Mapping Framework for Heterogeneous Reconfigurable Architectures

Combining Temporal Partitioning and Multiprocessor Scheduling

Popp, Andreas

Publication date:
2010

Document Version
Accepted author manuscript, peer reviewed version

Mapping Framework for Heterogeneous Reconfigurable Architectures

Combining Temporal Partitioning and Multiprocessor Scheduling

PhD Dissertation

Andreas Popp

AALBORG UNIVERSITY

Mapping Framework for Heterogeneous Reconfigurable Architectures
- Combining Temporal Partitioning and Multiprocessor Scheduling

PhD Dissertation

ISBN: 978-87-92328-44-1
August 2010

Technology Platforms Section
Department of Electronic Systems, Aalborg University
Niels Jernes Vej 12
9220 Aalborg Øst

Preface

This dissertation is submitted to the Faculty of Engineering, Science and Medicine at Aalborg University in partial fulfillment of the requirements for the degree of Doctor of Philosophy. The work was conducted from April 2007 to August 2010 as a project funded by the Center for Software Defined Radio at Aalborg University in cooperation with Rohde & Schwarz Technology Center A/S in Aalborg.

Acknowledgements

I would like to thank my supervisors Yannick Le Moullec and Peter Koch for their inspiration, contributions, and support during the whole process from initiation to finish of the work. I also thank Rohde & Schwarz Technology Center A/S in Aalborg for their time for good discussions, pleasant working environment, and support for the project. A special thank goes to my colleagues at OFFIS, Institute for Information Technology in Oldenburg, Germany during my stay from Feb. 2009 to June 2009. Especially thanks to Kim Grüttner for his willingness to organize the stay, and to both him and Andreas Herrholz for their cooperation and contribution during my stay in Germany. I also thank Christophe Jégo from Telecom Bretagne for his hospitality in September 2009 as well as our cooperation. Furthermore, I would like the thank my colleagues at Aalborg University, Jesper Michael Kristensen, Mehmood Ur Rehman Awan, and Rasmus Abildgren for doing your best in making every workday an enjoyable time. The same goes for the rest of my colleagues in the Technology Platforms Section.

Last but not least I thank my family and friends for your encouragement, company, and support - I know that your company has given me the energy to work. A final thank goes to Helle for withstanding my lack of presence during the lasts months of work - your company and support has been of great value to me.

Aalborg, Denmark, August 2010 Andreas Popp

List of Abbreviations

The following abbreviations are used throughout the thesis:

ASIC Application-Specific Integrated Circuit

CAD Computer Aided Design

CLB Configurable Logic Block

CDFG Control Data Flow Graph

CORDIC COordinate Rotation DIgital Computer

DAG Directed Acyclic Graph

DFG Data Flow Graph

DLS Dynamic Level Scheduling

DPR Dynamic Partial Reconfiguration

DSP Digital Signal Processor

EDA Electronic Design Automation

EDLS Extended Dynamic Level Scheduling

FFT Fast Fourier Transform

FPGA Field-Programmable Gate Array

GPP General-Purpose Processor

HW Hardware

ICAP Internal Configuration Access Port

ICD Inter-processor Communication Delay

ILP Integer Linear Programming

ISE (Xilinx) Integrated System Environment

EDK (Xilinx) Embedded Developer's Kit

MMSE Minimum Mean Square Error

MIMO Multiple-Input Multiple-Output

RTR Run-Time Reconfiguration

SDF Synchronous Data Flow

SDR Software Defined Radio

SW Software

VHDL Very-High-Speed-Integrated-Circuit (VHSIC) Hardware Description Language

Contents

Abstract

The topic of this PhD dissertation is the development of a mapping framework to assist the designers and developers when developing hardware (HW) and software (SW) for reconfigurable systems.

The dissertation focuses on the combination and modification of existing temporal partitioning algorithms for reconfigurable architectures with existing multiprocessor scheduling algorithms. The result is the mapping framework that utilizes these algorithms. Temporal partitioning is used in order to generate HW supernodes, which are clusters of operations performed in HW. The proposed heterogeneous multiprocessor scheduling algorithm treats HW supernodes and tasks performed in SW similarly. The mapping framework provides the designer with a possible schedule for implementation, derived based on input from the designer. This input is composed of three elements: an application model describing the tasks to be performed, an architecture model describing the processing units and their interconnection, and a cost library describing the cost of the implementation of each task on the processing units. In addition to the mapping framework, a cost-model for reconfigurable implementations is proposed. The model is used to conduct a study in order to determine when it is beneficial (in terms of area and execution time costs) to perform a reconfigurable implementation as opposed to a static implementation.

The dissertation is composed of four parts: first, the related challenges and state-of-the-art in relation to reconfigurable architectures is provided. Focus is on design methods and the need for a framework which can handle the mapping of applications onto reconfigurable architectures. Second, the background, considerations, and assumptions of the proposed methodology are given with regard to the utilized models as well as the developed mapping framework. Third, a summary of the contributions is provided related to the published papers. Finally, a conclusion and outlook are given.

The main body of the dissertation is composed of three peer-reviewed papers and one paper submitted for peer-review and presents the contributions. The results show that the cost of a reconfigurable implementation is largely dependent on the reconfiguration time, and thereby dynamic partial reconfiguration has lower cost than dynamic full reconfiguration. The reconfiguration time is relatively high for current Field-Programmable Gate Arrays (FPGAs), but can be accepted for applications with large execution time or massively parallel applications, e.g. multimedia processing.

The developed framework is able to handle binding and scheduling for heterogeneous reconfigurable architectures based on provided application and architecture models, and cost library. Results show that our mapping framework provide feasible schedules that it is possible to implement. Level-based temporal partitioning has the best performance in combination with Extended Dynamic Level Scheduling.

Resumé

Emnet for denne ph.d. afhandling er udviklingen af et mappingværktøj, som kan assistere designere og udviklere under udvikling af hardware (HW) og software (SW) til rekonfigurerbare systemer.

Afhandlingen fokuserer på kombination og modificering af eksisterende algoritmer til ”temporal partitioning” til rekonfigurerbare arkitekturer med eksisterende multiprocessor scheduleringsalgoritmer. Resultatet heraf er mappingværktøjet som anvender disse algoritmer. ”Temporal partitioning” anvendes til at generere HW super-knuder, der er klynger af hardwareoperationer. Den foreslåede heterogene multiprocessor-scheduleringsalgoritme behandler HW super-knuder og operationer eksekveret i SW ens. Mappingværktøjet tilvejebringer designeren en mulig køreplan, ”schedule”, til implementering, afledt baseret på designerens input. Dette input består af tre elementer: en model for applikationen, som beskriver de operationer, som skal udføres, en arkitekturmodel, som beskriver processeringsenhederne, og hvordan de er forbundne samt et ”cost”-bibliotek, som indeholder ”cost” af implementeringen af hver operation i processeringsenhederne. I tillæg til mappingværktøjet er foreslået en model for ”cost” af rekonfigurerbare implementeringer. Modellen er blevet anvendt i et studium til bestemmelse af, hvornår det er fordelagtigt (målt på ”cost”) at foretage en rekonfigurerbar implementering i modsætning til en statisk implementering.

Afhandlingen består af fire dele: Først gives udfordringer og state-of-the-art i forbindelse med rekonfigurerbare arkitekturer. Der fokuseres på design metoder samt behovet for et værktøj som kan håndtere mapping til rekonfigurbare arkitekturer. Dernæst gives baggrund, overvejelser og antagelser anvendt i mapping værktøjet med hensyn til de anvendte modeller af det udviklede mappingværktøj. Derefter gives et overblik over bidrag til området med relation til de publicerede artikler. Sidst gives en konklusion og perspektivering.

Hoveddelen af afhandlingen består af tre peer-reviewede artikler samt en artikel indsendt til peer-review og beskriver bidrag til emnet mapping metoder til heterogene rekonfigurerbare arkitekturer. Resultaterne viser at cost ved en rekonfigurerbar implementering er overvejende afhængig af rekonfigurationstiden, og derfor har dynamisk partiel rekonfiguration lavere cost end dynamisk fuld rekonfiguration. Rekonfigurationstiden er relativt høj for tidssvarende Field-Programmable Gate Arrays (FPGA'er), men kan accepteres for en applikation som har høj eksekveringstid eller en høj grad af parallellitet i applikationen, f.eks. multimedia-processering.

Det udviklede værktøj kan håndtere binding og schedulering til heterogene, rekonfigurerbare arkitekturer forudsat applikations- og arkitekturmodel samt ”cost”-bibliotek. Resultaterne viser, at mapping-værktøjet tilvejebringer schedules, som det er muligt at implementere. ”Level-based temporal partitioning” yder bedst i kombinationen med ”extended dynamic level scheduling”.

1 Introduction

This chapter contains a motivation of reconfigurable computing architectures, highlighting the research challenges of reconfigurable computing. This is followed by an overview of state-of-the-art in mapping of applications to heterogeneous reconfigurable architectures. Finally, the thesis is formulated and an outline of the dissertation is given.

1.1 Motivation

Implementation of signal processing algorithms in e.g. telecommunication and multimedia applications require processing architectures that provide enough computational power and are, at the same time, subject to constraints in terms of chip area or power consumption. Furthermore, telecommunication devices are required to provide multiple functionalities, some simultaneously, other only at some time-instants.

The aforementioned telecommunication and multimedia applications require more and more advanced functionalities, e.g. for improved spectrum usage or higher video quality. This causes an increase in the performance requirements and complexity of the reconfigurable architecture. When designing such a system, a designer may often follow the design trajectory, shown in Figure 1.1, which starts from a set of specifications and requirements. This is followed by design space exploration and mapping to evaluate implementation alternatives and find the best mapping for the application. The best mapping means fulfilling the requirements and constraints as well as minimizing the cost function

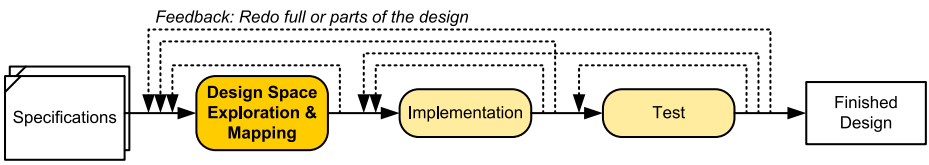


Figure 1.1: Overall design trajectory: The outset is a set of specifications describing the requirements to the application (behavior, physical factor, cost etc.) Then design space exploration and mapping are performed in order to evaluate various implementation alternatives and find the mapping that is the optimum for the application. This is then implemented and tested in order to obtain the final design. At any step of the design, it may be necessary to do an extra iteration or go one or more steps back in order to fulfill the requirements.

(see below). This mapping is then implemented and tested. The design may include several iterations, which in turn, cost money to the company developing the application due to the caused extension of development time.

Finding the correct final design is often a matter of trade-off between costs and performance, with a large design space covering the possible architectures and mappings. Mapping is composed of two design challenges: binding, where it is decided onto which processing unit a task should be implemented, and scheduling, where it is decided when tasks should be executed. We define scheduling to be relative in time, i.e. tasks are scheduled in relation to each other. Design space exploration is the evaluation of possible solutions with regard to a cost function, (1.1), where some of the parameters (e.g. area or execution time) may be constrained by the requirements. Searching the full design space is an extensive task, where obtaining a minimal cost solution within constraints of e.g. execution time or hardware resources is known to be NP-hard [3], and may thus be impossible for designers to complete in reasonable time. Therefore, there is a need for methodologies and tools that can help the designers to prune the design space in a *reasonable* time and thereby be compatible with the company's time schedule. We believe that a tool that can aid the designer during the design space exploration process is necessary in order to shorten the product development process and thereby meet the time-to-market requirements.

The cost, C, of an implementation may be measured by a cost function,

$$C = f(T, A, P, N) \quad , \tag{1.1}$$

where T is execution time, A is area or resource consumption, P is power consumption, and N is the numerical noise. Each of the parameters are assigned weight factors that define their importance, giving a cost function such as $C = \alpha T + \beta A + \gamma P + \ldots$. A cost function could also include other parameters such as price and development time.

Typical architectures for signal processing equipment consist of either (or a combination of) Application-Specific Integrated Circuits (ASIC) and Digital Signal Processors (DSPs). However, these components can generally be considered as fixed and efficient, or flexible and less efficient, respectively. In order to obtain a reasonable trade-off between these two end-points, the trend goes towards reconfigurable architectures [1], which offers this trade-off between ASICs and DSPs. The comparison between various architectures, as proposed in [2], is shown in Table 1.1 and illustrates that reconfigurable hardware can potentially offer the best trade-off by providing medium-high performance and high flexibility.

Reconfigurable hardware is motivated due to several advantages compared to ASIC or SW architectures. First, it is possible to increase the functionality of the same hardware by time-sharing of resources and thus obtain a reduction in chip area. Area-reduction also reduces the static power of the circuit, which is becoming a larger and larger part of the power consumption [4]. As a side effect of area reduction as well as the possibility of using reconfiguration to deactivate circuits, reconfiguration can also reduce the energy consumption. Finally, the flexibility of a reconfigurable hardware platform is of great use for e.g. Software Defined Radio (SDR) [5], where there is a large requirement for flexibility of the architecture. Further motivations and more precise examples of application of reconfigurable hardware are given in Section 1.2.

Technology	Performance	System Design Cost	Cost/Chip	Power	Flexibility
General-Purpose Processor (GPP)	Low	Low	Low-Medium	High	High
Digital Signal Processor (DSP)	Medium	Medium	Low-Medium	Medium	Medium
Reconfigurable Hardware	Medium-High	Medium	Medium-High	Medium-High	High
Application-Specific Integrated Circuit (ASIC)	High	High	Low	Low	Low

Table 1.1: Qualitative comparison of implementation technologies, excerpt from Gokhale and Graham [2, Table 5.1]. Performance is a measure of execution time and throughput. System Design Cost is the total cost of design, i.e. the expenses for tools and engineering.

Even though we have introduced and briefly motivated reconfigurable hardware, it has not yet been defined. We use the definition inspired by Compton and Hauck [6] and define the following terms in Definition 1 and 2.

Definition 1. *Reconfigurable hardware* is hardware incorporating some form of hardware programmability, by which we mean the ability to change the behavior of the logic residing in the hardware.

Definition 2. *Reconfigurable architectures* are processing architectures consisting of one or more reconfigurable hardware units.

Reconfigurable hardware is characterized by its granularity, i.e. the smallest amount of logic that can be reconfigured. We generally distinguish between fine- and coarse-grained reconfigurable logic [7]. *Fine-grained logic* can be reconfigured down to 1 or 2 bit, both in function logic blocks as well as routing between the blocks. An example of fine-grained logic is Field-Programmable Gate Arrays (FPGAs). *Coarse-grained logic* is reconfigurable in quantities of 8-32 bit, and has a structure similar to microprocessors with configurable logic or computation units. The differences between fine- and coarse-grained reconfigurable HW, as proposed in [7], are highlighted in Table 1.2.

Parameter	Fine	Coarse
Configurable logic size	1-2 bit	8-32 bit
Design approach	Logic design	SW design
Performance	Less High	High
Flexibility	High	High-medium

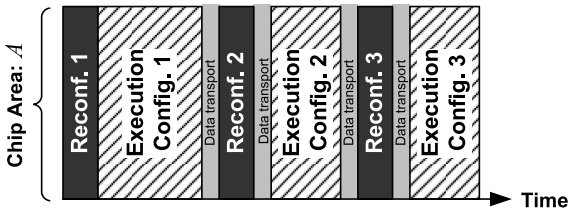
Table 1.2: Qualitative comparison of fine and coarse-grained reconfigurable hardware, excerpt from Hartenstein et al. [7]. Design approach describes the similarity of the design to other technologies.

In parallel with the granularity, reconfiguration is also divided into three categories depending on the frequency and fraction of the device that is reconfigured. These terms are technology independent.

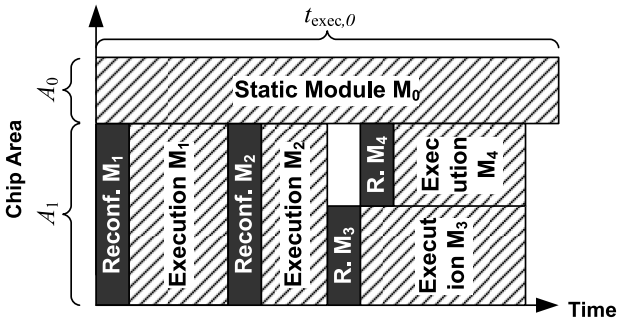
Static configuration is the configuration of the full device at only one single time during operation. This may be either before starting the system, or during the start-up sequence. The functionality of the HW is fixed during the runtime of the system.

Dynamic full reconfiguration is dynamic in the sense that the functionality of the hardware can be reconfigured during the run-time of the system. However, reconfiguration is performed for the full hardware, thus interrupting execution and overwriting internal signals in the hardware. Therefore, it may be necessary to store the internal signals in external memory, as illustrated in Figure 1.2a.

Dynamic Partial Reconfiguration (DPR) or Run-Time Reconfiguration (RTR) is the most flexible type of reconfiguration. Similar to dynamic full reconfiguration the functionality can be changed during runtime. However, it is possible to reconfigure only part of the hardware, while the other parts are performing execution, as illustrated in Figure 1.2b. Those parts performing execution can be either static parts, that are constant during the full runtime, or parts that will be (or have been) reconfigured at other times.



(a) Dynamic full reconfiguration: The full chip area is reconfigured during runtime. Internal signals that must be used later need to be saved to/loaded from external memory. Thus data transport is marked in light gray.



(b) Dynamic partial reconfiguration: Parts of the chip area (A_1) are reconfigured during runtime while other parts (A_0) are still executing.

Figure 1.2: Illustration of dynamic full and partial reconfiguration. Reconfiguration is marked by dark gray whereas execution is marked by the diagonal line pattern.

To the best of our knowledge, the earliest work on reconfigurable architectures started by Estrin et al. in 1959 with the proposed "Fixed-plus-Variable (F+V) Structure Computer" consisting of a fixed computer plus a board with problem-specific optimized logic

blocks that could be replaced according to the application [8]. However, our definition of reconfigurable hardware is different as we rely on programmability and not the physical replacement of modules. Other reconfigurable architectures have been investigated, coarse-grain architectures mainly in the research domain [9, 10]. For further historic architectures, we refer to the surveys referenced later in this section.

Among the reconfigurable hardware devices, we believe that FPGAs are the most used at time of writing. Therefore, the remainder of this work considers reconfigurable architectures based on FPGAs. A recent study among embedded system developers showed that 42% of all current embedded designs contain an FPGA [11]. The most frequently used FPGA vendor was Xilinx. FPGAs contain three layers, where the first layer contains arrays of logic blocks and the second layer the routing between them. The actual configuration of the FPGA is determined by a third layer of memory, determining the functionality of the logic and routing in the first two layers.

At the time of writing, Xilinx is the main company offering a commercially available solution for DPR. Xilinx FPGAs have several interfaces to the configuration memory [12]. For dynamic reconfiguration, the most useful ones are the SelectMAP and Internal Configuration Access Port (ICAP) interfaces with a throughput of 32 bit at 100 MHz. The selectMAP interface can be accessed from outside the FPGA (requiring an external controller) as illustrated in Figure 1.3. Furthermore, the configuration interface can also be accessed by internal signals inside the FPGA, after the FPGA has initially been configured from an external interface [12]. The internal interface is the Internal Configuration Access Port and provides the same throughput as the SelectMAP interface. The ICAP interface can be used from inside an FPGA in order to perform internally controlled partial reconfiguration via a controlling soft-core processor in the static part of the logic.

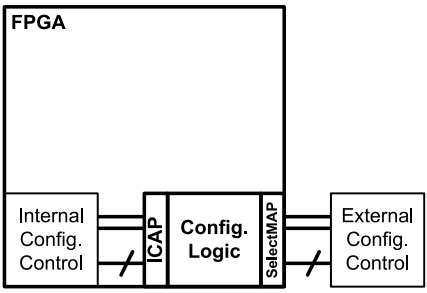


Figure 1.3: Illustration of SelectMAP and ICAP configuration interfaces for configuring Xilinx FPGAs. ”Config.” is short for *configuration*.

The FPGA can only allow one active reconfiguration interface at a time. Therefore, parallelized modules cannot be reconfigured in parallel and have to be scheduled sequentially on the configuration logic. This is illustrated in Figure 1.2b for the reconfiguration of module M_3 and M_4, where the reconfiguration of module M_4 has to wait for the reconfiguration of M_3 to finish.

Surveys of the area of reconfigurable computing have been given by Compton and Hauck [6], Todman et al. [13], Shoa and Shirani [14] and several others, e.g. [15]. It is clear that reconfigurable hardware is not the sole component of a system, but does often form part of an architecture that both consists of several processing units, and is

heterogeneous in its composition in order to meet the constraints of the application. By a *heterogeneous architecture* we mean an architecture consisting of more than one processing unit and that there are at least two different units in the architecture, e.g. a HW/SW architecture.

The cost function in (1.1) reflects that the cost of implementation is dependent on factors such as execution time, area, and power consumption. Signal processing applications typically consist of different blocks of algorithms with different characteristics as illustrated in Figure 1.4. Some may be massively parallel with high throughput and execution time requirements, while other may be sequential in structure with lower requirements to execution time. In order to maintain a low cost (defined by the cost function), a heterogeneous architecture often yields the lowest cost by e.g. performing time-critical parallel parts in HW and more sequential parts in SW. Other solutions may be a multi-processing solution with parallel processors to execute the parallel parts of the algorithm.

It is commonly accepted that there exist many challenges in the design of a system containing heterogeneous reconfigurable architectures. In the next section, we present our view on these challenges.

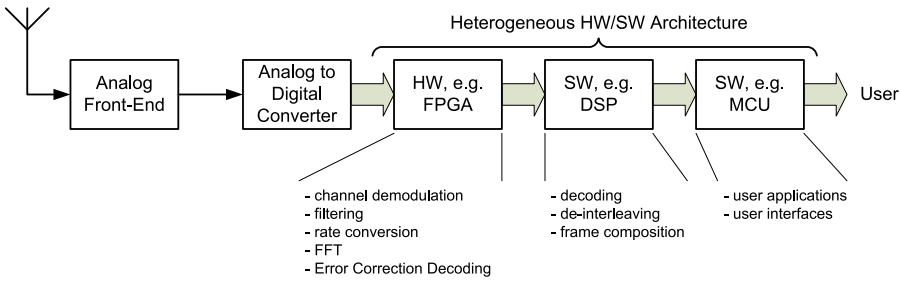


Figure 1.4: Example of a heterogeneous architecture in a radio receiver. After digitization in the analog-to-digital converter, the signal is processed in HW and SW. Some tasks (proposed below the blocks) are most suitable for HW, whereas some are more suitable for SW.

