A configurable hardware scheduler for real-time systems. Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, NV, 96–101.
- Kumar A, Mesman B, Theelen B, Corporaal H & Ha Y (2008) Analyzing composability of applications on MPSoC platforms. *Journal of Systems Architecture* 54(3-4): 369–383.
- Kumar R, Tullsen D, Jouppi N & Ranganathan P (2005) Heterogeneous chip multiprocessors. *Computer* 38(11): 32–38.
- Kumar S, Hughes CJ & Nguyen A (2007) Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. Proc. International Symposium on Computer Architecture, ACM, San Diego, CA, 162–173.
- Kumar VV & Lach J (2006) Highly flexible multimode digital signal processing systems using adaptable components and controllers. *EURASIP Journal on Applied Signal Processing* Article ID 79595.
- Kuon I & Rose J (2007) Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26(2): 203–215.
- Lattard D, Beigne E, Clermidy F, Durand Y, Lemaire R, Vivet P & Berens F (2008) A reconfigurable baseband platform based on an asynchronous network-on-chip. *IEEE Journal of Solid-State Circuits* 43(1): 223–235.
- Lee E (2001) Computing for embedded systems. Proc. IEEE Instrumentation and Measurement Technology Conference, Budapest, Hungary, 3: 1830–1837.
- Lee E & Ha S (1989) Scheduling strategies for multiprocessor real-time DSP. Proc. Global Telecommunications Conference. Communications Technology for the 1990s and Beyond, Dallas, TX, 1279–1283 vol.2.
- Lee E & Messerschmitt D (1987) Synchronous data flow. *Proceedings of the IEEE* 75(9): 1235–1245.
- Lee E & Sangiovanni-Vincentelli A (1998) A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17(12): 1217–1229.
- Lee EA (1988) VLSI Signal Processing III: Recurrences, Iteration, and Conditionals in Statically Scheduled Block Diagram Languages, 330–340. IEEE Press.
- Lin CC, Chang HC, Guo JI & Chen KH (2004) Reconfigurable low power MPEG-4

- texture decoder IP design. Proc. The IEEE Asia-Pacific Conference on Circuits and Systems, Tainan, Taiwan, 1: 153–156 vol.1.
- Ling N & Wang NT (2003) A real-time video decoder for digital HDTV. *Journal of VLSI Signal Processing* 33(3): 295–306.
- Liu TM, Lin TA, Wang SZ, Lee WP, Yang JY, Hou KC & Lee CY (2007) A 125 μ W, fully scalable MPEG-2 and H.264/AVC video decoder for mobile applications. *IEEE Journal of Solid-State Circuits* 42(1): 161–169.
- Lucarz C, Mattavelli M, Thomas-Kerr J & Janneck J (2007) Reconfigurable Media Coding: a new specification model for multimedia coders. Proc. IEEE Workshop on Signal Processing Systems, Shanghai, China, 481–486.
- Lucarz C, Mattavelli M, Wipliez M, Roquier G, Raulet M, Janneck JW, Miller ID & Parlour DB (2008) Dataflow/actor-oriented language for the design of complex signal processing systems. Proc. Conference on Design and Architectures for Signal and Image Processing, Bruxelles, Belgium, 168–175.
- Martins E, Almeida L & Fonseca JA (2005) An FPGA-based coprocessor for real-time fieldbus traffic scheduling: Architecture and implementation. *Journal of System Architecture* 51(1): 29–44.
- McKeown S, Woods R & McAllister J (2008) Power efficient dsp datapath configuration methodology for fpga. Proc. International Conference on Field Programmable Logic and Applications, 515–518.
- MPEG (2004) Information technology - coding of audio-visual objects - part 2: Visual, ISO/IEC international standard 14496-2:2004, May 2004.
- MPEG (2006) Information technology - generic coding of moving pictures and associated audio information: Video, ISO/IEC international standard 13818-2:2000, April 2006.
- MPEG (2008a) Information technology - coding of audio-visual objects - part 10: Advanced Video Coding, ISO/IEC international standard 14496-10:2008, September 2008.
- MPEG (2008b) ISO/IEC FCD 23001-4 Information technology – MPEG systems technologies – Part 4: Codec configuration representation.
- MPEG (2008c) ISO/IEC FCD 23002-4 Information technology – MPEG video technologies – Part 4: Video tool library.
- Munshi A (2008) OpenCL: Parallel computing on the GPU and CPU, <http://s08.idav.ucdavis.edu/munshi-opencl.pdf>.
- Naveh B & Sichi JV (2008) JGraph Sourceforge project,