1.2 Challenges in the Design of Heterogeneous Reconfigurable Architectures

The design of applications implemented in heterogeneous reconfigurable architectures is a challenge within several areas. Based on surveys of reconfigurable computing, we see the challenges in three main groups, outlined in the sections below.

Applications

In order to justify the advantages of reconfigurable architectures over ASIC and DSP archictectures, we provide a short survey of some applications implemented in reconfigurable hardware. The survey is a necessary part of the justification, as a quantification of the difference between ASIC and FPGA have shown that FPGA is a factor of 119x

larger in area compared to ASIC for a range of benchmarks ranging from Finite Impulse Response (FIR) filters to Reed-Solomon encoders/decoders [16].

A survey by Todman et al. [13] has shown a 500x speedup and 70% energy savings when using FPGAs instead of microprocessors for some applications. Helmschmidt et al. have presented an implementation of a Rake receiver for a Universal Mobile Telecommunications System (UMTS) and Wireless Local Area Network (WLAN) system in a heterogeneous reconfigurable architecture instead of parallel implementation of the two functionalities [17]. The architecture was a multiprocessor architecture with a controlling microprocessor, a digital signal processor, and a coarse-grain reconfigurable array.

In the field of Software Defined Radio, a reconfigurable Global System for Mobile communication (GSM) and Enhanced Data for GSM Evolution (EDGE) transmitter was implemented on a DSP/FPGA platform by Delahaye et al. [18]. The considered implementation consisted of a Gaussian filter and a Minimum-Shift Keying (MSK) and 8-Phase Shift Keying (8PSK) encoder. The authors investigated and compared full reconfiguration and partial reconfiguration to a static implementation. Results showed a 50% reduction in FPGA utilization due to the sharing of functionalities using both full and partial reconfiguration. The reconfigured parts had significant commonalities between them, so partial reconfiguration reduced the reconfiguration time from 130 ms to 11 ms.

Similarly, Tumeo et al. investigated the use of reconfigurable hardware instead of software for accelerating JPEG encoding [19]. Results showed a speedup for HW execution time of 3.02 compared to SW execution time. Furthermore, area savings of 30% were achieved.

Wang et al. [20] considered an FPGA-implementation of a Multiple-Input Multiple-Output (MIMO) square-root-decoder utilizing COordinate Rotation DIgital Computer (CORDIC) elements. The execution consisted of several stages utilizing the same CORDIC elements, but with changing CORDIC element registers and routing between the stages. Therefore, the reconfigurable area was relatively small leading to relatively low reconfiguration times. The results showed 30% area-savings due to the sharing of resources in the FPGA.

Implementations in reconfigurable hardware are illustrated by these examples to provide opportunities in terms of area and power consumption savings. Furthermore, reconfigurable architectures provide flexibility and adaptability that may be useful in Over-The-Air upgrade scenarios of radio terminals within Software Defined Radio. Other scenarios are that reconfigurable architectures provide redundancy and fault correction for flight and space missions [21].

An implementation in a reconfigurable FPGA provides the opportunity of power savings due to the fact that the accelerator can be more power efficient than a generic implementation as it is customized for the currently running task. However, energy must be considered as the power reduction may easily be nullified by the power consumed during reconfiguration [22]. Recent publications have also shown power and energy gains for FIR filters by Becker et al. [4] who have also presented the GroundHog benchmark [23] for power consumption evaluation for reconfigurable architectures. Power dissipation reduction in embedded controlling applications for the automotive industry have also been demonstrated by [24].

Although many applications have been proposed, we find it an open question to decide what and when to consider a reconfigurable implementation for static applications without the flexibility requirements that are present for SDR applications. We define

static applications as applications where the operations and flow of the algorithms are known at compile-time. In order to be able to determine under which conditions reconfigurable implementations prove more efficient than static HW or SW architectures, we find it necessary to evaluate models of implementations in reconfigurable hardware in order to qualitatively determine the feasibility of reconfigurable architectures. We define *feasibility* based on a cost function. If the costs are lower (or equal) for a reconfigurable than for a static implementation, we define it as feasible. This is further discussion in [A] and Section 2.1.

Architectures and Technologies

Many challenges lie in the reconfigurable architecture, which was also identified as a main challenge in the survey by Todman et al. [13]. The authors emphasized structure of the reconfigurable hardware as well as the interfaces between the elements of the architecture: hardware part, processor, and memory.

Similar conclusions have been made by Hartenstein [25], who identifies one of the main challenges as finding the right trade-off between fine- and coarse-grained architectures, also investigated by [26].

We have already outlined some successful applications of reconfigurable hardware. However, the efficiency of these applications proved to be very dependent on the reconfiguration time of the reconfigurable hardware. In reconfigurable hardware, e.g. FPGAs, the behavior of the reconfigurable hardware is described by a configuration file or bitstream. The bitstream is loaded through a configuration interface, thus the reconfiguration time is dependent on the throughput of the configuration interface. Furthermore, the size of the bitstream is proportional to the configured area [12].

The reduction of reconfiguration time is a challenge that has been covered in many research papers, with some approaches outlined here. These works use the strategy of adding extra hardware resources to provide faster reconfiguration.

Configuration caching has been investigated by Li et al. [27] and consists of adding a cache inside the chip in order to reduce the amount of data to transfer across the chip boundary. Similarly, a multi-context FPGA approach consisting of several layers (contexts) of configuration data in the overlaying configuration memory has been suggested in [28]. The switch between configurations is then performed by a multiplexer. This reduces the configuration time between pre-loaded configurations, but the configuration interface will still be a bottleneck if the number of configurations exceed the number of contexts. This approach (developed by Xilinx), is not integrated in the Virtex family of FPGAs [12], but recently the company, Tabula Inc., has marketed a time-multiplexed FPGA along with necessary development tools [29].

Another technique for reducing the communication across the chip boundary is the compression of configuration data, suggested by the same research group [30], [31]. This has been further investigated by Dandalis and Prasanna [32] proposing an improvement of the area and speed efficiency of the compression technique.

While the mentioned approaches require extra hardware area for the configuration controller and caching memory, another approach is *prefetch* [33], that hides configuration delay by performing reconfiguration and execution in parallel. This has been further developed by Qu et al. [34] where multiple configuration interfaces have been utilized for a single device. The motivation was that when reconfiguration delay is longer than one

execution, more reconfigurations must be performed in parallel, thus a single configuration interface may become a bottleneck.

Design Methods for Reconfigurable Architectures

The lack of design methods for heterogeneous reconfigurable architectures has been emphasized by among other Hartenstein [25] with the lack of Electronic Design Automation (EDA) for increasing and maintaining designer productivity. Furthermore, Todman et al. [13] consider Computer Aided Design (CAD) and compilation tools as significant research challenges.

Design challenges and methods for coping with these have many facets and directions. We see the design problem as a tree, illustrated in Figure 1.5 and outlined in Section 1.3.

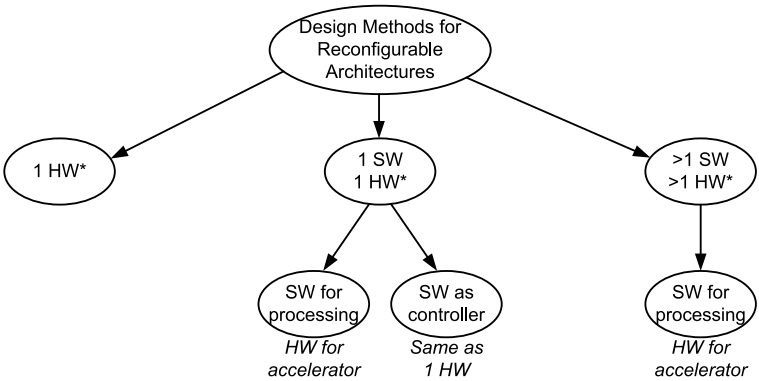


Figure 1.5: Suggested overview of the design cases in reconfigurable computing. HW* denotes that HW may be utilized for either either full reconfiguration or partial reconfiguration. 1 HW is the case when the architecture consists of only one single reconfigurable unit. 1 SW/1 HW is the case of an architecture with a single software and a single reconfigurable hardware unit. This case is again split; if the SW is used for processing, the HW is considered a reconfigurable accelerator for the SW. However, if SW is used only for control of the reconfigurable HW, the case is similar to the 1 HW scenario. If there is more than one of either SW processor or reconfigurable HW unit, the architecture is a heterogeneous reconfigurable architecture. All cases are discussed in Section 1.3.

In the next section, we provide a state-of-the art of the existing work in design methods for reconfigurable computing in general, followed by heterogeneous reconfigurable architectures.

1.3 Mapping of Applications onto Heterogeneous Reconfigurable Architectures

In the following, we describe the state-of-the-art, related and grouped to the overview in Figure 1.5.

Single Reconfigurable Unit As we have described, FPGA execution can be performed in parallel, but the reconfiguration of regions must be scheduled sequentially. A scheduling algorithm for handling this problem for a single FPGA based on single-processor scheduling is proposed by Dittman and Götz [35]. The algorithm assumes equally sized reconfigurable slots on the FPGA. The task model is based on aperiodic tasks, starting at arbitrary times, independent of each other.

Bobda [36] has investigated methods for scheduling data flow graphs onto a reconfigurable device. The work includes both full reconfiguration and partial reconfiguration. For full reconfiguration, the tasks need to be partitioned into clusters of tasks. Each cluster corresponds to a configuration, and the activity of dividing a set of tasks into sequentially executable clusters of tasks is called *temporal partitioning*. Bobda investigates a list-scheduling algorithm for temporal partitioning, followed by an optimization step to reduce the communication across clusters or configurations.

Other approaches to temporal partitioning are the purely list-based approaches by Purna and Bhatia [37]. Purna and Bhatia suggest two temporal partitioning algorithms (level- and clustering-based), each for one of two optimization goals, execution time or communication interface, respectively.

For partially reconfigurable architectures, Bobda [36] has suggested an approach for *temporal placement* which is the task of dealing with both temporal and spatial partitioning of the application. The approach consists of: firstly a decision as to how to cluster components to be placed on the same device, and secondly a placement of the components on the reconfigurable device. For the clustering approach, Bobda used a list-based clustering algorithm followed by spectral-based optimization with higher computational complexity. The partitioning task set is input to a temporal placement algorithm based on a list-based schedule.

Another scheme suggested for utilization of reconfigurable architectures is based on configuration prefetch, and it thus hides the reconfiguration delay if the application allows the parallelization of reconfiguration and execution. Kim et al. [38] have proposed a temporal partitioning algorithm based on an extension of the work by Ganesan and Vemuri [39]. The work by Ganesan and Vemuri was based on a simple division of the hardware area into two equally sized parts, and then a partitioning of the tasks into equally sized blocks. Kim et al. suggest a method that divides a Control Data Flow Graph (CDFG) into sub-CDFGs that fit inside the area-constraints. The algorithm seeks to perform partitioning to maximize speedup of each loop iteration. Another approach for real-time scheduling has been performed by Clemente et al. [40, 41], where a task scheduler was implemented for multiple reconfigurable units.

While the mentioned approaches are developed for architectures with a single reconfigurable HW unit, they cannot handle HW/SW architectures and the required partitioning of the task-set into HW and SW.

Single SW, Single Reconfigurable Unit For HW/SW systems, one design challenge is partitioning of the task graph into HW and SW tasks. Chatha and Vemuri have investigated an iterative algorithm for architectures consisting of one SW unit and one reconfigurable HW unit using full reconfiguration [42]. The approach uses a list-scheduler to obtain the makespan of a partitioning, and uses an iterative partitioner to evaluate the design space until no further minimization of the makespan is achieved.

Similarly to Chatha and Vemuri, Galanis et al. [43] perform early HW/SW partitioning based on execution time, followed by temporal partitioning of the HW tasks. Further on, HW tasks are partitioned into coarse- and fine-grained HW. However, [43] does not describe the combined scheduling of HW and SW tasks, as well as the handling of heterogeneous architecture with more than one SW and one HW unit.

Banerjee et al. have investigated HW/SW partitioning for similar architectures consisting of one SW and one HW unit [44]. The HW is partially reconfigurable and the partitioning scheme takes into account the exclusiveness of the reconfiguration interface as well as a requirement for physically placing tasks in adjacent columns on the FPGA. The approach starts by a dependency graph describing tasks. The optimization problem is formulated as an Integer Linear Programming (ILP) problem, solved by a Kernighan-Lin/Fiduccia-Matheyes heuristic. The results showed that it is necessary to consider partitioning and scheduling simultaneously.

Inspired by Banerjee et al., Redaelli et al. have utilized the same ILP approach to schedule tasks and their reconfiguration on a partially reconfigurable FPGA [45]. Redaelli et al. obtain better results (in terms of lower makespan) than Banerjee, due to the consideration of configuration prefetch, module reuse, and anti-fragmentation strategies. However, they consider only HW execution and not SW. Furthermore, the exploration time for finding a solution was found to be prohibitively long [45]. As an extension to this work, the same research group has developed a methodology for performing partitioning of a task graph into cores with the aim to reuse the same cores for several nodes in the task graph [46].

Miramond and Delosme [47] have investigated HW/SW partitioning, temporal partitioning, and SW scheduling for a HW/SW system with fully flexible partial reconfiguration. The solution space was explored based on a simulated annealing algorithm initiated by a random partition.

Stitt et al. [48] have introduced guidelines for mapping the time-critical parts to HW. Criticality is based on the execution time of that part in relation to the total execution time. One decision parameter was speed-up based on Amdahl‘s law, but no formalized method was considered. Noguera and Badia [49] considered movement of SW to HW to minimize the overall execution time. Prefetch of configurations was considered to hide reconfiguration overhead by overlaying HW reconfiguration with SW execution.

Commonly, all the mentioned approaches cover the mapping and/or implementation process for architectures consisting of one SW unit and one HW unit. However, they do not handle the scheduling and communication overhead for heterogeneous reconfigurable architectures consisting of several SW and several reconfigurable HW units.

Several SW and Several Reconfigurable HW Units As an extension to Banerjee et al., Dittmann et al. [50] have proposed a method for mapping and design of heterogeneous reconfigurable systems consisting of several HW units. The outset is again a dependency

graph describing the tasks to implement. An architecture graph describes the architecture model that consists of several FPGAs and their interconnection. The FPGAs have predetermined (similarly sized) area slots that can be reconfigured, a fixed communication bus, and is constrained by a single configuration interface. The solutions are explored by a genetic algorithm and the outcome is a binding and schedule that can be implemented on the architecture. The quality of the solutions is dependent on a cost library containing the cost of implementation of each task.

Solid work in the field of mapping methods for reconfigurable architectures is available. However, the investigated approaches are either not adapted for heterogeneous, reconfigurable architectures with several SW and HW units, or they have a high computational complexity like those based on genetic algorithms. Therefore, we propose to investigate less complex methodologies for mapping applications onto heterogeneous, reconfigurable architectures. Our proposal consists of an extension and modification of existing low-complexity approaches such that they cover heterogeneous, reconfigurable architectures with several SW and HW units. The goal is to obtain a methodology that does not necessarily give the optimum solution - but provides a possible solution without the risk of the prohibitively long exploration times given by high computational complexity.

1.4 Thesis Formulation

The thesis is formulated based on the questions formulated in Section 1.2. These can be summarized into two questions:

- What are the characteristics to be fulfilled in order for a reconfigurable implementation to be more feasible in terms of execution time and resource consumption than static implementation?
- Is it possible to extend and combine existing mapping methodologies to support heterogeneous architectures composed of several SW and HW units.

Based on these questions, we formulate the following theses and sub-clauses:

1. It is possible to investigate the feasibility of implementations in reconfigurable architectures based on a model describing reconfiguration and execution in coarse terms.
2. Multiprocessor scheduling algorithms for SW architectures can be extended and combined with temporal partitioning algorithms for reconfigurable architectures.
    a) Temporal partitioning can generate clusters of HW task
    b) The HW clusters and their reconfiguration can be scheduled by a multiprocessing scheduler for SW architectures, taking into account the structure and delay of communication.

We believe that in case both theses can be supported, this work is a useful tool for the designer in the design space exploration process and can shorten development time by providing the designer with feasible bindings and schedules for implementation.

1.5 Outline of the Dissertation

The remainder of this dissertation is organized as follows: First, the considerations behind the used methodology is described in Chapter 2. In Chapter 3 the summary of our contributions is given. This is followed by the conclusion and outlook in Chapter 4. Papers A-D (overview given in Chapter 5) are the contributions to the dissertation , and we refer to these papers by giving the letter in brackets, e.g. [A]. References to other resources are numbers in brackets, e.g. [1], referring to the list on page page 33.

2 Methodology

This chapter describes the considerations made during the development of the methodology used by the mapping framework. The chapter starts with the modeling of reconfigurable architectures, followed by the considerations of the mapping framework. Results are summarized in Chapter 3 and papers [A]-[D].

2.1 Modeling

The modeling performed in the work is split in two parts. The approach of the modeling is formulated such that application and architecture are orthogonalized, as suggested for several methodologies [51, 52]. This allows the independent modeling of the application and of the architecture, thereby enabling the use of the same application model for several architectures, and vice versa. The application and architecture model are related via a cost-estimate library, describing each alternative of implementation of a task or operation. The cost-estimate library is described in Section 2.2.

Application Modeling

The applications of concern are signal processing and communication processing applications. We have decided to consider static applications, where the operations and flow of the algorithms are known at compile-time. This limits the number of applications, however, it suits many algorithms inside the concerned domain of applications.

In order to handle adaptive applications (e.g. changing of modes in multi-mode and multi-standard radios), a consideration of the multiple use-cases analysis by Kumar et al. [53] could be considered. Use-cases are all uses of the application or product. These are analyzed to generate sets of functionalities resulting in a set of application graphs that should be implemented. Each application graph represents an implementation of one or more use-cases, and the switch between implementations may be performed by full reconfiguration of the system.

Our application model, as outlined in [D], models tasks as nodes and their dependencies as edges in a Directed Acyclic Graph (DAG), describing the precedence-relations between the tasks. Furthermore, we have added properties to the model to describe the amount of data transferred between nodes, so that communication delay can be modeled. Our application model allows the description of parallelism and dependencies, and can thereby describe a variety of signal processing applications. The nodes may have varying

granularity, which allows them to capture both fine-grain (down to bit-level) and coarse-grain (up to large chunks of functionality like FFT etc) tasks.

The precedence graphs are a subset of Synchronous Data Flow (SDF) graphs, describing precedence-relations and rate-specifications. The data-flow description is considered to be a natural selection as signal processing applications mostly consist of data-flow oriented algorithms. SDF allows the modeling of pipelined streaming and cyclic dependencies [54]. The cyclic dependencies allow modeling of digital Infinite Impulse Response (IIR) filters, but we assume that the full filter can be modeled as one node in the DAG. In [C] we have introduced a methodology to convert an application described as SDF into a directed acyclic precedence graph corresponding to our application model.

Some mapping approaches [55] have specified systems by Control Data Flow Graphs (CDFG) that allow the modeling of control path structures of the applications. However, we find that the DAG limitation is acceptable since the flexible granularity allows us to hide the control structure inside the node.

Architecture Modeling

Section 1.1 described how a heterogeneous, reconfigurable architecture is composed of one or more reconfigurable hardware units in connection with one or more SW processors. Therefore, the requirements to the architecture model is that it must be able to capture heterogeneity, granularity, and reconfiguration capabilities.

This resulted in the general architecture model which is shown in Figure 2.1. The architecture consists of a number of hardware and software units, connected via dedicated or shared buses.

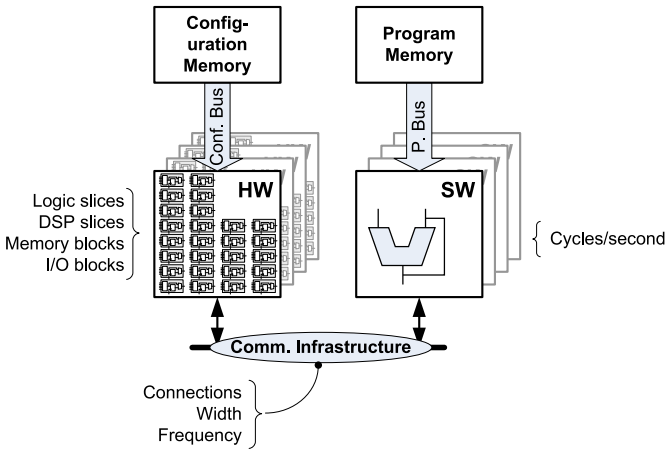


Figure 2.1: General architecture model for a heterogeneous reconfigurable architecture. The properties that are included in the model for each unit type are marked by {-brackets.

The model is based on the assumption that each hardware and software unit can execute in parallel. Inherent parallelism as such inside the SW units is not reflected in the general model. Communication between units is performed before the initiation and after the start of executions.

Execution in SW is assumed to be performed sequentially, requiring a certain execution time to perform the task. The execution of a task may not be interrupted by other tasks. The SW units may have external memory, which is not included in our abstract model. This means that estimated execution times may vary due to the fact that the data may be saved either in internal registers, cache, or external memory. However, this limitation is chosen since it reduces the complexity of the model. The simple architecture model allows the modeling of multiple SW processors as well as multiple cores. However, the abstraction level does not cover multiple datapaths inside a single core (like e.g. the Texas Instruments C6000 series DSP [56]). The use of the multiple datapaths is assumed to be reflected in the cost estimates as described in [D].

Execution in HW is assumed to be performed both in parallel and possibly sequentially in time, i.e. the reconfigurable HW can have parallel processing cores implemented, but reconfiguration does also allow those to be reconfigured during run-time. However, the input and output data of an executed context must be transported before and after execution as described in Figure 1.2a. The reconfiguration of the reconfigurable hardware is performed and controlled from an external reconfiguration controller with external memory. This controller is assumed to be inherent of the architecture and is abstracted away to be represented only by the reconfiguration delay. The effect of this abstract model is that the reconfiguration delay is independent of the availability of the configuration interface. However, this is not an issue for full reconfiguration or partial reconfiguration with only one reconfigurable region.

Our architecture model in Figure 2.1 does not include partial reconfiguration as such. Therefore, it can be argued that our model does not capture partial reconfigurability. However, the synthesis flow for generation of bitstreams for partial reconfiguration of Xilinx FPGAs does require an early decision on the placement and size of the reconfigurable region [57, 58], which is also illustrated in Figure 2.2. The modules that are reconfigured into these regions may have more parallel functionalities, and can be considered as fully reconfigurable partitions of the partially reconfigurable hardware. Thus our architecture model for heterogeneous, reconfigurable architectures does cover a design in partially reconfigurable FPGAs like the Xilinx Virtex series.

Feasibility Modeling

The feasibility of a reconfigurable implementation was defined based on a cost function. If the cost of a reconfigurable implementation is equal to or lower than a static implementation - the reconfigurable implementation is said to be *feasible*. The feasibility study is explained in [A].

In order to evaluate the feasibility, a cost function has been defined based on the product of area and time, $C = A \times T$. The area-time product has been selected as we consider these the main resources that are consumed during an implementation. The area, A is a measure of the resources in an architecture (for static and full dynamic reconfiguration), whereas, for partial dynamic reconfiguration, A is a measure of the resources consumed by a task. Time, T, is the execution time of the full application for a static implementation; execution and reconfiguration times for the dynamic reconfigurations.

Our model compared the full execution time, so that the same deadline was maintained for both a static and a (fully and partially) reconfigurable implementation. The approach was not similar to [59] which considers the maintaining of throughput and does

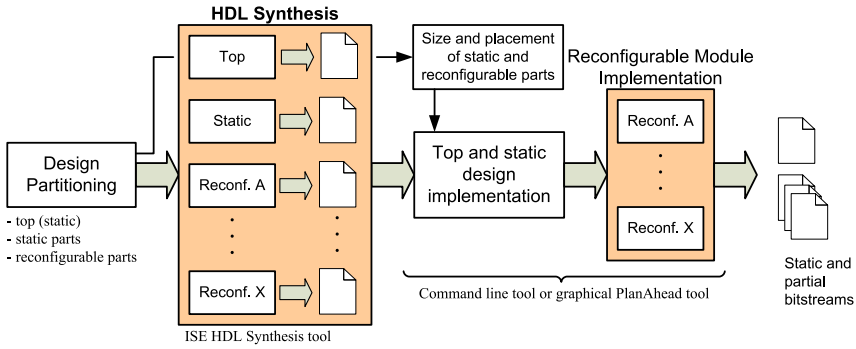


Figure 2.2: Illustration of the design flow of partial reconfiguration for Xilinx FPGAs based on [57], [58]. The design starts with a partitioning of the application into static and reconfigurable parts. The top-level contains the clock signals as well as relation (in terms of signals) between static and reconfigurable parts. Then each part is synthesized to obtain netlists for each module, using the Xilinx Integrated System Environment (ISE) synthesis tool and possibly the Embedded Developer's Kit (EDK). Then the size and placement of static and reconfigurable parts are decided (by the designer), and the top and static modules are implemented, followed by the implementation of each reconfigurable module. Finally, the full bitstream of the static module is generated, followed by the partial bitstreams.

not consider dynamic partial reconfiguration. Our cost function does not contain power consumption. However, a recent study by Becker et at. [4] concluded that feasibility in terms of energy consumption is related (not linearly) to the area consumption.

2.2 Mapping Framework

The framework was developed and is described in detail in [B] and [D]. The framework was developed for mapping static applications onto heterogeneous, reconfigurable architectures as described in Section 2.1. The outcome of the framework is the generation of a schedule that can then be implemented in the architecture. The schedule describes which tasks to perform in which unit - and when to execute them.

Mapping and scheduling for HW/SW architectures is a constrained optimization problem [3]. The constraints and costs are typically described by a cost function (as in (1.1)), which can be dependent on parameters such as execution time, area or resources (Configurable Logic Block (CLB), DSP blocks etc), power consumption, numerical accuracy etc. We formulate our constraints as *minimization of overall execution time (makespan) while satisfying the resource-constraints of the reconfigurable hardware*.

Another approach to the optimization problem could be to have a constraint on the execution time, and thus minimize the resource usage. However, we find that the first approach is the most useful since it suits the Xilinx development flow in Figure 2.2.

The outline of the mapping framework is given in Figure 2.3. For details and formalization, please consult [D]. The mapping framework takes as input the specification of the application, architecture, and their interrelation via a library of cost estimates as

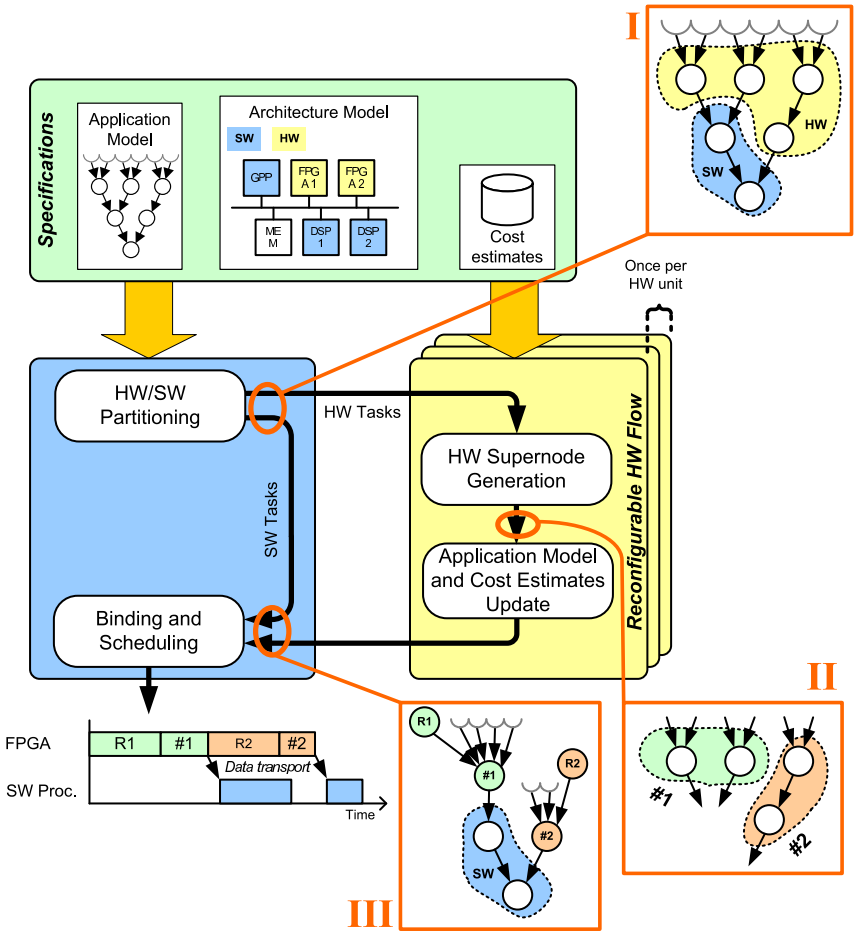


Figure 2.3: Outline of our developed mapping framework. The orange marked circles and boxes (I-III) are an example of task-graph transformations, utilizing only one SW and one HW processing unit.

briefly outlined in Section 2.1 and more detailed in [D]. First, the tasks are split into HW and SW tasks in the *HW/SW Partitioning* step, resulting in a partitioning as indicated in box I. The HW tasks for each HW-unit are treated in the *Reconfigurable HW Flow* where super-nodes of HW tasks are created in *HW Supernode Generation* by temporal partitioning. Each super-node corresponds to a full configuration of the reconfigurable area (as illustrated in box II), and these are updated in the application graph and cost-table. The *Reconfigurable HW Flow* is performed once for each reconfigurable HW unit. The *HW Supernode Generation* can be performed in parallel, but the *Application Model and Cost Estimates Update* have to be performed sequentially in order to maintain a graph with updated edges. The resulting updated application graph is illustrated in box III. The *binding and scheduling* is performed by a multiprocessor scheduling algorithm for heterogeneous architectures. The scheduling algorithm treats the HW super-nodes similar to SW tasks, but only for the reconfigurable unit. Furthermore, the algorithm must ensure that reconfiguration for one task is followed by that task, and not another task or reconfiguration. The final output of the framework is a schedule that describes which tasks should be packed into one configuration, and which unit should perform which tasks (and at which time).

The orthogonalization of application and architecture allows the evaluation of several architecture models for the same application, and thus we can explore several points in the design space. This was also performed in papers [C] and [D].

The cost-estimate library contains costs of binding alternatives for each task, which can also be noted ”design point”. Each task may have more design points, e.g. on different processing units. A design point for a HW implementation is a tuple of costs, including execution time, logic resource consumption, plus possibly embedded DSP and embedded memory consumption. Our modeling allows the HW implementation to have several design points for a single HW unit. This reflects the area-time trade-off that FPGA designers can make when performing their implementation.

The modeling and our mapping methodology are highly dependent on the ability of the designer to provide realistic cost-estimates for the tasks. This requires the designer to have some knowledge about the implementation before starting the actual implementation. It can be argued that exactly this requires the designer to have performed the implementation before starting the exploration using the proposed framework. However, we argue that most designs are based on previous designs or IP blocks - mainly utilizing the same architecture. This is also supported by the Embedded Market Study [11] which show that 57% of all projects are an update or improvement of earlier projects, with 86% code reuse. Furthermore, 48% of the projects reuse the same SW processor. Very often the designer does not need to synthesize everything, but only the building blocks that are not available from previous designs/IPs. Therefore, we believe that the requirement for cost-estimates is an acceptable workload for the designer.

In the following, we point to specific points in the mapping framework, related to Figure 2.3.

HW/SW Partitioning

The HW/SW partitioning decides (based on the implementation alternatives), whether tasks should be executed in SW or HW. The decision can be made from one of several schemes. 1) The simplest is the comparison of execution speed in HW and SW; if HW is fastest, the task is executed in HW. 2) Other factors could be resource consumption. If a task would take up a large portion of the resources in HW, it may be beneficial to place it in SW. 3) HW/SW communication is also known to take time, so it may be beneficial to execute predecessors and successors of a task in the same unit or processor. 4) Analyzing the inherent parallelism of the applications is also an option, as SW is typically considered sequential whereas HW is considered parallel.

However, even though the sole consideration based on execution time may lead to lower performance, it has been selected since it has been shown that considering execution time in the partitioning and even only implement a few of the tasks in HW lead to a significant speedup [48].

Banerjee et al. [44] showed that the best performance of the scheduling algorithm was obtained by considering partitioning and scheduling simultaneously. However, one of the goals of the methodology was to reduce the overall complexity, which is achieved by separation of partitioning and scheduling.

Reconfigurable HW Flow

The reconfigurable HW flow creates super-nodes of tasks based on temporal partitioning algorithms. These algorithms partition the tasks into clusters that each correspond to a full reconfiguration of the device. The temporal partitioning algorithms are generally list-based or the more computationally complex refinement based (like the ones described by Bobda [36]). We selected to extend the list-based algorithms by Purna and Bhatia [37] which had a low computational complexity. The algorithms were based on either level- or clustering-based temporal partitioning, which had different performance in terms of overall execution time and communication delay.

Our extension to the work is rooted in the fact that we have split HW and SW. Thus the temporal partitioning algorithm does not handle all tasks, which gives the risk of cyclic dependencies. We added a path search to the algorithms that ensured no precedence problems. For further details, see [D].

Binding and Scheduling

The binding and scheduling algorithm was subject to the requirement of being able to handle the following:

1. Heterogeneous architecture, consisting of
    a) Several SW processors
    b) Several reconfigurable HW units
2. Schedule reconfiguration for HW units correctly
3. Communication delay and the occupation of the communication resources

Our work is based on the Extended Dynamic Level Scheduling (EDLS) algorithm [60] that has a relatively low computational complexity while still taking into account the cost (in terms of time) of inter-processor communication. EDLS is an extension for heterogeneous architectures of the Dynamic Level Scheduling (DLS) algorithm.

The algorithm is further detailed in [60] and [D]. The algorithm computes a Dynamic Level (DL) value for each node-processor combination, and schedules the combination with the highest DL value. The algorithm has been modified such that in case a HW processor is configured for a specific task, the DL level is adjusted to a value such that only the combination with the corresponding execution will be chosen.

Since the modified EDLS algorithm maintains reconfiguration-execution relations, it is suitable for models where the usage of the configuration memory and interface is transparent. However, if more reconfigurable areas reside inside an FPGA, they share the same configuration port as illustrated in Figure 1.3. However, this can be modeled in our system by inserting the ICAP interface as a processing element, with corresponding reconfiguration nodes for this element.

The used methods in the developed framework allow us to handle static applications and map them onto heterogeneous reconfigurable architectures. Our approach has been to seek a computationally simple solution, and thereby keep the exploration time low. Thereby, we have implicitly accepted that our resulting schedule may not be the most optimal in terms of execution time and resource usage. However, we believe that the obtained schedule and the combination with several architecture models form a sufficient outset for the design space exploration of a heterogeneous reconfigurable implementation and thus allows rapid design space pruning. Furthermore, we assume that the solutions found by the mapping framework can actually be implemented in a heterogeneous reconfigurable architecture.

The used models for application and architecture modeling are general and allows flexible modeling that we believe is technology independent. The level of detail in the models is relatively low, which on the other hand eases the fitting of the model to the application and architecture by the designer.

3 Summary of Contributions

This chapter provides a summary of the contributions of this work. The summary is related to the papers A-D.

The body of this dissertation is formed by the papers A-D. First, an overview of the papers is given:

Paper A: In this paper we investigate the feasibility of reconfigurable architectures in order to evaluate whether reconfigurable architectures are more feasible than static architectures for signal processing applications. Case studies were performed for a Fast Fourier Transform (FFT) and a Digital Audio Broadcasting (DAB) receiver.

Paper B: Here, we have presented the idea of our mapping methodology alongside with evaluations of two partitioning algorithms and two multiprocessor scheduling algorithms by simulations on abstract application models and cost-entries.

Paper C: In this paper we have presented a method to obtain application graphs and cost-estimates for the mapping framework, based on a SystemC-AMS model of the application. The methodology is illustrated by a case study of a static implementation of a Bluetooth baseband unit in a Xilinx Virtex-5 FPGA.

Paper D: This paper contains a formalized description of the algorithms used by the mapping framework. We outline the simulations of using the mapping framework and perform a case-study based on the implementation of a Minimum Mean Square Error (MMSE) equalizer for a MIMO-system.

The papers treat the two topics from our thesis in Section 1.4, which are further described in the following sections:

3.1 Feasibility of Reconfigurable Implementations

In order to evaluate whether a static application could benefit from a reconfigurable implementation as opposed to a fully static FPGA implementation, we propose a model and cost function of the execution of applications in both static and reconfigurable FPGA implementations. Our work in this field is described in paper A.

The model is based on a split of execution into reconfiguration and execution, possibly with data-communication as described in Section 2.1 of the dissertation. Our contribution

is the proposed model based on execution time instead of throughput as performed by Wirthlin and Hutchings [59]. Thereby, our model allows the evaluation of time-sharing of the same area resources, as well as partial reconfiguration. Our cost function is based on an area-time product.

Our results (based on a case-study of an FFT core and a DAB receiver) show that the reconfiguration time for reconfiguring the full area is too high to make a reconfigurable implementation feasible. However, for applications where timing constraints are less stringent, a reconfigurable implementation may be beneficial. In the case of partial reconfiguration, the reconfiguration time is low enough to make the partially reconfigurable implementation feasible as compared to a static implementation.

In the paper we propose a similar feasibility study of the energy consumption aspects of full and partial reconfiguration. Since publication of our work, new contributions have been made by Becker et al. [4]. Although their study is related to energy aspects of reconfiguration, their model is also based on a split between execution and reconfiguration, and the conclusion about feasibility is similar to ours, i.e. it can be concluded that reconfiguration time is the limiting factor of the feasibility of reconfigurable implementation. They also conclude that in order to reduce the reconfiguration time as much as possible, the area should obviously be as small as possible.

3.2 Mapping Framework

The mapping framework is described in paper B and D. Paper B are the preliminary results of the mapping framework, indicating which temporal partitioning algorithm yields the lowest makespan. We also formulate a light version of the Integer-Linear-Programming (ILP) formulation of placement-aware scheduling in partially reconfigurable FPGAs [44] for comparison purpose. Our contribution in this connection is a placement-unaware model, as we only consider area and not placement for contemporary FPGA design. The results are based on an abstract application and cost model, and was evaluated for uniprocessor solutions.

Paper D contains the formal description of our developed framework. Furthermore, a case-study of an MMSE equalizer for a MIMO system [61, 62] is included to compare the resulting schedule of the framework with an actual implementation in a Xilinx Virtex-5 FPGA [63]. Our contribution is the formal description of the algorithms in the framework, especially the additions to the temporal partitioning algorithms [37] and their interplay with the Extended Dynamic Level Scheduling (EDLS) algorithm by Sih and Lee [60]. Our overall algorithm has a complexity of $O(P\ M\ N^3)$ for high M and N. This is lower than the iterative algorithm by Chatha and Vemuri [42] which has a relatively high complexity $O(N^4\ B + N^3\ B^2 + N^3\ B\ M)$ where N is the number of nodes in the task graph, M is the number of edges, and B is the maximum number of design points for each task.

Both the works in [B] and [D] are dependent on input from the designer in terms of application and architecture models, as well as a cost estimates library describing the cost of implementing each task. Paper C contains the description of a methodology to obtain the cost-estimates by the use of a VHDL synthesis tool. Furthermore, the resulting schedules of the framework are used in design-space exploration in order to select the most suitable architecture.

In paper C, we investigate a methodology to derive a directed acyclic application graph and corresponding cost-estimates from an executable description of an application. The executable description is a SystemC-AMS model describing the application by a synchronous data flow graph. The methodology is evaluated on a case-study of a Bluetooth baseband transmitter. The resulting graph and cost-estimates are fed to the mapping framework for a static architecture. Results show that by using the methodology in combination with the mapping framework, we are able to perform design space exploration and evaluate various architecture designs.

All contributions are targeted for use as parts of a tool for designers of an application implemented on a heterogeneous reconfigurable architecture, for which we assume the design trajectory in Figure 1.1. Both the feasibility estimation methodology and the mapping framework are tools for use in the *Design Space Exploration & Mapping* phase of the design, before the designer performs the actual implementation and test.

4 Conclusion

This chapter gives the conclusion of the work, supporting the theses. Furthermore, suggestions for future work are provided.

In this dissertation we investigate feasibility, modeling, and mapping methodologies for heterogeneous reconfigurable architectures. Our model of reconfigurable architectures allows us to describe the capabilities of a reconfigurable architecture, and is composed of 1) a pure model of the reconfigurable hardware for feasibility investigation, and 2) a model of a heterogeneous, reconfigurable architecture consisting of several reconfigurable hardware (in the form of FPGA) and software processing units. The model is input to a mapping framework for such architectures, together with an application model and a cost-library integrating the two. The mapping framework generates schedules for implementation of algorithms onto heterogeneous, reconfigurable architectures. The framework is a tool that may aid the designers in the *Design Space Exploration & Mapping* phase in the design trajectory illustrated in Figure 1.1.

In the following, we provide answers to the questions that lead to the formulated theses in Section 1.4.

1. *It is possible to investigate the feasibility of implementations in reconfigurable architectures based on a model describing reconfiguration and execution in coarse terms.*

We propose a model for execution in reconfigurable architectures [A]. Both full and partial reconfiguration are considered. The model reflects the major cost of a reconfigurable implementation, namely the reconfiguration time itself. Furthermore, the area cost of an on-chip reconfiguration controller is included in the cost-model. We conclude that the simple execution model allows realistic evaluation of feasibility, also supported by the work of Becker et al. [4] that propose a similar model for investigation of feasibility from an energy perspective.

Our conclusion on the case-studies is that reconfiguration time may in many cases be too high for a reconfigurable implementation to perform better (in terms of the area-time product) than a fully static implementation in an FPGA. This is also similar to the energy-study [4]. However, we can conclude that for applications with large execution time or applications which are massively parallel, the reconfiguration overhead may be less significant. Thus, we conclude that it is likely that multimedia (e.g. image and video) signal processing will show feasibility of implementations in reconfigurable hardware.

Furthermore, we only investigate static applications as we argue that it covers a large variety of signal processing applications. It is natural that reconfigurable implementations are feasible for adaptive applications, where some parts (filters, encoders etc.) are required to change due to a change in channel conditions, communication standards etc. [21].

2. a) *Temporal partitioning can generate clusters of HW task*

Based on our work in [B] and [D], we conclude that temporal partitioning is a useful approach to generate clusters of HW tasks that can be generated as configurations for the reconfigurable HW. However, since we work in a heterogeneous multiprocessing environment, we need to ensure that the precedence constraints between clusters are maintained. Therefore, it is necessary to add a PathSearch algorithm to avoid cyclic dependency problems. The temporal partitioning algorithm in the reconfigurable HW flow has a low computational complexity, which makes it a suitable algorithm to be used in the methodology.

2. b) *The HW clusters and their reconfiguration can be scheduled by a multiprocessing scheduler for SW architectures, taking into account the structure and delay of communication.*

In order to schedule reconfiguration and execution by a multiprocessing scheduler, we include the step *Application Model and Cost Estimates Update*. The step generates super-nodes (and corresponding cost-table entries) of the HW tasks (including their reconfiguration) to be inserted into the global task graph, handled by the multiprocessing scheduler.

The utilization of the Extended Dynamic Level Scheduling algorithm ensures the inclusion of the communication delay on the bus. However, changes are made to the algorithm to ensure that a reconfiguration-execution pair is not interrupted. Furthermore, the application model update includes a precedence update of edges between the reconfiguration nodes to ensure that precedence between these reconfiguration nodes is preserved.