- <http://sourceforge.net/projects/jgraph>.
- Nedjah N & de Macedo Mourelle L (2007) *Co-design for System Acceleration: A Quantitative Approach*. Springer-Verlag New York, Inc., Secaucus, NJ.
- Nezan J, Raulet M, Wipliez M & Piat J (2008) SDF4J dataflow Sourceforge project, <http://sourceforge.net/projects/sdf4j>.
- Oh H & Ha S (2002) Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints. *Proc. International Symposium on Hardware/Software Codesign, ACM, Estes Park, CO*, 133–138.
- Oksman V, Ollikainen V, Noppari E, Herrero C & Tammela A (2008) Podracing: Experimenting with mobile TV content consumption and delivery methods. *Multimedia Systems* 14(2): 105–114.
- Paci G, Poletti F, Benini L & Marchal P (2007) Exploring temperature-aware design in low-power MPSoCs. *International Journal of Embedded Systems* 3: 43–51.
- Palmer DS (1965) Sequencing jobs through a multistage process in the minimum total time: A quick method of obtaining a near optimum. *Operations Research Quarterly* 16(1): 101–107.
- Patel K, Macii A & Poncino M (2004) *Ultra Low-Power Electronics and Design*, chapter Energy-efficient Shared Memory Architectures For Multi-Processor Systems-on-Chip, 84–102. Kluwer Academic Publishers.
- Patterson DA & Hennessy J (2005) *Computer Organization and Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 3 edition.
- Pereira FC & Ebrahimi T (2002) *The MPEG-4 Book*. Prentice Hall, Upper Saddle River, NJ.
- Pino JL, Bhattacharyya SS & Lee EA (1995) A hierarchical multiprocessor scheduling system for DSP applications. *Proc. Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA*, 1: 122–126.
- Pozzi L, Atasu K & Jenne P (2006) Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25(7): 1209–1229.
- Rabaey JM (2000) LNCS Volume 1896: *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, chapter Silicon Platforms for the Next Generation Wireless Systems - What Role Does Reconfigurable Hardware Play?, 277–285. Springer.
- Rahal-Arabi T & Park HJ (2008) Optimizing the mobile platform for energy efficiency. *Proc. Electronic Components and Technology Conference, Orlando, FL*, 502–507.

- Ramamoorthy CV & Li HF (1974) Efficiency in generalized pipeline networks. Proc. AFIPS '74: National Computer Conference and Exposition, ACM, Chicago, IL, 625–635.
- Ramamritham K, Stankovic J & Shiah P (1990) Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems* 1(2): 184–194.
- Rambaldi R, Ugazzoni A & Guerrieri R (1998) A 35 μ W 1.1 V gate array 8x8 IDCT processor for video-telephony. Proc. IEEE International Conference on Acoustics, Speech and Signal Processing, Seattle, WA, 5: 2993–2996.
- Reinikka T (2009) Transport triggered architecture based motion compensation of H.264 video. Master's thesis, University of Oulu, Oulu, Finland.
- Rintaluoma T, Silvén O & Raekallio J (2006) Interface overheads in embedded multimedia software. Proc. International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, Samos, Greece, 5–14.
- Sáez S, Vila J, Crespo A & Garcia A (1999) A hardware scheduler for complex real-time systems. Proc. IEEE International Symposium on Industrial Electronics, Bled, Slovenia, 43–48.
- Saito H, Stavrakos NJ, Polychronopoulos CD & Nicolau A (2000) The design of the PROMIS compiler - towards multi-level parallelization. *International Journal of Parallel Programming* 28(2): 195–212.
- Salmela P, Antikainen J, Pitkänen T, Silvén O & Takala J (2009) 3G Long Term Evolution baseband processing with application-specific processors. *International Journal of Digital Multimedia Broadcasting Special Issue on Software-Defined Radio and Broadcasting*: to appear.
- Schumacher P, Denolf K, Chirila-Rus A, Turney R, Fedele N, Vissers K & Bormans J (2005) A scalable, multi-stream MPEG-4 video decoder for conferencing and surveillance applications. Proc. IEEE International Conference on Image Processing, Genoa, Italy, 2: 886–889.
- Seidel H, Robelly P, Herhold P & Fettweis G (2005) A low power soft core for multi-standard video applications. Proc. Global Signal Processing Expo & Conference, Santa Clara, CA.
- Shapiro LG & Stockman GC (2001) *Computer Vision*. Prentice Hall.
- Silvén O & Jyrkkä K (2005) Observations on power-efficiency trends in mobile communication devices. Proc. LNCS 3553: *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Springer, 142–151.