2. *Multiprocessor scheduling algorithms for SW architectures can be extended and combined with temporal partitioning algorithms for reconfigurable architectures.*

The conclusion is that temporal partitioning can be used to generate the HW clusters that can be scheduled by a multiprocessing scheduler, if it takes the previously mentioned conditions into account. However, in our developed framework, we are heavily dependent on the pre-partitioning between HW and SW. Our partitioning is based only on a comparison of execution times for SW and reconfiguration and execution times for HW. The weight, α, of the reconfiguration was shown to provide best results for $\alpha = 1$. Other more complex schemes can be used, and this may influence the result of the work. The partitioning has significant influence on the performance of the mapping framework. Furthermore, our results showed that the level-based temporal partitioning algorithm resulted in the lowest overall execution time.

The framework uses as input an application and architecture model describing the tasks and the architecture at which they should be implemented. In order to keep the computational complexity low, our models have a relatively high level of abstraction.

This allows a simple overall specification of the application, and thereby specify e.g. nested loops inside a node. Furthermore, the large level of abstraction and granularity of tasks improves the quality of the estimates for each task.

The overall conclusion of the work is that we have suggested a mapping framework for handling application models consisting of directed acyclic precedence graphs. This allows us to model many types of signal processing applications, and we believe that the framework is a significant tool for the designer of signal processing applications for heterogeneous, reconfigurable systems.

The framework is relying on the designer to provide suitable and sufficiently accurate estimates of the costs of implementing each task. We believe that since many projects are built on previous projects or include already implemented parts, the workload is acceptable in comparison to the gain by using a mapping tool. In order to aid the designer in this process, our work in [C] is considered a useful methodology for obtaining the cost estimates. We believe that this work is a necessary step in developing an automated approach for mapping of algorithms onto reconfigurable architectures, which may significantly aid the designer in the design space exploration process.

During our work we have found that reconfiguration of reconfigurable fine-grain architectures has a significant impact on the feasibility of such implementations. We have developed a mapping framework that provides the designer with the possibility of evaluating the costs (area and time) of implementation alternatives as a pre-implementation step.

Our work has solely considered fine-grained reconfigurable hardware such as FPGAs. However, we believe that the abstraction of our modeling allows coarse-grained reconfigurable hardware as well as other SW processor types. The reason is that our model does not consider the exact structure of the reconfigurable fabric - but instead the capabilities of the fabric in terms of area and time. The area measure could be reconfigurable tiles (e.g. like the 4x4 array in a coarse-grained architecture) instead of large number of CLBs.

Although we believe that this work paves the way for improved feasibility analysis and mapping of applications onto heterogeneous, reconfigurable architectures, there is still room for improvement, as discussed in Section 4.1.

4.1 Future Work

As an inherent consequence of research, the path for future work may be manyfold. However, we suggest the work to have four parallel trajectories:

- Improve the modeling of the application to include flexible cases (or multiple functionalities), similar to the work by [53]. An outline of this suggestion in given in Figure 4.1. The work is used to generate potential implementations that are each fed to the mapping framework and thereafter implemented.

- In order to fully evaluate the feasibility of reconfiguration, we suggest to evaluate case-studies with massively parallel processing as opposed to the mainly sequential application that we have investigated. Such application may be in multimedia applications or multiple-functionality applications with fully parallel (as in time-exclusive) functionalities. We also propose to evaluate more architecture cases with several reconfigurable HW units in order to evaluate the full potential of the

mapping framework. As we have shown in the work [A] and [D], it is necessary to realistically consider the reconfiguration time.

- Improve the performance of the mapping framework by investigating whether employing other HW/SW partitioning algorithms can improve the overall performance of the mapping framework. One proposition is to be more "selective" as to which tasks to accelerate in HW, and thereby accelerate fewer tasks in HW. Another proposition is to include area considerations in the HW, e.g. based on a area-time product as in our feasibility model. A last proposition is to analyze the resulting schedule for the time/area consuming parts and possibly redo the partitioning based on another partitioning priority. This feedback from *Binding and Scheduling* to *HW/SW Partitioning* will increase the execution time significantly, but gives a better possibility to prune the design space.

- The mapping framework could be developed as part of an automated approach, combining and extending our work in [C] with the mapping framework [D] and thereby obtain an automated tool for design space exploration and binding and scheduling for heterogeneous reconfigurable architectures. The framework will serve as a front-end tool, used in the beginning of the design process. After having obtained the mapping, the back-end synthesis and compilation tools will be used with the input schedule as well as HW and SW code (e.g. VHDL and C, respectively) developed by the designer. Even though the work in [C] has been performed based on Xilinx synthesis and compiler tools, they can be replaced by any synthesis and compiler tools - as the methodology is not technology dependent. However, in case other input languages are used, it may require that the development of the process must be repeated depending on the commonality and compatibility between the languages.

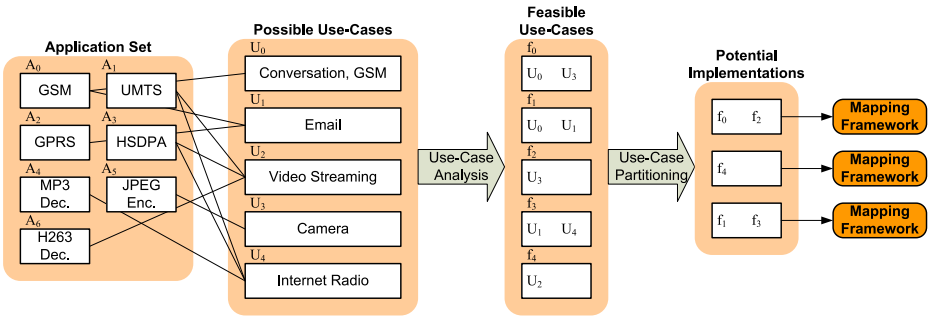


Figure 4.1: Proposed extension for multiple use-cases inspired by [53]. Each use-case is a composition of applications from the application set. The possible use-cases are analyzed to find the feasible use-cases, i.e. the use-cases that may be active at the same time. The feasible use-cases are partitioned with regard to the HW in order to find potential implementations. Each potential implementation may be fed to the mapping framework proposed in this dissertation.

5 List of Publications

The main body of this dissertation consists of the following publications.

Peer-Reviewed Conferences

[A] A. Popp, Y. Le Moullec, and P. Koch, ”Fast Feasibility Estimation of Reconfigurable Architectures”, *4th IEEE Conference on Industrial Electronics and Applications*, May 2009.

[B] A. Popp, Y. Le Moullec, and P. Koch, ”Scheduling Temporal Partitions in a Multiprocessing Paradigm for Reconfigurable Architectures”, *NASA/ESA International Conference on Adaptive Hardware Systems*, August 2009.

[C] A. Popp, A. Herrholz, K. Grüttner, Y. Le Moullec, P. Koch, and W. Nebel, ”SystemC-AMS SDF Model Synthesis for Exploration of Heterogeneous Architectures”, *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, April 2010.

Peer-Reviewed Journals

[D] A. Popp, C. Jégo, P. Koch, and Y. Le Moullec, ”A Mapping Framework for Heterogeneous Reconfigurable Architectures - Combining Temporal Partitioning and Multiprocessor Scheduling”, *International Journal on Reconfigurable Computing*, submitted for peer-review in June 2010.

Other Publications

In addition to the above, we have the following publications that have not been included in the main body of the dissertation, since they summarize part of the contents of the other papers:

- A. Popp, Y. Le Moullec, and P. Koch, ”Temporal Partitioning and Multi-Processor Scheduling for Reconfigurable Architectures”, Poster presentation at HiPEAC network summer school, *Advanced Computer Architecture and Compilation for Embedded Systems*, June 2008.
- A. Popp, Y. Le Moullec, and B. Olech, ”Designing Heterogeneous Reconfigurable Systems : Feasibility Analysis, Temporal Partitioning and Multi-Processor Scheduling”, *Electronics -Constructions, Technologies, Applications*, 12th issue, 2009.

References

[1] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective," in *Proceedings on Design, Automation and Test in Europe*, March 2001, pp. 642–649.

[2] M. Gokhale and P. S. Graham, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, 2005.

[3] P. Arató, Z. Ádám Mann, and A. Orbán, "Algorithmic aspects of hardware/software partitioning," *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, pp. 136–156, 2005.

[4] T. Becker, W. Luk, and P. Y. K. Cheung, "Energy-aware optimisation for run-time reconfiguration," in *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 55–62.

[5] J. M. III, "Software radios: survey, critical evaluation and future directions," *IEEE Aerospace and Electronic Systems Magazine*, vol. 8, no. 4, pp. 25–36, April 1993.

[6] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, June 2002.

[7] R. Hartenstein and T. Kaiserslautern, *Designing Embedded Processors - A Low Power Perspective*. Springer, 2007, ch. Basics of Reconfigurable Computing, pp. 451–501.

[8] G. Estrin, "Reconfigurable computer origins: The ucla fixed-plus-variable (f+v) structure computer," *IEEE Annals of the History of Computing*, vol. 24, no. 4, pp. 3–9, 2002.

[9] J. Hauser and J. Wawrzynek, "Garp: a mips processor with a reconfigurable coprocessor," in *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, Apr 1997, pp. 12–21.

[10] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer, "Piperench: a coprocessor for streaming multimedia acceleration," in *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999, pp. 28–39.

[11] EE Times Group, "Embedded market study 2010," May 2010.

[12] *Virtex-5 FPGA Configuration User Guide (UG191)*, Xilinx Inc., June 2009.

[13] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207, March 2005.

[14] A. Shoa and S. Shirani, "Run-time reconfigurable systems for digital signal processing applications: a survey," *Journal of VLSI Signal Processing Systems*, vol. 39, no. 3, pp. 213–235, 2005.

[15] K. Tatas, K. Siozios, and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing*. Springer, 2008, ch. A Survey of Existing Fine-Grain Reconfigurable Architectures and CAD tools, pp. 3–87.

[16] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, February 2007.

[17] J. Helmschmidt, E. Schüler, P. Rao, S. Rossi, S. di Matteo, and R. Bonitz, "Reconfigurable signal processing in wireless terminals," in *IEEE Proceedings of the Design,Automation and Test in Europe Conference and Exhibition*, 2003, pp. 244–249.

[18] J. P. Delahaye, G. Gogniat, C. Roland, and P. Bomel, "Software radio and dynamic reconfiguration on a dsp/fpga platform," in *3rd Karlsruhe Workshop on Software Radios*, 2004.

[19] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto, "A self-reconfigurable implementation of the jpeg encoder," in *IEEE International Conf. on Application-specific Systems, Architectures and Processors*, July 2007, pp. 24–29.

[20] H. Wang, J.-P. Delahaye, P. Leray, and J. Palicot, "Managing dynamic reconfiguration on mimo decoder," in *IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[21] P. Manet, D. Maufroid, L. Tosi, G. Gailliard, O. Mulertt, M. D. Ciano, J.-D. Legat, D. Aulagnier, C. Gamrat, R. Liberati, V. L. Barba, P. Cuvelier, B. Rousseau, , and P. Gelineau, "An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications," *EURASIP Journal on Embedded Systems*, vol. 2008, pp. 1–11, 2008.

[22] J. Becker, M. Hübner, and M. Ullmann, "Power estimation and power measurement of xilinx virtex fpgas: trade-offs and limitations," in *16th Symposium on Integrated Circuits and Systems Design, 2003*, 2003.

[23] P. Jamieson, T. Becker, W. Luk, P. Cheung, T. Rissa, and T. Pitkänen, "Benchmarking reconfigurable architectures in the mobile domain," in *IEEE Symposium on Field Programmable Custom Computing Machines*, 5-7 2009, pp. 131–138.

[24] J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka, "Dynamic and partial fpga exploitation," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 438–452, February 2007.

[25] R. Hartenstein, "Trends in reconfigurable logic and reconfigurable computing," in *9th International Conference on Electronics, Circuits and Systems*, vol. 2, September 2002, pp. 801–808.

[26] M. J. Myjak and J. G. Delgado-Frias, "Medium-grain cells for reconfigurable dsp hardware," *IEEE Transactions on Circuits and Systems*, vol. 54, no. 6, pp. 1255–1265, June 2007.

[27] Z. Li, K. Compton, and S. Hauck, "Configuration caching management techniques for reconfigurable computing," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2000, pp. 22–36.

[28] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed fpga," in *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997, pp. 22–28.

[29] "Tabula, inc." [Online]. Available: http://www.tabula.com

[30] S. Hauck, Z. Li, and E. Schwabe, "Configuration compression for the xilinx xc6200 fpga," in *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998, pp. 138–146.

[31] Z. Li and S. Hauck, "Configuration compression for virtex fpgas," in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001, pp. 147 – 159.

[32] A. Dandalis and V. Prasanna, "Configuration compression for fpga-based embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 12, pp. 1394–1398, December 2005.

[33] S. Hauck, "Configuration prefetch for single context reconfigurable coprocessors," in *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 1998, pp. 65–74.

[34] Y. Qu, J.-P. Soininen, and J. Nurmi, "Using multiple configuration controllers to reduce the reconfiguration overhead," in *NORCHIP Conference*, 21-22 2005, pp. 86–89.

[35] F. Dittmann and M. Götz, "Applying single processor algorithms to schedule tasks on reconfigurable devices respecting reconfiguration times," in *20th International Parallel and Distributed Processing Symposium*, April 2006.

[36] C. Bobda, "Synthesis of dataflow graphs for reconfigurable systems using temporal partitioning and temporal placement," Doctor's dissertation, Faculty of Computer Science, Electrical Engineering and Mathematics of the University of Paderborn, May 2003.

[37] K. M. G. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *Transactions on Computers*, vol. 48, no. 6, pp. 579–590, June 1999.

[38] J. Kim, J. Cho, and T. G. Kim, "Temporal partitioning to amortize reconfiguration overhead for dynamically reconfigurable architectures," *IEICE Transactions on Information and Systems*, vol. E90, no. 12, pp. 1977–1985, December 2007.

[39] S. Ganesan and R. Vemuri, "An integrated temporal partitioning and partial reconfiguration technique for design latency improvement," in *Design, Automation and Test in Europe Conference and Exhibition*, 2000, pp. 320–325.

[40] J. Clemente, C. Gonzalez, J. Resano, and D. Mozos, "A hardware task-graph scheduler for reconfigurable multi-tasking systems," in *International Conference on Reconfigurable Computing and FPGAs*, 2008, pp. 79–84.

[41] J. A. Clemente, J. Resano, C. Gonzalez, and D. Mozos, "A hardware implementation of a run-time scheduler for reconfigurable systems," to be published in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1 –14, 2010.

[42] K. S. Chatha and R. Vemuri, "An iterative algorithm for hardware-software partitioning, hardware design space exploration and scheduling," *Design Automation for Embedded Systems*, vol. 5, pp. 281–293, 2000.

[43] M. D. Galanis, G. Dimitroulakos, and C. E. Goutis, "Partitioning methodology for heterogeneous reconfigurable functional units," *The Journal of Supercomputing*, vol. 38, no. 1, pp. 17–34, October 2006.

[44] S. Banerjee, E. Bozorgzadeh, and N. D. Dutt, "Integrating physical constraints in hw-sw partitioning for architectures with partial dynamic reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 11, pp. 1189–1202, November 2006.

[45] F. Redaelli, M. D. Santambrogio, and D. Sciuto, "Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems," in *Design, Automation and Test in Europe*, 2008.

[46] R. Cordone, F. Redaelli, M. A. Redaelli, M. D. Santambrogio, and D. Sciuto, "Partitioning and scheduling of task graphs on partially dynamically reconfigurable fpgas," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, May 2009.

[47] B. Miramond and J.-M. Delosme, "Design space exploration for dynamically reconfigurable architectures," in *Design, Automation and Test in Europe*, vol. 1, March 2005, pp. 366–371.

[48] G. Stitt, F. Vahid, and S. Nematbakhsh, "Energy savings and speedups from partitioning critical software loops to hardware in embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 1, pp. 218–232, February 2004.

[49] J. Noguera and R. M. Badia, "A hw/sw partitioning algorithm for dynamically reconfigurable architectures," in *Proceeding of Design, Automation and Test in Europe*, March 2001, pp. 729–734.

[50] F. Dittmann, M. Götz, and A. Rettberg, "Model and methodology for the synthesis of heterogeneous and partially reconfigurable systems," in *IEEE Parallel and Distributed Processing Symposium*, March 2007, pp. 1–8.

[51] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, July 1997, pp. 338 – 349.

[52] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1523–1543, December 2000.

[53] A. Kumar, S. Fernando, Y. Ha, B. Mesman, and H. Corporaal, "Multiprocessor systems synthesis for multiple use-cases of multiple applications on fpga," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 3, pp. 1–27, July 2008.

[54] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[55] Y. Le Moullec, J.-P. Diguet, and J.-L. Philippe, "Design-trotter: a multimedia embedded systems design space exploration tool," in *IEEE Workshop on Multimedia Signal Processing*, December 2002, pp. 448–451.

[56] *TMS320C6000 Programmer's Guide*, rev. J, Texas Instruments Inc., 2010.

[57] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas," in *International Conference on Field Programmable Logic and Applications*, 2006.

[58] *Partial Reconfiguration User Guide (UG702)*, Xilinx Inc., May 2010.

[59] M. J. Wirthlin and B. L. Hutchings, "Improving functional density using run-time circuit reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 2, pp. 247–256, June 1998.

[60] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175–187, February 1993.

[61] D. Karakolah, C. Jégo, C. Langlais, and M. Jezequel, "Architecture dedicated to the mmse equalizer of iterative receiver for linearly precoded mimo systems," in *3rd International Conference on Information and Communication Technologies: From Theory to Applications*, April 2008, pp. 1–6.

[62] D. Karakolah, C. Jégo, C. Langlais, and M. Jezequel, "Design of an iterative receiver for linearly precoded mimo systems," in *IEEE International Symposium on Circuits and Systems*, May 2009, pp. 597–600.

[63] *Virtex-5 FPGA User Guide (UG190)*, Xilinx Inc., May 2008.

Contributions

Paper A: Fast Feasibility Estimation of Reconfigurable Architectures

Paper B: Scheduling Temporal Partitions in a Multiprocessing Paradigm for Reconfigurable Architectures

Paper C: SystemC-AMS SDF Model Synthesis for Exploration of Heterogeneous Architectures

Paper D: A Mapping Framework for Heterogeneous Reconfigurable Architectures - Combining Temporal Partitioning and Multiprocessor Scheduling

---

Paper A

Fast Feasibility Estimation of Reconfigurable Architectures

Andreas Popp, Yannick Le Moullec, and Peter Koch

This paper was published in:
4th IEEE Conference on Industrial Electronics and Applications

Abstract

Reconfigurable architectures are often said to be able to exploit the possibilities of resource savings by means of hardware time-sharing. However, existing literature does not point clearly at which conditions must be fulfilled for considering a reconfigurable architecture for the implementation of signal processing applications. Therefore, we propose a fast method to perform high-level pre-implementation feasibility-based evaluation of a reconfigurable hardware implementation. The method is based on a light architectural model to compute costs of a static reference as well as costs for globally and partially reconfigurable architectures. Two case studies have been performed for an FFT and an FPGA-based DAB application. Our results show that implementation on reconfigurable architectures is only feasible when the reconfiguration time is low, which generally means that a dynamically partially reconfigurable solution is preferred.

1 Introduction

Reconfigurable hardware architectures have been introduced as a possibility to provide an intermediate solution between Application Specific Integrated Circuits (ASIC) or Application Specific Instruction-set Processors (ASIP) and Digital Signal Processors (DSP) [1]. Reconfigurable hardware is known to offer the opportunity of resource and energy savings for some applications due to the possibility of time-sharing of the hardware resources, as well as run-time circuit specialization allowing an accelerator that is ultimately customized to the task executing at any given moment of operation.

One of the most utilized reconfigurable architectures is the Field Programmable Gate Array (FPGA), where an example is the Xilinx Virtex series with the improved version of Dynamic Partial Reconfiguration (DPR) [2] in the newest Virtex-4 and 5 series. In DPR, also noted *partial reconfiguration* in the rest of this text, parts of the logic can be reconfigured while maintaining operation on the other parts. The application of the inherent flexibility of DPR has been demonstrated especially in the field of Software Defined Radio (SDR), among others by Delahaye et al. [3] and Ihmig et al. [4] where DPR allows the implementation of several functionalities without having to perform parallel implementations of all functionalities. Furthermore, extensive research efforts in both academia and industry have been put into *i)* synthesis tools and methods to perform scheduling of algorithms onto reconfigurable architectures by e.g. Bobda [5], and *ii)* technical solutions to reduce the reconfiguration overhead suggested by e.g. Hauck [6].

However, even though the use of reconfiguration in FPGA architectures seems promising, it is important for the designer to realize and remember that it is associated with certain costs to provide and use reconfiguration capabilities. Firstly, reconfiguration takes time (known as reconfiguration overhead) and consumes power. Secondly, reconfigurable architectures generally also consume more power, area, and have longer execution time than non-reconfigurable or static solutions. Finally, development time is also longer than for non-reconfigurable architectures, as reconfigurable hardware requires the developer to spend more time on design as well as test and debugging of the implementation.

Shoa & Shirani [8] have given a survey on reconfigurable systems in the context of digital signal processing operations. One conclusion is that FPGA implementation is suitable for data-intensive operations like FIR-filters, FFT and DCT transforms. In traditional

FPGA implementations the reconfiguration capabilities are not utilized. However, the inherent lack of flexibility during run-time is a motivation for considering reconfigurable architectures. The survey concludes that reconfigurable architectures should be considered due to possibilities of run-time circuit specialization and logic resource savings by time-sharing among hardware resources.

Although many applications of reconfigurable architectures based on FPGAs have been built, there is, to the best of our knowledge, a lack of clear pointers in the direction of determining when a reconfigurable implementation is feasible. In this paper, feasibility is defined as a non-reduction in performance as compared to a static implementation. Thereby, the study does not include the implementation effort in terms of development costs. The ability to derive a pre-implementation estimate before conducting the final implementation is considered an important basis for deciding whether it is worth the man-hours to perform the implementation in reconfigurable hardware architectures versus non-reconfigurable hardware. Therefore, we have posed the question:

"What high-level characteristics of the application must be fulfilled, and in which conditions is it feasible to make an implementation using reconfigurable hardware?"

Previous approaches to answer similar questions have mainly been focused on developing a full implementation and comparing it to another implementation in static hardware or programmable processors. Typically, solutions are compared by means of a cost-function or metric that weighs time, silicon area or resource usage, energy or power consumption, and other factors such as numerical properties. Such a cost-metric is used by Wirthlin & Hutchings [7], who provide an estimation method to evaluate the feasibility of a reconfigurable implementation. The evaluation is based on functional density, D, which is a throughput oriented cost metric including area, A, and total operation time, T, and combining these by the expression $D = \frac{1}{A\,T}$. Feasibility is determined from

$$I_{\max} \geq f \quad ,$$

where $I_{\max} = \frac{D_{\text{reconfigurable}}}{D_{\text{static}}} - 1$ is the improvement in functional density over a static implementation and f is the configuration ratio defined as the relation between total time spent on configuration and the total time spent on execution. This means that in case the area A is reduced by a factor of two, a two-fold increase in execution time, T, gives exactly the same functional density, D. This leads to an improvement $I_{\max}$ of 0%, thus if any time is spent on reconfiguration, f will become greater than 0, and a reconfigurable solution is deemed infeasible despite that the static and reconfigurable solution have equal functional density D. Similarly, if execution time is only increased by a factor of 1.5, then as long as 33% or less of the execution time is spent on reconfiguration, the throughput will not be degraded compared to the static reference. While the work evaluates feasibility of reconfigurable architectures, it has two limitations:

- The throughput oriented metric does not reflect the possibility of time-sharing resources and thereby reduction of the area-costs.

- Partial reconfiguration cannot be evaluated, as DPR is not directly reflected in the configuration ratio.

Manet et al. [9] evaluated dynamic partial reconfiguration for non-consumer applications based on selected scenarios where DPR could be advantageous. The evaluation shows that DPR has clear advantages when changes occur in environment or functions (denoted ”mission change”). Furthermore, advantages are shown by the use of hardware time-sharing to obtain hardware resource reduction. However, the evaluation of the advantages of DPR is subjective and an objective measure is desired.

Contribution

In this work we develop a light architectural model for globally and partially dynamically reconfigurable architectures. The model describes high-abstraction level characteristics of the architecture. The characteristics are considered adjustable to the architecture under consideration. The feasibility estimation method consists of two subsequent steps:

1. **Analysis** of the application from a high level of abstraction to determine execution patterns for the reconfigurable architecture.

2. **Logic synthesis** of parts or modules to estimate costs. The costs can also be estimated based on a cost-library for basic functions. The estimates are input to the architectural model to evaluate the feasibility by means of a *cost-function*.

The focus of the cost-function is put on the time and area trade-off that is made possible by time-sharing of resources and not on the flexibility that is provided for applications. Area is counted based on logic resources and the costs of software processors or controllers are not considered.

In the following, the method is presented. This is followed by two case-studies and presentation of the result of these studies. Finally, the results are discussed followed by the conclusion.

2 Method

The method consists of a conveniently light architectural model to describe the characteristics of the architecture. This is followed by a description of the application analysis.

Architectural Model

The architectural model is limited to consider a reconfigurable unit, a controller with configuration memory, and external memory as depicted in figure 1.

The proposed architectural model is the basis of two cost models, describing globally reconfigurable architectures and partially reconfigurable architectures. The models describe the capabilities of the architectures from a high-level point of view and capture time and resource parameters. Time costs are categorized on the basis of time spent on execution/computation, reconfiguration, or data transfer. There are many possible area parameters for quantifying resource costs, such as Configurable Logic Block (CLB) and DSP slices, reconfiguration resources, and RAM/memory resources for data, configuration and intermediate data representation. However, in this work area resources are only counted in CLB slices allocated for execution, based on the results from logic synthesis.

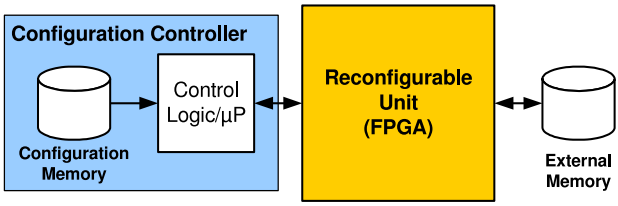


Figure 1: The basic reconfigurable architecture. The controller can be on-chip or off-chip, but the control resources are separated from the computational resources. The external memory is used to save intermediate data that is used in subsequent configurations.

The improvement or degradation caused by a reconfigurable implementation is based on a comparison to a static implementation. The static costs, C_{static}, are expressed by

$$C_{\text{static}} = A_{\text{static}} \cdot T_{\text{static}} \quad [\text{s} \cdot \text{slices}] \quad , \tag{1}$$

where A_{static} is the total area in CLB slices of the architecture, and T_{static} is the total execution time in seconds. The area is inherently two-dimensional thus the costs are conveniently described in three dimensions.

In our proposed cost model, time and area are given equal weight in order to fully reflect the area-time trade-off in time-sharing of resources. In case certain area or time constraints must be fulfilled, these constraints are evaluated externally to the cost evaluation.

Dynamic Global Reconfiguration

In the case of dynamic global reconfiguration, it is assumed that reconfiguration and execution cannot be overlapped, which is a general assumption for globally reconfigurable FPGAs. The execution flow is illustrated in figure 2 and proceeds as follows: First, the controller configures the FPGA. Then the FPGA executes the tasks of configuration 1 and stores intermediate data in the external memory. Then configuration 2 is programmed into the FPGA, followed by reading the intermediate data from memory. This process repeats itself until all configurations have been executed.

Time costs can easily be described by the sum in (2) that describes time-consuming parts of execution in a globally reconfigurable system:

$$\begin{aligned} T_{\text{exec}} &= \sum_i t_{\text{exec},i} \\ T_{\text{reconf}} &= \sum_i t_{\text{reconf},i} \\ T_{\text{transfer}} &= \sum_i t_{\text{read},i} + \sum_i t_{\text{write},i} \\ T_{\text{global}} &= T_{\text{exec}} + T_{\text{reconf}} + T_{\text{transfer}} \quad , \end{aligned} \tag{2}$$

where the symbols are defined as in table 1.

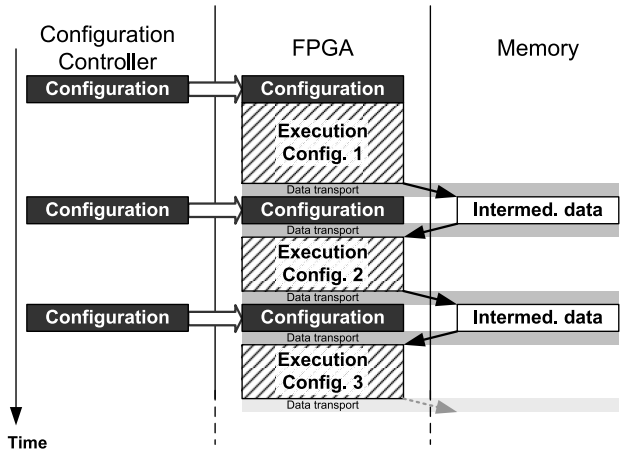


Figure 2: Execution flow in global reconfiguration.

Table 1: Definition of symbols in (2).

I	Total number of configurations
i	Configuration index, $i \in \{1, 2, \ldots, I\}$
T_{global}	Total time spent in the global reconfiguration scenario [s]
T_{exec}	Total time spent on execution [s]
T_{reconf}	Total time spent on reconfiguration [s]
T_{transfer}	Total time spent on data transfer [s]
$t_{\text{exec},i}$	Execution time of configuration i [s]
$t_{\text{reconf},i}$	Reconfiguration time of configuration i [s]
$t_{\text{read},i}$	Memory read-time for input to configuration i [s]
$t_{\text{write},i}$	Memory write-time for output from configuration i [s]

The total cost of the globally reconfigurable solution is given by multiplying equation (2) by the area in CLB slices, A_{global}, of the globally reconfigurable architecture:

$$C_{\text{global}} = A_{\text{global}} \cdot T_{\text{global}} \quad [\text{s} \cdot \text{slices}] \quad , \tag{3}$$

which can then be compared to C_{static}, (1).

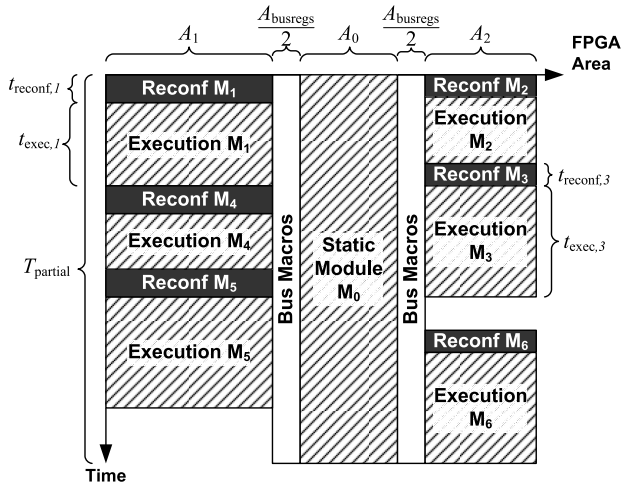


Figure 3: Execution flow in dynamic partial reconfiguration.

Dynamic Partial Reconfiguration

The partial reconfiguration model is basically similar to the model for global reconfiguration. However, instead of multiplying the time and total area for global configurations, the sum of reconfiguration and execution time is multiplied by the resources consumed by each reconfigurable module.

The model assumes that transfer of data between modules is performed by special bus registers, so called Bus Macros in Xilinx tool flows [2], and the transfer delay across Bus Macros is assumed negligible. However, the bus registers consume area during the whole operation. Furthermore, the placement of bus macros is assumed fixed during operation, as this is similar to current DPR implementation in Xilinx FPGAs [2]. The execution flow is illustrated in figure 3. The figure has one static module, M_0, that is active during the whole execution T_{partial}. There are two bus macros that handle communication of data between the reconfigurable modules and the static module. The configuration of the static module and the bus macros is not included in the costs, as it is assumed being a part of the general start-up of the FPGA. The six static modules M_1-M_6 are reconfigured prior to their execution. As indicated in the figure, there are periods where some of the resources are unused for execution. This is not included in the costs, as the area is theoretically available for other functionalities.

Table 2: Definition of symbols in (4), also illustrated in figure 3.

j	Module index
A_j	Area of module j [slices]
A_{busregs}	Area of the bus registers [slices]
$t_{\text{exec},j}$	Execution time of module j [s]
$t_{\text{reconf},j}$	Reconfiguration time for loading module j [s]
T_{partial}	Total execution time [s]
$C_{\text{partial,proc}}$	Cost of processing and reconfig. in DPR [s·slices]
$C_{\text{partial,comm}}$	Cost of communication in DPR [s·slices]
C_{partial}	Total cost of dynamic partial reconfiguration [s·slices]

The total cost of a partially reconfigurable implementation, C_{partial}, is expressed by

$$\begin{aligned} C_{\text{partial}} &= C_{\text{partial,proc}} + C_{\text{partial,comm}} \quad [\text{s} \cdot \text{slices}] \\ C_{\text{partial,proc}} &= \sum_j A_j \cdot t_{\text{exec},j} + \sum_j A_j \cdot t_{\text{reconf},j} \\ C_{\text{partial,comm}} &= A_{\text{busregs}} \cdot T_{\text{partial}} \quad , \end{aligned} \tag{4}$$

where the symbols are defined as in table 2. C_{partial} can be compared to C_{static}, (1), as well as C_{global}, (3).

Application Analysis and Logic Synthesis

The application analysis is performed by an examination of the application to demonstrate how to extract the parameters of the architecture model described in the previous section.

From a high level of abstraction the application and specifications are analyzed to determine the deadline, T_{deadline}, at which the task-set, (i.e. all operations), must be finished. The task-set can either be a one-time running application or periodic tasks. For periodic tasks, the deadline is equal to the longest period of the tasks. Since the static reference occupies all resources from execution start to deadline, T_{static} is set to T_{deadline}.

In the second part of the analysis, the application is examined to determine whether it can be divided into configurations that can be executed sequentially thus suitable for global reconfiguration. It may, however, be that it is judged more suitable for partial reconfiguration, and tasks are then grouped or defined by modules. The process can either be performed manually, or by an automated scheduling approach including temporal partitioning and placement similar to Bobda [5]. The latter does however, require knowledge or estimates of execution time and resource usage. Those estimates have to be acquired by logic synthesis, as described in the next paragraph.

The determined configurations or modules are provided as input to a synthesis program to obtain estimates of execution time and area consumption. The reconfiguration time, $t_{\text{reconf},i}$, is estimated by dividing the bitstream size estimate by the speed of the configuration interface (up to 100 MHz using the Xilinx Virtex-4 SelectMAP interface [2]) as in

$$t_{\text{reconf},i} = \frac{W + 1312}{100} \quad [\mu\text{s}] \quad , \tag{5}$$

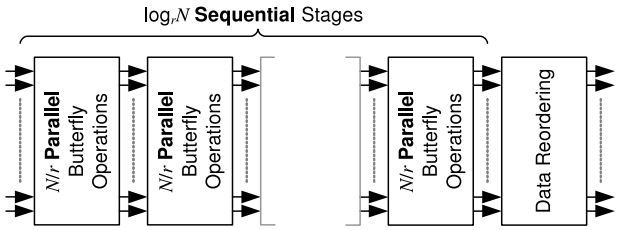


Figure 4: Organization of an FFT.

where W is the configuration array size of 147600, 726520, and 426810 for the LX15, LX80, and SX35 Virtex-4 FPGAs respectively [11]. In a similar manner, the data transferred between the configurations are quantified and divided by the read/write speeds of the external memory.

Finally, the costs are calculated, and the use of reconfigurable architectures is deemed feasible if the conditions (6) and (7) are satisfied. The left hand side arguments in curly braces indicate that only one argument is considered at a time; This is determined by the selection of global or partial reconfiguration:

$$\{C_{\text{global}}, C_{\text{partial}}\} \leq C_{\text{static}} \quad \text{AND} \tag{6}$$

$$\{T_{\text{global}}, T_{\text{partial}}\} \leq T_{\text{deadline}} \ , \tag{7}$$

which ensures that the total cost is lower than or similar to the static implementation, and that the deadline is fulfilled.

In case T_{global} or T_{partial} are lower than T_{deadline}, it may be considered to utilize reconfiguration capabilities even further i.e. trade off execution time for area reduction, or select an architecture with a lower clock speed as idle resources are available.

3 Case Studies

The previous sections described our proposed architecture model and how to do the application analysis. This is demonstrated by two case-studies in this section. The first study considers global reconfiguration, whereas the second study considers both global and partial reconfiguration.

Fast Fourier Transform

The Fast Fourier Transform (FFT) algorithm is widely used in multimedia applications and communications systems. In the latter case it is known as an efficient implementation of orthogonal frequency-division multiplexing (OFDM). An FFT is composed of parallel butterfly operation blocks that are executed sequentially followed by data reordering as illustrated in figure 4. N is the number of points in the FFT and r is the radix of the butterfly operations. The computation consists of $\log_r N$ sequential blocks of $\frac{N}{r}$ parallel radix-r butterfly operations.

The case is selected to be a 32 point radix-2 FFT ($N = 32$ and $r = 2$) operating at 16 bit resolution. The static reference is a fully parallel implementation with constant twiddle-factor multipliers synthesized for a Virtex-4LX80 FPGA with 35840 CLB

Table 3: Details of Configurations for the FPGA-based DAB Receiver [4]:

Configuration i	$t_{\mathrm{exec},i}$ [ms]	Content
0	2.26	Mixer, FIR filter, and fine frequency offset correction
1	1.14	Fast Fourier Transform
2	0.48	Coarse freq. offset correction, demodulator, frequency and time deinterleaving
3	0.11	Viterbi decoding and energy dispersion

slices [10] executing an FFT-operation at a rate selected to be 1 kHz. The reordering of data at the output is not considered for the static reference.

The alternative implementation is a globally reconfigurable solution at which each stage is implemented as a full configuration, theoretically making it possible to reduce the hardware area by a factor of $\log_r N = 5$. Reconfiguration time is estimated by (5) for a Virtex-4LX15 FPGA containing totally 6144 CLB slices [10], as the number of CLB-slices in this FPGA is close to 5 times smaller than the Virtex4-LX80.

The memory read and write times are estimated based on SDRAM memory running at 266 MHz, by using the expression

$$t_{read} = t_{write} = \frac{n_{\mathrm{bytes}}}{4\mathrm{bytes} \times 266\mathrm{MHz}} + \frac{3}{266\mathrm{MHz}} \quad , \tag{8}$$

where the last part of the sum is based on the latency of the memory. In this case study, the transferred amount of data, n_{bytes}, was 128 Bytes.

The static and globally reconfigurable implementations were synthesized in Xilinx ISE 9.1 based on VHDL code to obtain the necessary estimates.

FPGA-based DAB Receiver

The second case is based on a study of the results by Ihmig et al. [4]. The work consists of a digital audio broadcasting (DAB) receiver that is investigated for combining the tasks in a sequential execution on a Xilinx Virtex-4 SX 35 FPGA. The reference is a pipelined architecture consisting of 10 stages running at multiple rates, and is characterized by having a very relaxed latency requirement. The authors investigate a solution where the 10 stages are partitioned into four configurations, listed in table 3, that are executed sequentially at a higher clock frequency (100 MHz) than the pipelined architecture (8.2 MHz).

The buffered sequential implementation is assumed based on a 50 Hz cycle, which determines the time, T_{static}, of the static implementation. In their work, the read/write time for external memory is not listed, and is therefore estimated as in (8). The transferred amount of data between configurations, n_{bytes}, is conservatively assumed based on the maximum data rate of 8192 kbytes/s, which gives 8192/50 kbytes between each configuration.

The static and global area were both determined by the size of the FPGA to 15360 CLB slices [10] and the reconfiguration time was estimated as described in (5).

Table 4: Results: Fast Fourier Transform:

Static Implementation (Virtex-4LX80)		
Time	$T_{\text{static}} = T_{\text{deadline}}$	1 ms
Area	$\{A_{\text{static}}$,synthesis$\}$	{35840, 24516} slices
Cost	C_{static}	35.8 s·slices
Globally Reconfigurable Implementation (Virtex-4LX15)		
Time	$\{T_{\text{exec}}, T_{\text{reconf}}, T_{\text{transfer}}\}$	{27.0E-6, 7.4, 1.32E-3} ms
Area	$\{A_{\text{global}}$,synthesis$\}$	{6144, 5815} slices
Cost	C_{global}	45.8 s·slices

The above referenced work also considers partial reconfiguration, where the four configurations are set to the size of the largest configuration of 2048 CLB slices. Reconfiguration time was given to be 750 μs, and $C_{\text{partial,comm}}$ was estimated by multiplying the memory controller area (668) by the total period of 20 ms.

4 Results

The results were obtained as described in section 3. The results from the FFT case study are shown in table 4. A_{static} and A_{global} are the actual values for the FPGAs [10], whereas "synthesis" is the synthesis result obtained by ISE 9.1. In addition to CLB slices, DSP48 resources were also utilized. However, these are not included in the cost-model, thus not showed in the table.

From the synthesis results, it is clear that one FFT-stage only consumes 16% of the FPGA's resources in the full static implementation. However, due to the high reconfiguration overhead, the costs and time are higher, 28% and 540% respectively, than for the static reference.

For the second case of the DAB receiver, the results are shown in table 5. The results are a combination of extracts from [4] and the estimates described in section 3.

5 Discussion

For the investigated FFT-case, the results clearly showed that a globally reconfigurable implementation had significantly higher costs than a static implementation, in spite of the possibility of HW sharing. The cost can be reduced by packing more operations into each configuration, and thereby reduce the number of reconfigurations. However, this will not make the reconfigurable solution feasible for this case, as (6) and (7) still cannot be fulfilled. It can be argued that the investigated fully parallel FFT implementation is not a realistic reference and is inefficiently implemented in the FPGA. However, we find that the suggested scenario describes the problem of feasibility estimation for block-processing applications in an illustrative and easily understandable way.

For the investigated DAB-receiver case, the global reconfiguration did not fulfill the conditions (6) and (7), and it was thereby concluded that a globally reconfigurable implementation is not feasible compared to a static solution. This is mainly caused by the long time spent on reconfiguration as shown in table 5. However, the reconfiguration time

Table 5: Results: FPGA-based DAB receiver:

Static Implementation (Virtex-4SX35)		
Time	$T_{\text{static}} = T_{\text{deadline}}$	20 ms
Area	A_{static}	15360 slices
Cost	C_{static}	307 s·slices
Globally Reconfigurable Implementation (Virtex-4SX35)		
Time	$\{T_{\text{exec}}, T_{\text{reconf}}, T_{\text{transfer}}\}$	{4.4, 17.1, 0.63} ms
Area	A_{global}	15360 slices
Cost	C_{global}	340 s·slices
Partially Reconfigurable Implementation (Virtex-4SX35)		
Time	$t_{\text{reconf},0}, \ldots, t_{\text{reconf},3}$	750 μs
	$\{t_{\text{exec},0}, \ldots, t_{\text{exec},3}\}$	{2.26,1.14,0.48,0.11} ms
	T_{partial}	20 ms
Area	$A_0, \ldots, A_3$	2048 slices
	A_{busregs}	668 slices
Cost	$C_{\text{partial,proc}}$	15.1 s·slices
	$C_{\text{partial,comm}}$	13.4 s·slices
	C_{partial}	28.4 s·slices

can be decreased by selecting a smaller FPGA - thus reducing the cost of the globally reconfigurable implementation.

The partially reconfigurable solution for the DAB-case did show a significant reduction in cost and only 9.3% of the resources were utilized. The rest of the resources can either be utilized for other functionalities or a smaller FPGA can be selected. The feasibility conditions (6) and (7) were fulfilled, so a partially reconfigurable implementation is feasible for this application.

The investigated cases show that a reconfigurable implementation may be feasible and may satisfy the time-constraints either due to a very relaxed deadline, or by running the reconfigurable architecture at a higher clock-speed than the non-reconfigurable implementation. Increasing the clock-speed leads to an increased power-consumption, thus we suggest extensive evaluation of power-consumption for future work.

An advantage of the methodology is that it is relatively simple to obtain the estimates and set up the feasibility conditions. However, it requires that the designer performs the partitioning of the application into configurations or modules and performs logic synthesis of these configuration or modules. The partitioning of the application can be performed by an automatic scheduling approach as suggested in section 2.