- Silvén O & Jyrkkä K (2007) Observations on power-efficiency trends in mobile communication devices. *EURASIP Journal on Embedded Systems* 2007.
- SMPTE (2005) Technology comm. C24 on video compression technology, proposed SMPTE standard for television: VC-1 compressed video bitstream format and decoding process, SMPTE 421M.
- Sriram S & Bhattacharyya SS (2000) *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, New York, NY.
- Stabernack B, Wels KI & Hubert H (2007) A system on a chip architecture of an H.264/AVC coprocessor for DVB-H and DMB applications. *IEEE Transactions on Consumer Electronics* 53(4): 1529–1536.
- Stärner J, Adomat J, Furunäs J & Lindh L (1996) Real-time scheduling co-processor in hardware for single and multiprocessor systems. *Proc. Euromicro Workshop on Real-Time Systems, L'Aquila, Italy*, 164–168.
- Steinke S, Wehmeyer L, Lee BS & Marwedel P (2002) Assigning program and data objects to scratchpad for energy reduction. *Proc. Design, Automation and Test in Europe Conference and Exhibition, Paris, France*, 409–415.
- Stolberg HJ, Berekovic M, Pirsch P & Runge H (2001) The MPEG-4 Advanced Simple profile - a complexity study. *Proc. Workshop and Exhibition on MPEG-4, San Jose, CA*, 33–36.
- Thiele L, Strehl K, Ziegenbein D, Ernst R & Teich J (1999) FunState—an internal design representation for codesign. *Proc. IEEE/ACM International Conference on Computer-Aided Design, San Jose, CA*, 558–565.
- Tiwari V, Malik S, Wolfe A & Lee MTC (1996) Instruction level power analysis and optimization of software. *The Journal of VLSI Signal Processing* 13(2-3): 223–238.
- Todman T, Constantinides G, Wilton S, Mencer O, Luk W & Cheung P (2005) Reconfigurable computing: Architectures and design methods. *IEE Proceedings - Computers and Digital Techniques* 152(2): 193–207.
- Tomasulo RM (1967) An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development* 11(1).
- Udupa A, Govindarajan R & Thazhuthaveetil MJ (2009) Synergistic execution of stream programs on multicores with accelerators. *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, ACM, Dublin, Ireland*, 99–108.
- Varea M, Al-Hashimi BM, Cortés LA, Eles P & Peng Z (2006) Dual flow nets: Modeling the control/data-flow relation in embedded systems. *Transactions on Embedded*