So far, our methodology does only consider the CLB slices, but other conditions are currently being investigated for memory blocks and DSP slices.

6 Conclusion

In this work we propose a method to evaluate the feasibility of implementing signal processing applications in reconfigurable architectures.

A general condition for feasibility of a globally reconfigurable architecture is closely related to the reconfiguration time and thus the size of the reconfigurable area. The size must be carefully selected so that the reconfiguration time does not exceed the execution time of the static configuration reference. However, as the reconfiguration time is potentially significantly smaller for partially reconfigurable implementations than for globally reconfigurable implementations it is generally preferable to choose a partially reconfigurable solution.

We conclude that the proposed cost-metric makes it possible to evaluate the feasibility considering area-usage and timing. An observation is that timing constraints may be fulfilled by adjusting the clock-speed, thus consideration of power-consumption in the cost-metric is suggested as future work.

References

[1] R. Hartenstein, "Trends in reconfigurable logic and reconfigurable computing," *9th International Conference on Electronics, Circuits and Systems*, vol. 2, September 2002, pp. 801–808.

[2] P. Lysaght et al., "Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas," *International Conference on Field Programmable Logic and Applications*, 2006.

[3] J. P. Delahaye et al., "Software radio and dynamic reconfiguration on a dsp/fpga platform," *3rd Karlsruhe Workshop on Software Radios*, 2004.

[4] M. Ihmig et al., "Resource-efficient sequential architecture for fpga-based dab receiver," *5th Karlsruhe Workshop on Software Radios*, 2008.

[5] C. Bobda, "Synthesis of dataflow graphs for reconfigurable systems using temporal partitioning and temporal placement," Doctor's dissertation, Faculty of Computer Science, Electrical Engineering and Mathematics of the University of Paderborn, May 2003.

[6] S. Hauck, "Configuration prefetch for single context reconfigurable coprocessors," *ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pp. 65–74, 1998.

[7] M. J. Wirthlin and B. L. Hutchings, "Improving functional density using run-time circuit reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 2, pp. 247–256, June 1998.

[8] A. Shoa and S. Shirani, "Run-time reconfigurable systems for digital signal processing applications: a survey," *Journal of VLSI Signal Processing Systems*, vol. 39, no. 3, pp. 213–235, 2005.

[9] P. Manet et al., "Evaluation of dynamic partial reconfiguration in professional electronics applications," *DASIP Workshop on Design and Architectures for Signal and Image Processing*, November 2007.

[10] Xilinx Inc., "Virtex-4 FPGA User Guide," *UG070*, June 2008.

[11] Xilinx Inc., "Virtex-4 FPGA Configuration User Guide," *UG071*, April 2008.

Paper B

Scheduling Temporal Partitions in a Multiprocessing Paradigm for Reconfigurable Architectures

Andreas Popp, Yannick Le Moullec, and Peter Koch

This paper was published in:
NASA/ESA International Conference on Adaptive Hardware Systems

Abstract

In this paper we describe a mapping methodology for heterogeneous reconfigurable architectures consisting of one or more SW processors and one or more reconfigurable units, FPGAs. The mapping methodology consists of a separated track for a) the generation of the configurations for the FPGA by level-based and clustering-based temporal partitioning, and b) the scheduling of those configurations as well as the software tasks, based on two multiprocessor scheduling algorithms: a simple list-based scheduler and the more complex extended dynamic level scheduling algorithm. The mapping methodology is benchmarked by means of randomly created task graphs on an architecture of one SW processor and one FPGA. The results are compared to a 0-1 integer linear programming solution in terms of exploration time as well as the finish-time of all tasks of the application. The results show that, in 90% of the investigated cases, the combination of level-based temporal partitioning and extended dynamic level scheduling gives the best performance in terms of finish-time of the full task-set.

1 Introduction

Most signal processing architectures are both reconfigurable and heterogeneous, consisting of several software processors as well as configurable hardware, typically Field-Programmable Gate Arrays (FPGAs). Moreover, FPGAs provide reconfiguration during runtime, either for the full FPGA area - or for a portion of the area, noted Dynamic Partial Reconfiguration (DPR). Such systems have the possibility to provide better performance than compile-time configured systems in terms of total execution time, logic resource usage, and power consumption [1]. However, in order to obtain such performance benefits, it is necessary to have efficient scheduling techniques and methods which we denote ”mapping methods” in the following.

Existing solutions for mapping applications to reconfigurable heterogeneous architectures target architectures consisting of a software processor connected to a reconfigurable FPGA via a common bus. The software processor serves as the host, either being 1) a simple configuration controller for the reconfigurable hardware, or 2) a processor that utilizes the reconfigurable hardware for acceleration of computationally heavy tasks.

In case 1 where the processor works solely as a configuration controller, approaches for temporal partitioning have been suggested by, among others, Kaul&Vemuri [2] and Purna&Bhatia [3]. Temporal partitioning is the task of dividing a large application into partitions that are mutually exclusive in time, and thus can be executed sequentially on a device that is smaller than needed for fully parallel implementation of the entire application.

In case 2, approaches have been suggested by, among others, Banerjee et al. [4] that formulated the solution as a 0-1 Integer Linear Programming (ILP) problem to obtain the minimum cost in terms of overall execution time. Noguera&Badia [5] proposed a HW/SW partitioning algorithm where tasks are moved between HW and SW until a minimum overall execution time is obtained. The method considers prefetching of configurations to reduce the reconfiguration overhead. The computational complexity of both works is high (non-polynomial for the first), leading to prohibitively long execution times of exploration algorithms, which we from hereon will denote ”exploration times”. An approach with lower computational complexity has been proposed by Chatha&Vemuri [6].

The work consists of an algorithm of five steps: a) HW/SW partitioning, b) temporal partitioning of HW tasks, c) scheduling of HW and SW tasks, d) scheduling of HW reconfigurations, and e) scheduling of communications.

However, as these three approaches do cover a subset of heterogeneous reconfigurable architectures, they are not suited for architectures consisting of several units, both in HW and SW.

Mapping methods for homogeneous SW architectures have been well studied for some time. One of the well known methods is Dynamic Level Scheduling (DLS) by Sih&Lee [7], who in the same connection propose an extended DLS algorithm for heterogeneous architectures.

The previously mentioned approaches do not cover heterogeneous reconfigurable architectures consisting of several processing units, thus in this work we combine the known temporal partitioning algorithms with multiprocessor scheduling algorithms in a scheduling methodology for heterogeneous reconfigurable architectures. The methodology is inspired by Chatha&Vemuri [6] that starts with an initial HW/SW partitioning followed by creation of temporal partitions for HW nodes. The temporal partitions are then treated as super-nodes in a multiprocessing framework - where the super-nodes are tied to a particular unit, the reconfigurable HW unit.

This paper describes the suggested methodology in section 2, including the underlying application and architecture model. This is followed by a series of experiments in section 3 where the mapping results are compared to a 0-1 ILP solution that serves as a lower boundary reference. The results are presented in section 4, followed by a discussion and a conclusion in section 5 and 6, respectively.

2 Mapping Methodology

The proposed mapping methodology is a combination of multiprocessor scheduling and temporal partitioning for reconfigurable architectures, and is outlined in figure 1. The starting point is the specifications of the application, architecture, and cost-library which are all expanded in section 2. Following the specification, the application's tasks are partitioned between HW and SW units, and between several HW units. This is fed back to the original SW multiprocessor scheduling flow, as described in section 2.

Specifications and Modeling

The application is specified as a directed acyclic task-graph, consisting of nodes and edges. The nodes represent tasks, whereas the edges represent data dependencies. The edges are assigned a width, describing the amount of data transferred between the nodes. The task granularity can vary, being both single algorithmic operations as well as larger blocks of operations. The general architecture model is illustrated in figure 2. The model is a composition of Processing Units (PUs), memories, and ports, all connected via buses. PUs are again either SW or HW. SW units have a certain number of cycles per second, whereas HW has a number of resources, each corresponding to the number of logic slices, DSP resources, memory blocks etc. Buses are described by the units they connect, their direction, width, and frequency.

The cost-library binds the application and the architecture together. It contains the cost of various implementation alternatives for each task, i.e. execution time for SW and

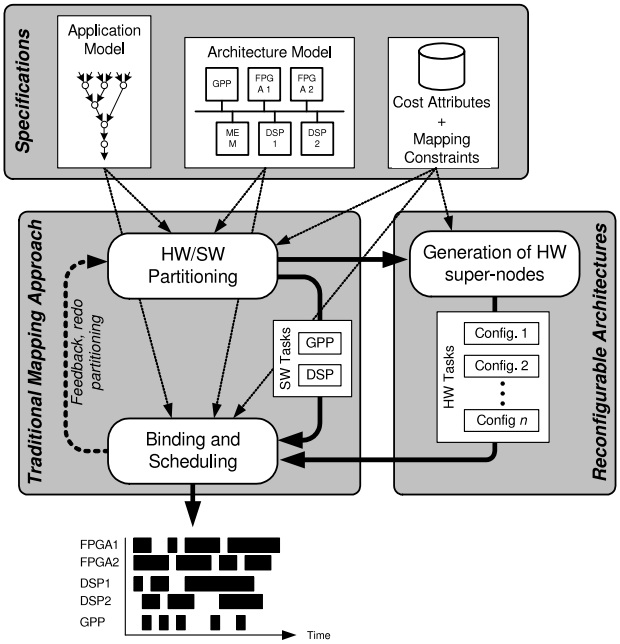


Figure 1: The proposed mapping methodology. The first step is the specification of the application, architecture, and their interrelation via a cost-library. This is followed by a partitioning between HW and SW tasks. The HW tasks are sent to the HW-flow, where the tasks are partitioned into temporal partitions of HW tasks. The HW tasks and their reconfiguration are each considered super-nodes of tasks, which are fed to the multiprocessor binding and scheduling process.

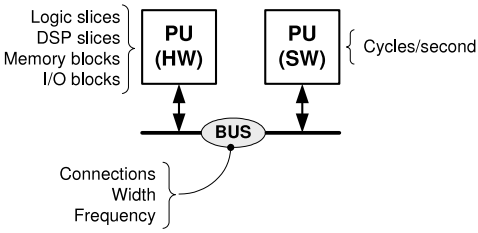


Figure 2: General Architecture Model. The attributes for each architecture element is found by studying the data sheets of the architecture.

execution time, reconfiguration time, and resource usage for HW. Reconfiguration time is derived from the size of the reconfigurable HW. The cost-library is derived by sample implementations of each task, without having to perform the full implementation of the application. Another option is to provide estimates based on previous experiences.

Partitioning

The partitioning approach is based on the values of the cost-library: t_i^{sw} is the SW execution time of task i, t_i^{hw} is the hardware execution time of task i, and $t_{\mathrm{reconf}}^{\mathrm{hw}}$ is the full reconfiguration time of the HW unit. The HW/SW partitioning is based on the principles described in the list below:

1. If logic slice resource usage for task i is larger than the capacity of the HW unit, then partition to **S**W
2. Else If $t_{\mathrm{reconf}}^{\mathrm{hw}} + t_i^{\mathrm{hw}} < t_i^{\mathrm{sw}}$, then partition to **H**W
3. Else $t_{\mathrm{reconf}}^{\mathrm{hw}} + t_i^{\mathrm{hw}} \geq t_i^{\mathrm{sw}}$ is true, so partition to **S**W

As seen in the partitioning scheme, reconfiguration time is included in the HW execution time, assuming that each HW execution must be preceded by reconfiguration.

HW Flow

The partitioning is followed by an extraction of the HW tasks from the application graph.

The task-set is then temporally partitioned, following the two list-scheduling temporal partitioning algorithms by Purna&Bhatia [3]. However, it is a requirement to the execution scheme that temporal partitions do not start execution before all inputs are ready. Thus, there must not be a path through other nodes or partitions from an output to an input in the same partition. Therefore, the temporal partitioning algorithms are extended with a search for paths outside the current partition. If such a path exists, a new partition is created, and the current node is placed in that new partition.

The result of the HW flow is fed to the binding and scheduling by performing an application graph and cost-table update. In the application graph update, the temporal partitions are considered as HW super-nodes, and are fed to the SW flow as super nodes. The cost-table entries for the HW tasks are removed and replaced by cost-table entries for the super-nodes.

The application graph and cost-table update follows the scheme as described below and refers to the illustration of the application graph update in figure 3:

1. All nodes in the same temporal partition are replaced by a single super-node (#1 and #2 in figure 3). This is performed for all temporal partitions. All edges going to/from those nodes are being redirected to the corresponding super-nodes, preserving the direction of the edge.
2. Reconfiguration nodes (**R**1 and **R**2) are added to all the new super-nodes. The reconfiguration nodes have no predecessors, and their only successor will be the corresponding super-node.
3. The cost-table is updated by firstly removing the entries for the nodes that are replaced by super-nodes. Secondly, entries are added for each super-node. The execution time is the maximum execution time of the tasks in the super-node. The resource cost is the sum of all tasks in the super-node.

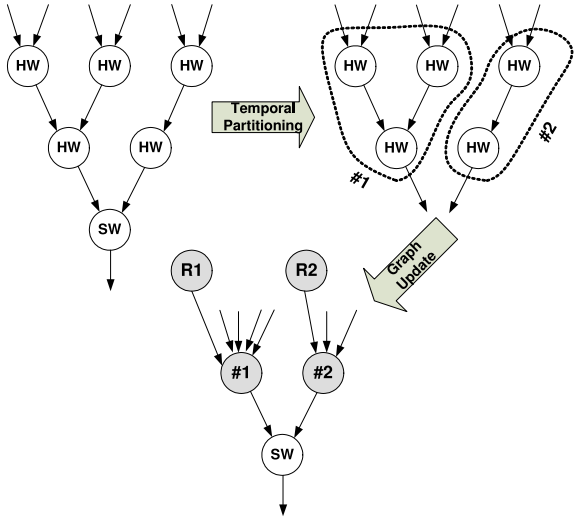


Figure 3: Illustration of the application graph update. Firstly, HW nodes are temporally partitioned. Secondly, nodes in temporal partitions are replaced by super-nodes, followed by insertion of reconfiguration nodes for each super-node.

4. Similarly, entries are added for the reconfiguration nodes. The execution is similar to the reconfiguration time of the unit, and the resource cost is similar to the super-node that is reconfigured.

SW Flow

The SW scheduling flow is based on two approaches:

1. A simple list-scheduler where nodes are scheduled in the order given by the finish-time of their predecessor as well as their mobility, such that the node with the lowest mobility is scheduled first.

2. The extended DLS algorithm by Sih&Lee [7] for heterogeneous processor systems.

For both approaches additional constraints have been included in order to ensure that reconfiguration and execution sequences are performed in the right order, without interruption by other tasks. The two approaches have been implemented in order to be able to compare two SW scheduling algorithms, thus they are both used for scheduling.

For both algorithms, we use a light communication model based on communication time. Communication time between tasks executed in the same unit is assumed to be zero. The transfer of data over the connecting bus is associated with a certain communication time based the the amount of data transferred, the bus width, and the bus frequency.

The extended DLS algorithm has been selected due to its ability to handle heterogeneous multiprocessing architectures consisting of several HW and SW units taking interprocessor communication costs into account. Heterogeneity is represented by varying

Table 1: Description of task-graphs for the experiments. CP denotes the length of the critical path in terms on number of nodes.

Experiment	Tasks	Edges/Task	CP [nodes]
1	5	1.2	3
2	5	1	4
3	10	1.6	3
4	10	0.8	4
5	10	1.2	5
6	10	1.8	5
7	15	0.8	5
8	15	1	8
9	15	1.2	6
10	15	1.53	6

execution times of tasks, which are included in the Dynamic Level (DL) computation. If a task-processor combination is invalid, its execution time is infinity, leading to a DL of minus infinity. This prevents that combination from being selected. The state of the communication resources are modeled as occupied slots of communication. The state is included in two steps of the algorithm:

- **D**L computation: If the communication resource is free to provide communication from the predecessor to the current node, the communication time is assumed to take place right after finishing the predecessor, else the communication is moved to the next free communication slot. Both possibilities influence the Data Available (DA) time, thus the computation of DL.
- **S**cheduling: When the node with the highest DL is scheduled, it is performed based on the calculated start time in the previous step. This is followed by an update of the state of the communication resources.

3 Mapping Experiments

Several mapping experiments have been performed during the development of the framework, they are explained in this section. The experiments were performed as a series of mapping experiments for various task-graphs. The task-graphs had the number of nodes $\{5, 10, 15\}$, with varying numbers of edges and length of the Critical Path (CP). The graphs are described in table 1. All graphs have only a single sink node.

The architecture for all experiments was the same, a HW/SW architecture consisting of one SW processor and one HW unit. The HW unit had 15 logic slices, and the reconfiguration time was 10 cycles. Reconfiguration was assumed not to overlap with HW execution, but has no influence on the SW execution. We assumed a constant transfer time of two cycles between the SW and HW units. This transfer was assumed not to interrupt HW nor SW execution.

The SW and HW execution times as well as the HW-cost were randomly created to each task, based on random distributions in the given intervals.:

- SW execution time: $[1; 20]$

Table 2: Algorithm options for the mapping experiments

No	Temporal Partitioning	Multiprocessor Scheduling
1	Level-based	Simple list-based
2	Clustering-based	Simple list-based
3	Level-based	Extended DLS
4	Clustering-based	Extended DLS
5	0-1 ILP-based Optimal Reference	

- HW execution time: $[1; 10]$
- HW Cost: $[1; 15]$

The experiments were performed for four combinations of our mapping framework as well as the optimal 0-1 ILP reference as indicated in table 2. The ILP problem formulation is outlined in the next section 3. The results were compared in terms of makespan (defined as the total execution time of the task-set) and the exploration time (defined as the execution time of the exploration algorithm). The mapping framework was executed in Matlab® on a standard PC.

ILP Formulation of Optimal Mapping

The optimal mapping reference is performed by an 0-1 ILP formulation of the problem. The formulation is a light version of the work by Banerjee et al. [4] and is described below. The major difference between their work and our work is that we only consider the area and have disregarded HW placement constraints that Banerjee et al. use to make sure that tasks that span several columns are placed in consecutive columns. Furthermore, we have added the precedence constraint for reconfiguration in equation (4), such that a HW area is reconfigured before its tasks are executed. The formulation of the problem allows partial reconfiguration, thus potentially a lower makespan than for the global reconfiguration case. First some binary variables are described, indexed by i as the task-index, $i \in \{0, \ldots, n_{\text{tasks}} - 1\}$, and j as the time-step, $j \in \{0, \ldots, n_{\text{timesteps}} - 1\}$. The variables are:

- $x_{i,j}$ is 1 if task T_i starts execution in HW at timestep j, 0 otherwise.
- $y_{i,j}$ is 1 if task T_i starts execution on the SW processor at timestep j, 0 otherwise.
- $r_{i,j}$ is 1 if the reconfiguration for task T_i starts execution at timestep j, 0 otherwise.
- in_{i_1,i_2} is 1 if the communication along the edge between task T_{i_1} and T_{i_2} incurs a communication delay, 0 otherwise.

Furthermore, the costs are given by the symbols:

- t_i^{sw} is the SW execution time of task T_i.
- t_i^{hw} is the HW execution time of task T_i.
- c_i^{hw} is the HW resource cost of task T_i.

- $t^{\mathrm{hw}}_{\mathrm{reconf}}$ is the time is takes to reconfigure the HW.
- C_{FPGA} is the full logic capacity in terms of CLB logic slices of the FPGA.
- ct_{i_1,i_2} is bus data transfer time from task T_{i_1} to T_{i_2}.

The variables are subject to a series of constraints:

Uniqueness Constraint

Every task executes only once:

$$\forall i, \sum_j (x_{i,j} + y_{i,j}) = 1 \tag{1}$$

SW Processing Constraint

At each time, at most one task is executing on the SW processor:

$$\forall j, \sum_i \sum_{m=j-t^{\mathrm{sw}}_i+1}^{j} y_{i,m} \leq 1 \quad , \tag{2}$$

where the sum over m is performed to include $y_{i,m}$ over all time-steps where a SW task can occupy the SW processor.

Reconfiguration Constraint

For each task, there is at most one configuration, expressed as mutual exclusiveness of SW execution and reconfiguration:

$$\forall i, \sum_j (y_{i,j} + r_{i,j}) \leq 1 \tag{3}$$

Furthermore, if the task is performed in HW, reconfiguration must precede execution:

$$\forall i, \sum_j j \cdot r_{i,j} + \sum_j t^{\mathrm{hw}}_{\mathrm{reconf}} \cdot r_{i,j} - \sum_j j \cdot x_{i,j} \leq 0 \tag{4}$$

FPGA Resource Constraint

For the FPGA, the sum of resources used for execution or reconfiguration at any timestep must not exceed the full size of the FPGA. A sum over m is included similarly to (2):

$$\begin{aligned} \forall j, \sum_i \Big(& \sum_{m=j-t^{\mathrm{hw}}_i+1}^{j} c^{\mathrm{hw}}_{\mathrm{i}} \cdot x_{i,m} + \\ & \sum_{m=j-t^{\mathrm{hw}}_{\mathrm{reconf}}+1}^{j} c^{\mathrm{hw}}_{\mathrm{i}} \cdot r_{i,m} \Big) \quad \leq \quad C_{\mathrm{FPGA}} \end{aligned} \tag{5}$$

Communication Constraint

Communication on the bus should only be performed when tasks connected by edges are performed on different units:

$$\forall \text{edges}(i_1, i_2), \sum_j y_{i_1,j} + y_{i_2,j} + in_{i_1,i_2} = \{0, 1\} \tag{6}$$

Precedence Constraint

$$\forall \text{edges}(i_1, i_2), \sum_j \big(j \cdot x_{i_1,j} + j \cdot y_{i_1,j}\big) + \tag{7}$$

$$\sum_j \big(t_{i_1}^{\text{hw}} \cdot x_{i_1,j} + t_{i_1}^{\text{sw}} \cdot y_{i_1,j}\big) + \tag{8}$$

$$ct_{i_1,i_2} \cdot in_{i_1,i_2} - \sum_j \big(j \cdot x_{i_2,j} + j \cdot y_{i_2,j}\big) \quad \leq \quad 0 \tag{9}$$

The optimization goal is given by minimization of the finish-time of the last task, which can be formulated as:

$$\min \sum_j \big(j \cdot x_{n,j} + j \cdot y_{n,j} + t_i^{\text{hw}} \cdot x_{n,j} + t_i^{\text{sw}} \cdot y_{n,j}\big) \ , \tag{10}$$

where n is the index of the last task (sink node).