- Computing Systems 5(1): 54–81.
- Vermeulen F, Catthoor F, Verkest D & de Man H (2000) Formalized three-layer system-level reuse model and methodology for embedded data-dominated applications. Proc. Design, Automation and Test in Europe Conference and Exhibition, Paris, France, 92–98.
- von Platen C & Eker J (2008) Efficient realization of a CAL video decoder on a mobile terminal. Proc. IEEE Workshop on Signal Processing Systems, Washington, DC, 176–181.
- Wang SH, Peng WH, He Y, Lin GY, Lin CY, Chang SC, Wang CN & Chiang T (2005) A software-hardware co-implementation of MPEG-4 advanced video coding (AVC) decoder with block level pipelining. Journal of VLSI Signal Processing 41: 93–110.
- Wipliez M, Roquier G, Raulet M, Nezan JF & Deforges O (2008) Code generation for the MPEG Reconfigurable Video Coding framework: From CAL actions to C functions. Proc. IEEE International Conference on Multimedia and Expo, Hannover, Germany, 1049–1052.
- Wismer D (1972) Solution of the flowshop scheduling problem with no intermediate queues. Operations Research 20: 689–697.
- Wolf W (2003) A decade of hardware/software codesign. Computer 36(4): 38–43.
- Wolf W (2004) The future of multiprocessor systems-on-chips. Proc. 41st Design Automation Conference, San Diego, CA, 681–685.
- Wolf W (2006) High-Performance Embedded Computing. Morgan Kaufmann, San Francisco, CA.
- Wolf W, Jerraya A & Martin G (2008) Multiprocessor system-on-chip (MPSoC) technology. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27(10): 1701–1713.
- Xu J & Parnas DL (2000) Priority scheduling versus pre-run-time scheduling. International Journal of Time-Critical Computing Systems 18(1): 7–23.
- Xvid Solutions (2008) The XViD codec, <http://www.xvid.org/>.

326. Selek, István (2009) Novel evolutionary methods in engineering optimization—towards robustness and efficiency
327. Härkönen, Janne (2009) Improving product development process through verification and validation
328. Peiponen, Kai-Erik (2009) Optical spectra analysis of turbid liquids
329. Kettunen, Juha (2009) Essays on strategic management and quality assurance
330. Ahonen, Timo (2009) Face and texture image analysis with quantized filter response statistics
331. Uusipaavalniemi, Sari (2009) Framework for analysing and developing information integration. A study on steel industry maintenance service supply chain
332. Risikko, Tanja (2009) Safety, health and productivity of cold work. A management model, implementation and effects
333. Virtanen, Markku (2009) Mathematical modelling of flow and transport as link to impacts in multidiscipline environments
334. Liimatainen, Henriikki (2009) Interactions between fibres, fines and fillers in papermaking. Influence on dewatering and retention of pulp suspensions
335. Ghaboosi, Kaveh (2009) Intelligent medium access control for the future wireless networks
336. Möttönen, Matti (2009) Requirements engineering. Linking design and manufacturing in ICT companies
337. Leinonen, Jouko (2009) Analysis of OFDMA resource allocation with limited feedback
338. Tick, Timo (2009) Fabrication of advanced LTCC structures for microwave devices
339. Ojansivu, Ville (2009) Blur invariant pattern recognition and registration in the Fourier domain
340. Suikkanen, Pasi (2009) Development and processing of low carbon bainitic steels
341. García, Verónica (2009) Reclamation of VOCs, n-butanol and dichloromethane, from sodium chloride containing mixtures by pervaporation. Towards efficient use of resources in the chemical industry

Book orders:
OULU UNIVERSITY PRESS
P.O. Box 8200, FI-90014
University of Oulu, Finland

Distributed by
OULU UNIVERSITY LIBRARY
P.O. Box 7500, FI-90014
University of Oulu, Finland

S E R I E S E D I T O R S

A
SCIENTIAE RERUM NATURALIUM

Professor Mikko Siponen

B
HUMANIORA

University Lecturer Elise Kärkkäinen

C
TECHNICA

Professor Hannu Heusala

D
MEDICA

Professor Helvi Kyngäs

E
SCIENTIAE RERUM SOCIALIUM

Senior Researcher Eila Estola

F
SCRIPTA ACADEMICA

Information officer Tiina Pistokoski

G
OECONOMICA

University Lecturer Seppo Eriksson

EDITOR IN CHIEF

University Lecturer Seppo Eriksson

PUBLICATIONS EDITOR

Publications Editor Kirsti Nurkkala

ISBN 978-951-42-9271-2 (Paperback)

ISBN 978-951-42-9272-9 (PDF)

ISSN 0355-3213 (Print)

ISSN 1796-2226 (Online)