Having the ILP-problem defined, it was passed to the solver, `glpsol` version 4.35, from the GNU Linear Programming Kit (GLPK) [8]. The `glpsol` was executed on a standard Linux PC. The results were compared to the result of the mapping framework as described in the previous section 3.

4 Results

The results of the mapping experiments are given by makespan and exploration times shown in the figures 4 and 5, respectively. For the cases where ILP experiments have been performed, the results are shown in the graphs as a rightmost grey bar for each task graph. The optimal ILP solution was only found for 20% of the task-graphs, as the exploration time was simply too long, going beyond more than eight hours for even relatively simple task-graphs with only 10 nodes.

Furthermore, we have included the resulting schedule for task-graph 6 in figure 6, as an illustration of the outcome of the mapping framework.

5 Discussion

When comparing the results presented in figure 5 it is clear that the ILP reference has a significantly higher exploration time than the framework that we propose in section 2 of this paper. However, when looking at the makespan results in figure 4, the mapping framework resulted in a slightly higher (12.5%) makespan for the experiments 1 and

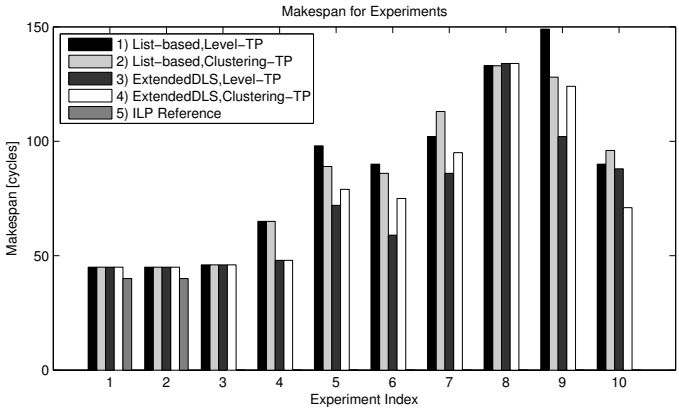


Figure 4: Results in terms of makespan. The bars 1-4 for each experiment are for the proposed framework. Bar 5 is an adapted version of [4]. In 90% of the cases, the Extended DLS algorithm gave better or equally good results compared to the list-based scheduling. Out of those cases, 44% showed additional improvement in makespan by using the level-based temporal partitioning.

2. However, the lower makespan of the optimal reference was made possible due to overlaying of HW execution and reconfiguration in dynamic partial reconfiguration.

When the results are compared for the four different combinations for the presented mapping framework, the results were less clear. For 9 of the 10 cases, the Extended DLS algorithm gave better or equally good results compared to the list-based scheduling. Out of those 9 cases, 4 of them showed that the level-based temporal partitioning gave better results than the clustering-based. Only in 1 of those 9 cases, the level-based performed worse than the clustering-based temporal partitioning. This was surprising since the level-based algorithm would normally lead to more connections to outside partitions, which could potentially increase the HW/SW communication delay. However, the level-based algorithms are less likely to create paths from output to input of the same partition that go through other partitions, thus leading to fewer partitions than the clustering-based approach.

However, it is beneficial to run all four algorithms and compare the results. Such runs do only take short time as seen in figure 5, but gave a highest-to-lowest makespan reduction between 0% and 34%.

The performance of the proposed mapping framework is highly dependent on the early HW/SW partitioning, and it is therefore relevant to consider if this can be improved. First, the reconfiguration time is included for each HW task, even though it may cover reconfiguration of several tasks in parallel (for HW supernodes). This may be improved by weighting the HW reconfiguration time relative to the logic resource usage. However, the partitioner may then not be aware of the risk that small tasks may still require their own partition as described in section 2. Second, there has not been incorporated any feedback loop into the partitioning as indicated in figure 1. This may be beneficial especially for the partitioning cases where the HW and SW execution times are close to each other.

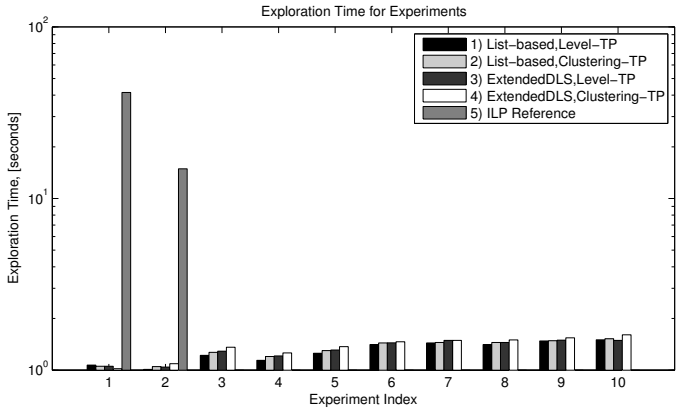


Figure 5: Resulting in terms of exploration time. The bars 1-4 for each experiment are for the proposed framework. Bar 5 is an adapted version of [4], described in section 3. The 0-1 ILP solution was only obtained for 20% of the cases, while it had prohibitively long exploration time for the rest of the cases. The results clearly showed that the 0-1 ILP solution is not a viable alternative, whereas the variation in exploration times in the four options of the proposed mapping framework was insignificant.

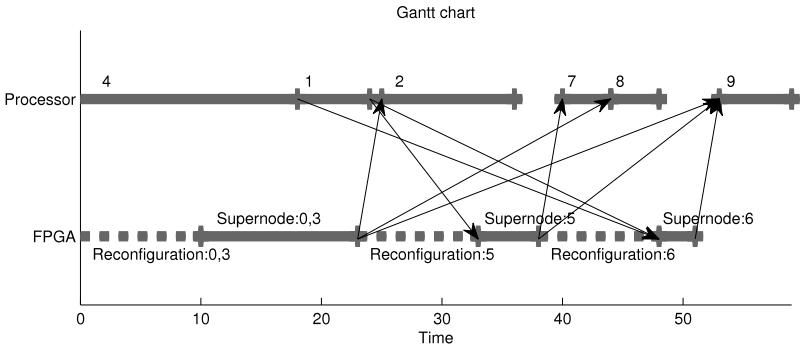


Figure 6: Resulting schedule of task-graph 6, obtained by level-based temporal partitioning and the Extended DLS scheduling. The dotted lines indicate reconfiguration of the HW, and the arrows represent data transfer on the bus.

6 Conclusion

In this paper we presented a mapping framework for reconfigurable heterogeneous architectures consisting of a SW processor and a HW unit with global reconfiguration capability. Our main contribution is that the framework has been developed with the explicit goal to be able to handle heterogeneous reconfigurable architectures consisting of multiple HW and SW units. The framework is based on an application and architecture description, related through a cost-library that provides information of implementation alternatives of each task. The mapping framework performs HW/SW partitioning, and uses

temporal partition algorithms to create HW partitions that can be handled by a scheduling and binding algorithm for heterogeneous multiprocessor architectures.

Mapping experiments were performed for ten task-graphs, with four combinations of two temporal partitions algorithms and two multiprocessor scheduling algorithms. The results showed that the mapping framework had very short exploration time as compared to the (existing) ILP approach, but that the selection of a specific mapping method (out of the four combinations) had an impact of up to 34% compared to the worst performing method. For 90% of the cases, the Extended DLS algorithm in combination with level-based temporal partitioning had the best performance.

We conclude that the proposed mapping methodology is promising and that it can provide designers with a tool for rapid exploration of scheduling strategies for reconfigurable heterogeneous architectures. In order to further improve the methodology, we will conduct the following as future work: a) improve the HW/SW partitioning algorithm, and b) add a feedback loop from the multiprocessor scheduler. Furthermore, future work will also include experiments that cover architectures consisting of multiple SW and HW units.

References

[1] A. Shoa and S. Shirani, "Run-time reconfigurable systems for digital signal processing applications: a survey," *Journal of VLSI Signal Processing Systems*, vol. 39, no. 3, pp. 213–235, 2005.

[2] M. Kaul and R. Vemuri, "Optimal temporal partitioning and synthesis for reconfigurable architectures," in *Proceedings of the conference on Design, automation and test in Europe*, 1998, pp. 389–397.

[3] K. M. G. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *IEEE Trans. Comput.*, vol. 48, no. 6, pp. 579–590, Jun. 1999.

[4] S. Banerjee, E. Bozorgzadeh, and N. D. Dutt, "Integrating physical constraints in hw-sw partitioning for architectures with partial dynamic reconfiguration," *IEEE Trans. VLSI Syst.*, vol. 14, no. 11, pp. 1189–1202, Nov. 2006.

[5] J. Noguera and R. M. Badia, "A hw/sw partitioning algorithm for dynamically reconfigurable architectures," in *Proceeding of Design, Automation and Test in Europe*, Mar. 2001, pp. 729–734.

[6] K. S. Chatha and R. Vemuri, "Hardware-software codesign for dynamically reconfigurable architectures," in *9th International Workshop on Field-Programmable Logic and Applications*, 1999, pp. 175–184.

[7] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, pp. 175–187, Feb. 1993.

[8] GNU, "Gnu linear programming kit (glpk)," http://www.gnu.org/software/glpk/.

Paper C

SystemC-AMS SDF Model Synthesis for Exploration of Heterogeneous Architectures

Andreas Popp, Andreas Herrholz, Kim Grüttner, Yannick Le Moullec, Peter Koch, and Wolfgang Nebel

This paper was published in:
13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems

Abstract

Cost efficient design of embedded HW/SW systems that need to meet certain requirements is a complex task due to the huge number of possible solutions, the ”design space”. Design space exploration methods depend on the designers' input in terms of application description, target architecture, and cost estimates for implementation alternatives. Obtaining feasible pre-implementation cost estimates causes lots of effort since the designer does not have confident information before implementation on the target architecture, or even different target architectures, has been performed.

In this paper we present a methodology suitable for automatic cost estimation of synchronous data flow (SDF) graphs. We propose to start from an executable SystemC-AMS SDF specification, and demonstrate its automatic transformation and implementation for cost estimation on heterogeneous HW/SW architectures. The presented methodology allows the estimation of both HW and SW implementation alternatives of each SDF node based on a quick synthesis approach. These cost estimates are fed to a mapping framework to obtain a static binding and schedule for the architectures under exploration. With the proposed methodology the designer does not have to perform full synthesis and implementation for design space exploration. This is demonstrated by a case study of a Bluetooth baseband unit considered for implementation on a Xilinx Virtex-5 FPGA.

1 Introduction

Heterogenous architectures consisting of different kinds of processing units, like CPUs, DSPs, ASIPs, and custom hardware, provide a vast amount of different implementation alternatives for a given application. In order to obtain the most cost efficient implementation as a System On-a-Chip (SoC), different architecture configurations in terms of area consumption, execution speed etc. need to be evaluated.

However, this is an extensive task due to the many degrees of freedom the designer has in aspects of

1. design of the processing and communication architecture, and
2. options for selecting where and when to perform which tasks (binding and scheduling).

The design of heterogeneous HW/SW systems is aided by the use of methodologies that output a binding and a schedule. We propose to build upon our existing mapping framework [1]. The mapping framework handles heterogeneous static HW/SW architectures and takes as input specifications: 1) application model, 2) architecture model, and 3) cost attributes and mapping constraints.

The mapping framework is a combination of multiprocessor scheduling and pre-scheduling of HW/SW partitions. Following the specification, the application's tasks (described by a directed acyclic data flow graph) are partitioned between HW and SW units based on execution time. The HW flow utilises temporal partitioning to create sequentially executable clusters of HW configurations, represented by super-nodes. The execution time of the super-node is the maximum execution time of the tasks it contains.

For static architectures, a HW super-node is composed based on which tasks are available first. The rest of the tasks are repartitioned to SW. The super-nodes are fed back to the original SW multiprocessor scheduling flow via an updated task graph. The scheduler uses the Extended DLS scheduling algorithm [13] for heterogeneous multiprocessor architectures to schedule SW and HW execution. The scheduling algorithm takes into account interprocessor communication between SW units as well as the HW/SW communication.

The result of the mapping framework mainly depends on the input of cost estimates for given architecture elements and the binding of tasks. Thus it is crucial to have feasible estimates for the mapping to the examined execution architectures in order to obtain realistic exploration results.

In our previous work the experiments were based on abstract application and costs estimates. Therefore, we propose a methodology to provide and obtain such information based on a specification of the application via a Synchronous Data Flow (SDF) model [3]. The methodology presented here serves as a preprocessor for the mapping framework. Thus, our contribution is an approach for HW/SW implementation cost estimation from an SDF application model combined with a heterogeneous multiprocessor system scheduling framework.

The paper is organised as follows: First, we take a look at related work and describe the proposed methodology for automatic cost estimation of SystemC-AMS models for heterogeneous HW/SW architectures. The methodology will be presented along a BlueTooth baseband processing unit from the domain of Software Defined Radio (SDR). This is followed by the application of the methodology to the BlueTooth case study and an exploration of different target architecture mappings is presented. The paper closes with a conclusion.

2 Related Work

Initial work on finding schedules for parallel and sequential execution of SDF models has been presented in [3]. In [4] this work has been extended towards buffer minimisation for sequential execution on a single processor. More recently, a buffer-minimising method for mapping SDF models on heterogeneous HW/SW architectures is presented in [2]; however, it is based on a formal non-functional model and does not consider implementation costs.

There is a lot of previous and ongoing work in the field of hardware and software synthesis from SDF and other data flow models. Most works on software synthesis are based on the work on static scheduling of SDF models but typically they do not take into account costs like area and system performance. In [6] a method for generating hardware from SDF models is proposed based on existing work on software synthesis. While it includes proposals for different hardware architectures, it does not explicitly consider any modelling language or tool-based design automation. Other approaches for hardware synthesis, as in [7] and [8], are based on predefined building blocks restricting the set of available computation primitives. A complete and automated design flow for SDF based hardware synthesis based on the actor language CAL is presented in [9].

A design space pruning tool for FPGA design is presented in [10] where the application is specified in C. Each operation corresponds to one or more basic Register Transfer

Level (RTL) architecture elements, and the cost of different RTL datapaths is based on scheduling onto a combination of those basic RTL elements. A quite similar approach to ours has been presented in [11]. The flow is based on a tool called *SystemCoDesigner* enabling automated exploration and system-level synthesis of HW/SW systems for data flow applications. Initial specification is done in terms of a dynamic data flow model called *SystemMoC* using a set of predefined SystemC modelling elements. This is different to our work, as we use SystemC-AMS as an initial specification of the application. To the best of our knowledge, there is no existing work on hardware synthesis of SDF models using SystemC-AMS for initial specification and SystemC for final hardware implementation.

3 Methodology

The methodology presented in this work is outlined in Fig. 1. The user application is specified as a SystemC-AMS [14] SDF model which describes the tasks to be performed, their interdependency, and their activation based on a user-defined transaction container, also called "token". The SDF model of computation has been selected as it allows the calculation of a static schedule [3], and thus allows its automatic transformation into an acyclic task graph required by the mapping framework. We have chosen SystemC-AMS, because it adds an SDF-layer to the SystemC discrete event simulation kernel enabling C/C++ based specification of executable SDF models and their integration into system level models. Furthermore it is non-proprietary, freely available and has recently become an Open SystemC Initiative standard.

The architecture model in our methodology is composed of architecture templates: Software processing elements with local memory, dedicated hardware processing elements, and communication infrastructure for the interconnection of these processing elements.

The cost estimation, as shown in Fig. 1, performs a characterisation of each SDF module for each processing element of the architecture template library. Our proposed cost estimation approach is based on automatic code transformation which allows the synthesis of the behaviour of each SDF module to either dedicated hardware or software. For hardware cost estimation we perform SystemC to VHDL synthesis with our synthesis tool [15]. Logic synthesis for the chosen target technology (e.g. Xilinx Virtex-5 FPGA) allows accurate cost estimates in terms of area, critical path length in terms of $f_{\max}$, and number of clock cycles per activation of each SDF module. For software cost estimation we propose the use of a lightweight SystemC data type library which can be compiled along with the behavioural code of the SDF block. Therefore, our methodology allows the direct cross compilation and profiling of the SDF module on the chosen target processor. For obtaining the number of clock cycles per SDF module activation we use a generic test bench for HW and SW modules. It generates input stimuli for profiling, based on recorded traces of the SystemC-AMS simulation model. This allows to profile the minimum, average, and maximum number of clock cycles per activation.

To guarantee the correct communication and activation according to the SDF model of computation, our methodology provides certain HW Module and SW Module wrappers, which can be connected via FIFOs. The communication controllers for the FIFOs are part of the architecture template library and can be customised in terms of packet format

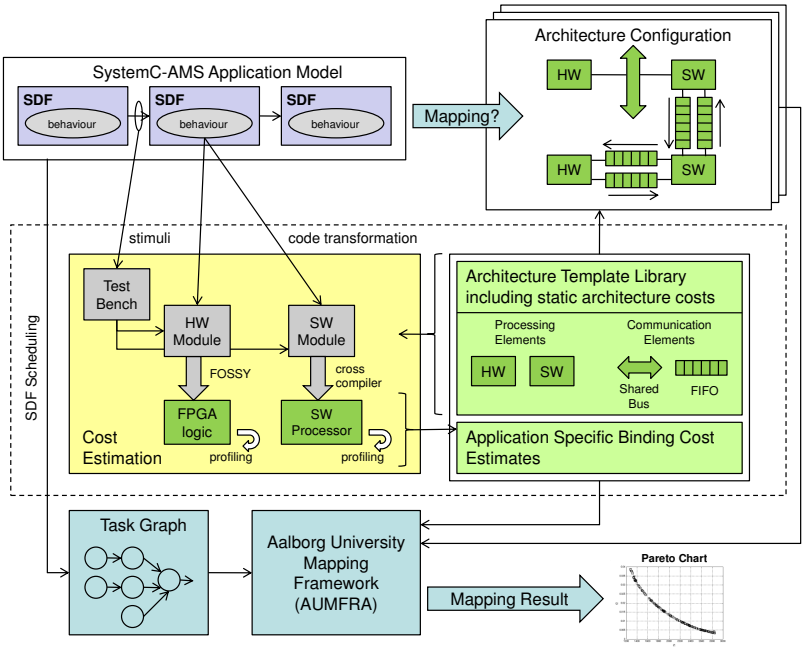


Figure 1: Overview of the proposed methodology.

(i.e. payload or token), communication width, and FIFO depth (number of payloads per FIFO).

The mapping framework takes an architecture configuration, which is an instantiation of processing and communication elements of the architecture template library. The second input of the mapping framework is an acyclic task graph which is generated from the SDF graph through static scheduling.

In the following, we describe the methodology and illustrate it by a case study.

Case Study

The performed case study is based on a baseband processing unit for a BlueTooth (IEEE 802.15.1) transmitter [12]. The baseband unit processes data and packs it into packets that are transmitted by the modulator and RF front-end in time-slots of 625 μs. The SDF representation of the baseband unit is shown in Fig. 2a. It composes a packet based on three parts; synchronisation word, packet header, and payload. We have chosen to focus on the mode of operation with the most processing requirements, the Frequency Hop Synchronisation (FHS) or Data-Medium Rate (DM1) payload types including Cyclic Redundancy Check (CRC) checksums to payload, whitening, and (10,15) Hamming Forward Error Correction [12].

The first step of the methodology is the transformation of the SDF graph into an acyclic task graph which can be processed by the mapping framework.

Application Model (Task Graph)

The application input model of the mapping framework is a directed acyclic data flow graph, $G = (V, E)$, where the nodes, $\{v_0, \ldots, v_{n_v-1}\} \in V$, represent tasks. During this work we call it Task Graph. The edges, $\{e_0, \ldots, e_{n_e-1}\} \in E$, represent data dependencies. Each edge is assigned a width, $w_0, \ldots, w_{n_e-1}$, describing how much data is generated and consumed by the nodes.

This application model is derived from the SDF specification through graph transformation. It starts with the calculation of a static schedule of the SDF specification wich can be represented as an Acyclic Precedence Graph (APG) [5]. The above mentioned application task graph only allows sequential execution along the dependency of tasks. Therefore, the APG needs to be "folded" into a sequential order of tasks. A valid static schedule of the SDF graph of the case study from Fig. 2a is shown as an acyclic task graph model in Fig. 2b.

The SDF scheduling and graph transformation for the case study consist of the following steps:

1. Compute a static schedule of the SDF model and map the scheduling sequence into an acyclic task graph.
   - Allow parallel execution of paths, but sequential execution of all nodes in each path.
   - Disregard the number of tokens in the SDF model.
   - If an SDF node is executed several times, its functionality is executed several times inside the node of the Task Graph (indicated with a number inside the node in Fig. 2a).
2. Add source and sink nodes, as the mapping framework needs these nodes.
3. Determine the number of data packets transferred between nodes
   a) Determine basic data packet size for data transfer between tasks. In the case study, we have selected 32 bits.
   b) Based on the static schedule, determine the total number of input and output bits of each node.
   c) Divide the input and output bits by the basic data packet size and round towards the ceiling to normalise the number of data packets.
   d) No packets are allocated on the edges from source or into sink nodes, as this data is assumed available for processing.

Architecture Model

The architecture model is an abstract architecture model describing the types and numbers of processing elements that are available, as well as how they are interconnected. The processing elements, $p_k \in P$, are assigned an index, $k = \{0, \ldots, n_{pu} - 1\}$, and a type, (sw $\in$ $\mathrm{PE_{SW}}$, hw $\in$ $\mathrm{PE_{HW}}$). $\mathrm{PE_{SW}}$ and $\mathrm{PE_{HW}}$ are the available sets of SW and HW processing elements, such as the software processor type and the type of reconfigurable logic fabric. Interconnection of processing elements is described in terms of buses, $b_m \in B$,

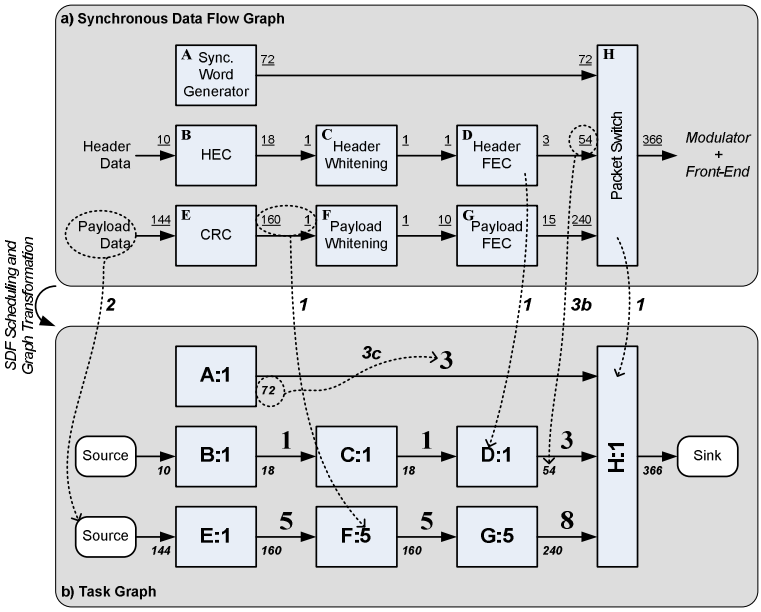


Figure 2: Illustration of the SDF scheduling and graph transformation procedure. The dashed arrows relate to a graph transformation step, with a number related to the steps in Section 3. **a**) Synchronous Data Flow graph for Baseband data path of the BlueTooth transmitter. The underlined numbers on the arrows are tokens, describing the relation between quantity of input and output data (in bit). The switch composes the packet from the three parts: synchronisation word, header, and payload. The header and payload data have been appended error checksums, whitened, and forward error correction coded [12]. The static schedule consists of three parallel paths: 1) A, 2) B-C-D, and 3) E-5F-5G, followed by the join node, H, after all three paths are finished. **b**) Task Graph for case study. The graph is derived from Fig. 2a through our proposed method. The nodes denote operations and the edges denote communication. The letters inside nodes relate to the task in Fig. 2a. The numbers inside nodes determine the number of times the functionality of the task is called before the node is finished. This is determined by the number of activations in the static schedule. The ***bold italic*** numbers is the total number of input and output bits of a node. The edges are annotated with a number, indicating the number of 32-bit data packets transferred between tasks.

also assigned an index, $m = \{0, \ldots, n_b - 1\}$. The buses are furthermore assigned a direction (unidirectional or bidirectional), width, and a speed (as time it takes to transfer one package of data with the size of the bus width). According to its type, the processing and communication elements are assigned their corresponding resource utilisation in terms of Look-Up Tables (LUTs), Flip-Flops (FFs), and Block RAMS (BRAMs).

The cost estimates provide the interrelation between the application and the architecture model. Each task, $v_i \in V$, has at least one cost entry, and there exist cost entries for each binding alternative of a task. The costs are execution times for both HW and SW units, as well as resource utilisation for custom HW units. Thus every cost entry contains at least a task index, a processing unit index, and an execution time.

The architecture investigated for implementation is the Virtex-5 FPGA. The architecture model consists of a number of processing elements that are either a MicroBlaze SW processor or dedicated hardware logic. The processing elements are interconnected by a common bus. The bus is assumed to be implemented by a 32 bit FIFO buffer, the Xilinx Fast Simplex Link (FSL).

The communication model distinguishes between intra and inter processing unit communication. Communication between tasks, executed in the same processing unit, is taken into account by the HW and SW module wrappers. This intra processing unit communication is performed via dedicated and local memories and thus can be considered as side effect free. The inter processing unit communication is implemented over dedicated communication resources with its own timing behaviour. Communication over FSL, as chosen in this work, takes a number of cycles, based on the number of 32 bit data packets that are transmitted. The communication time is estimated by multiplying the number of data packets with the communication delay.

The HW and SW cost estimates were performed using the estimation process as described in Section 3.

Model Transformation & Cost Estimation

To enable the estimation of HW and SW implementation costs the SystemC-AMS SDF model needs to be decomposed and transformed into separate modules suited for either HW or SW implementation. These modules are similar to the task nodes in the application model. One significant advantage of our methodology is that SystemC and its AMS extension are based on C++, and thus enables maximum reuse of the functional parts of each task, minimising the effort to adapt the tasks to different implementation flows and architectures. Therefore, the behavioural part of each block of the SDF model is implemented as a C++ class, as shown in Listing 1, which can either be used inside an SDF (Listing 2), a SystemC HW (Listing 3), or a SW module (Listing 4).

```
class behaviour_class_type {
protected:
  internal_var_type var_0;
public:
  behaviour_class_type() { this->init(); }
  void init() { var_0 = var_0_init_value; }

  void run(sca_sdf_in_if<token_in_type>* in,
           sca_sdf_out_if<token_out_type>* out) {
    //perform user-defined behaviour here
  }
};
```

Listing 1: The user-defined internal variables and behaviour of either the SystemC-AMS, the SystemC HW, and the SW model is implemented by a C++ class

```
SCA_SDF_MODULE(sdf_module_name) {
  sca_sdf_in<token_in_type>   data_in;
  sca_sdf_out<token_out_type> data_out;

  SCA_CTOR(sdf_module_name) { }

  void atributes() {
    data_in.set_rate(input_rate);
    data_out.set_rate(output_rate);
  }
  void init() { beh_inst.init(); }
  void sig_proc() { beh_inst.run(data_in, data_out); }

  behaviour_class_type beh_inst;
};
```

Listing 2: Outline of a SystemC-AMS SDF module to be estimated for HW and SW implementation. The user-defined internal variables and behaviour of the `sig_proc` method are implemented by a C++ class (Listing 1).

For the alternative implementation of the tasks in HW or SW we have created code stubs, already containing required control processes and interfaces for communication and data transfer. The stubs can be adapted to the specific task very easily by setting the class type of the behaviour object and by adapting the calls to the object's interface methods if necessary. For HW, an `SC_MODULE` container is used containing the required FSL interfaces and methods and a control process fetching input data via FSL, performing the required computation and writing the resulting data back via FSL. To obtain cost estimates for the HW implementation of a task, we use our SystemC/C++ synthesis tool [15] translating the transformed SystemC model to synthesisable RT-level VHDL. The generated VHDL code has been used with the RT-level synthesis tool Xilinx XST to obtain a first estimate of the hardware costs (logic blocks, maximum frequency). In our case, execution times in terms of clock cycles have been determined by profiling an early HW prototype on the FPGA platform, but they could also be estimated by simulating or statically analysing the generated VHDL model.

```
SC_MODULE(module_name) {
  sc_in<bool> clock, reset;
  sdf_fsl_in<token_in_type, input_rate>    data_in;
  sdf_fsl_out<token_out_type, output_rate> data_out;

  SC_CTOR(module_name) {
    SC_CTHREAD(proc, clock.pos());
    reset_signal_is(reset, true);
  }

 protected:
  void proc() {
    beh_inst.init();
    wait();
    while(true) {
      if(data_in->ready()) {
        for(int i=0; i<num_cycles; ++i)
          beh_inst.run(&data_in, &data_out);
        data_out->flush();
        wait();
```

```
      } else wait();
    }
  }

  behaviour_class_type beh_inst;
};
```

Listing 3: Outline of a SystemC HW module template that contains the connection to the FSL FIFOs with annotated input and output rates and a clocked thread that waits until the input data is available and calls the user-defined behaviour `num_cycles` times, as obtained from the static scheduling. The `wait` statements define the clock boundaries. After all cycles have been completed it updates the output FIFOs.

For SW, the behaviour class is used inside the `main` function also defining input and output of data. The function is compiled to the embedded target platform and can either be profiled by use of either an emulator or on the platform itself. The outcome is a number of cycles necessary for execution. To enable the comparison of the execution times for HW and SW the estimates are normalised in terms of clock cycles. To enable the usage of bit-true SystemC data types in HW as in SW, we have created a lightweight C++ library providing the same types and semantics as the SystemC data types without the additional overhead of the SystemC simulation kernel, making it usable for embedded SW targets.

```
sdf_fsl_in<token_in_type, input_rate>     data_in;
sdf_fsl_out<token_out_type, output_rate> data_out;

int main() {
  behaviour_class_type beh_name;
  while(true) {
    if(data_in->ready()) {
      for(int i=0; i<num_cycles; ++i)
        beh_name.run(&data_in, &data_out);
      data_out->flush();
    }
  }
}
```

Listing 4: Outline of a SW module template. It has the same structure as the HW module template from Listing 3, but does not contain any explicit timing information in terms of clock boundaries.

4 Experiments

In Section 3 we have presented a case study as an illustration of the methodology described throughout Section 3. This section describes how the architecture exploration is performed based on giving the derived models and cost estimates as input to the mapping framework. The study is composed of three parts: 1) the characterisation of architecture models, 2) obtaining cost estimates for the tasks of the application for the elements of the architecture, and 3) utilisation of the mapping framework to obtain a binding and schedule. The outcome is a Pareto Chart describing the costs of the architecture models, paired with the execution times of the obtained schedules. Section 3 described how to obtain an acyclic task graph as input specification to the mapping framework. The application model in Fig. 2b is the basis of the experiments.

The architecture exploration is based on a set of architecture models, which all share the properties described in Section 3. The architecture is based on the Virtex-5 FPGA with

Table 1: Architecture Configuration Models: The two bottom rows indicate the cost of the basic architecture elements

Description	# μBlaze Units	# FSLs	HW Area Size [LUT]	Total HW Cost [LUT]
Max. HW Area	1	1	62800	69033
10% HW Area	1	1	6900	13133
5% HW Area	1	1	3450	9683
2% HW Area	1	1	1375	7608
1% HW Area	1	1	650	6883
1 μBlaze only	1	0	0	5698
Max. HW Area	2	2	56000	68466
10% HW Area	2	2	6900	19366
5% HW Area	2	2	3450	15916
2% HW Area	2	2	1375	13841
1% HW Area	2	2	650	13116
2 μBlazes only	2	1	0	11931
3 μBlazes only	3	2	0	18164
μBlaze architecture template element				5698
FSL architecture template element				535

Table 2: Results of cost estimation. The execution times are normalised to 100 MHz clock cycles.

Task	HW Cost [LUT]	HW Exec-Time [cycles]	SW Exec-Time [cycles]
SyncWordGen	1165	547	11533
HECGenerator	931	460	141
Whitener (Header)	631	460	1188
EncoderFEC13	856	530	1512
CRCGenerator	981	581	843
Whitener (Payload)	631	577	10560
EncoderFEC23	1134	1162	1696
Packet Switch	-	-	1888
Sum	6329	4317	29361

two basic processing element types: MicroBlaze soft-core processor and FPGA logic. The processing elements are interconnected via FSLs. The costs of these basic elements are based on IP core synthesis using the Xilinx EDK and XST logic synthesis tools.

The MicroBlaze SW Processor is running at a clock frequency of 100 MHz, and the set of architecture models are shown in Table 1. The HW resource costs are measured in terms of Look-Up Tables (LUTs), which is the basis for comparing the cost of the architecture models, HW area size, and the cost of tasks in HW. The total cost of an architecture model is estimated by summing the costs of its elements.

The FSLs are 32 bit wide, and we assume that it is possible to transmit 32 bit for every two clock cycles between processing elements. Internally, within each processing element (FPGA logic or MicroBlaze processor), we assume communication costs to by included in the cost estimates.

The case study is subject to the cost estimation procedure described in Section 3, which results in the costs shown in Table 2. The packet switch task is placed inside a SW processor, as it is mainly communication oriented.

The cost estimates in Table 2 are provided as input to the mapping framework together with the application model and the architecture models. The framework is invoked for each of the architecture models, and outputs total execution time of all tasks. The execution time is interpreted as *makespan*, i.e. the span from the start-time of the first task, to the finish time of the last task.

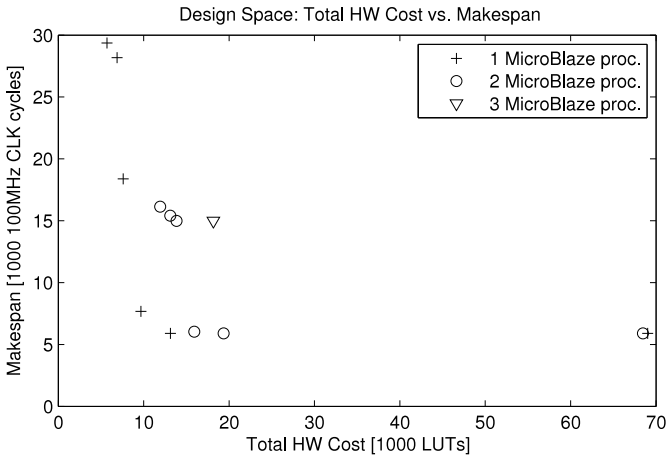


Figure 3: Pareto Chart showing the mapping result: The x-axis shows the total HW cost in terms of Look-Up Tables (Table 1), whereas the y-axis shows the makespan (as determined by the mapping framework) in 100MHz CLK cycles. (Please observe that both axes are in thousands).

The resulting Pareto Chart for all architecture models is shown in Fig. 3. The graph is composed by pairing the total HW cost (rightmost column in Table 1) with the mapping result (makespan) for the corresponding architecture model. Fig. 3 shows that the set of architecture models have different characteristics in terms of area consumption and makespan. Based on the results, it is concluded that the architecture with only 1 MicroBlaze processor is the most feasible as it does not give any speed improvement to increase the number of SW units. The lowest total HW cost (5698 LUTs) is obtained by the pure SW-processor solution with the highest makespan of 29361 cycles. This solution is accepted since the makespan is below the time-slot length of 625 μs. However, the graph shows that by e.g. allocating some HW area (e.g. 3450 LUTs) and thereby increase the area cost by 70% (to 9683 LUTs), the makespan can be reduced to 26% (7676 cycles) of the pure SW implementation.

5 Conclusion

In this paper we have presented a methodology for performing SystemC-AMS based exploration for heterogeneous architectures. Main contribution is the derivation of an acyclic task graph based on a synchronous data flow model in SystemC-AMS, and its cost estimation for HW and SW implementations based on a generic C++ class template representation. Each node of the acyclic task graph is subject to cost-estimation based on an architecture template library. The task graph and its cost characterisation for available architecture template elements are input to our mapping framework. Output of the framework is a Pareto Chart allowing evaluation of implementation costs of various architecture configurations.

The methodology has been demonstrated by an IEEE 802.15.1 BlueTooth transmitter

case study and has partly been performed manually for proof of concept. The methodology can easily be automated, reducing the effort of heterogeneous HW/SW architecture exploration from executable SystemC-AMS SDF specifications. The advantage of the full methodology is that the designer does not have to perform the full synthesis and implementation to obtain cost estimates for performing the exploration. Moreover, our approach is modular and IP-centric, allowing the designer to reuse cost estimates, even when new architecture template elements are added.

As future work we will consider pipelining of tasks in the mapping framework and we plan to investigate the impact of SDF granularity on the exploration process. Furthermore, we plan to extend the cost estimation methodology to reconfigurable architectures.

References

[1] A. Popp, Y. L. Moullec, and P. Koch, "Scheduling temporal partitions in a multiprocessing paradigm for reconfigurable architectures," in *NASA/ESA Conference on Adaptive Hardware and Systems*, 2009.

[2] J. Zhu, I. Sander, and A. Jantsch, "Buffer Minimization of Real-Time Streaming Applications Scheduling on Hybrid CPU/FPGA Architectures," *Proceedings of Design Automation and Test in Europe (DATE'09)*, Nice, France, April 2009.

[3] E.A. Lee and D.G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24-35, January 1987.

[4] P.K.M. Shuvra, S. Bhattacharyya, and E.A. Lee, "Software Synthesis from Dataflow Graphs," Norwell,MA, USA: Kluwer Academic Press, 1996.

[5] S.S. Bhattacharyya and W.S. Levine, "Optimization of Signal Processing Software for Control System Implementation," *Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design*, Munich, Germany, October 4-6, 2006.

[6] M. Edwards and P. Green, "The Implementation of Synchronous Dataflow Graphs Using Reconfigurable Hardware," *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, Springer LNCS, vol. 1896/2000, pp. 739-748, January 2000.

[7] J. Horstmannshoff and H. Meyr, "Efficient building block based RTL code generation from synchronous data flow graphs," *Proceedings of the 37th Annual Design Automation Conference (DAC'00)*, Los Angeles, USA, pp. 552-555, 2000.

[8] M.C. Williamson, "Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications," PhD thesis, EECS Department, University of California, Berkeley, 1998.

[9] J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs: an MPEG-4 Simple Profile decoder case study," *Proceedings of IEEE Workshop on Signal Processing Systems, 2008. (SiPS 2008)*, Washington, USA, 2008.

[10] S. Bilavarn, G. Gogniat, J. Philippe, and L. Bossuet, "Design Space Pruning Through Early Estimations of Area/Delay Tradeoffs for FPGA Implementations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, pp. 1950-1968, 2006.

[11] J. Keinert et al., "SystemCoDesigner: an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, issue 1, January 2009.

[12] IEEE Comp. Soc., "802.15.1: Wireless medium access control (mac) and physical layer (phy) specifications for wireless personal area networks (wpans)," May 2005.

[13] G.C. Sih and E.A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, pp. 175–187, February 1993.

[14] Open SystemC Initiative, "Standard SystemC AMS extensions Language Reference Manual," March 8, 2010.

[15] FOSSY - Functional Oldenburg System Synthesiser, `http://www.system-synthesis.org`.

Paper D

A Mapping Framework for Heterogeneous Reconfigurable Architectures - Combining Temporal Partitioning and Multiprocessor Scheduling

Andreas Popp, Christophe Jégo, Peter Koch, and Yannick Le Moullec

This paper is submitted in June 2010 to:
International Journal of Reconfigurable Computing, Hindawi Publishing Corppration

The layout has been revised

Abstract

Many processing architectures for signal processing algorithms in communication systems are heterogeneous and consist of signal processors and dedicated hardware in the form of Field-Programmable Gate Arrays (FPGAs) that also provide dynamic reconfiguration. However, reconfiguration poses extra challenges in terms of design complexity, making it a complex task for the designer to design reconfigurable architectures and map applications onto such architectures. In this paper we propose a design methodology consisting of a mapping framework to assist the designer in the process of using reconfigurable architectures. The mapping framework improves and combines existing partitioning and scheduling methods for heterogeneous and reconfigurable architectures. The performance of the mapping framework is demonstrated by a case-study of an equalizer for a MIMO receiver on a Xilinx Virtex-5 FPGA. Results show that there is consistency (0.2% difference) between the implementation results and the schedules provided by the mapping framework, when reconfiguration overhead is considered.

1 Introduction

Modern communication and multimedia applications put high requirements on the signal processing hardware/software (HW/SW) platform in terms of high throughput, low chip area usage, and power consumption. Meeting such requirements is often a trade-off between these characteristics as well as development and production costs. Signal processing hardware does normally consist of either one or a combination of Application Specific Integrated Circuits (ASIC), Application Specific Instruction-set Processors (ASIP) or Digital Signal Processors (DSP). However, an intermediate solution is reconfigurable hardware architectures [1]. Reconfigurable hardware can be reconfigured during run-time, thus allowing the same hardware to have several functionalities. Reconfigurable hardware offers the opportunity of resource and energy savings for some applications due to the possibility of time-sharing of the hardware resources.

Reconfiguration can be performed dynamically in two ways: dynamic full reconfiguration, where the full hardware is reconfigured, or Dynamic Partial Reconfiguration (DPR), where parts of the hardware is running, while other parts are being reconfigured. DPR is provided by Xilinx Virtex Field Programmable Gate Arrays (FPGAs), and its application has been demonstrated successfully, especially in the field of Software Defined Radio (SDR). Such demonstrations are e.g. Delahaye et al. [2] and Ihmig et al. [3] where DPR allows the implementation of several functionalities without having to perform parallel implementations of all functionalities. Manet et al. [4] have also suggested future use of DPR in image acquisition and SDR applications that can benefit from power reduction and context change.

Dynamic reconfiguration (both full and partial) provides the possibility of sharing functionality across the chip area. This offers an advantage in terms of area savings and possible energy savings as well. The latter has been demonstrated for FIR filters by Becker et al., [5]. The sharing of functionalities has also been demonstrated in a real-time vision system for production line and robot systems using an architecture with both one software (SW) processor and two FPGAs by Komuro et al., [6]. Similarly to this application, heterogeneous multiprocessing architectures are widely used in many applications. The Embedded Market Study [7] show that 50% of the current (2010) projects

utilize more than one microprocessor or controller, and 60% of those are heterogeneous consisting of different processors or controllers. Multiprocessing SW architectures are employed to be able to scale performance linearly with complexity [8] and thereby provide enough computational power to meet the requirements. Heterogeneous architectures are utilized since telecommunication and multimedia applications represent a variety of algorithms that require heterogeneous architectures in order to obtain an efficient implementation. Such heterogeneous architectures typically consist of SW processors and ASIC or FPGAs to match the algorithm-architecture affinity. Since these architectures contain FPGAs, they are truly heterogeneous, reconfigurable multiprocessing architectures.

The design and implementation of such architectures is a complex task due to the large design space caused by the high number of algorithm-architecture combinations. Furthermore, the binding and scheduling (denoted mapping) for such architectures is a problem requiring methodologies to help the designer perform the task.

Thus we propose a methodology for mapping applications onto heterogeneous, reconfigurable architectures consisting of one or more SW processors and one or more reconfigurable HW units. Our specific contributions are detailed at the end of Sec. 2.

This paper is organized as follows. First, we present related work and outline our contribution in Sec. 2. Second, modeling of reconfigurable architectures is described in Sec. 3, followed by, third, a description of our developed mapping framework in Sec. 4. Fourth, we present simulations of the mapping framework in Sec. 5, followed by a case-study of an equalizer for a MIMO system in Sec 6. Finally, the discussion and conclusion are provided in Sec. 7 and 8, respectively.

2 Related Work

Methods for mapping applications onto reconfigurable heterogeneous architectures mainly target architectures with just one SW and one HW unit. The SW unit is either used for processing or as a configuration controller, where approaches for temporal partitioning have been suggested by, among others, Kaul and Vemuri [9] and Purna and Bhatia [10]. Temporal partitioning is the task of dividing a large application into partitions that are mutually exclusive in time, and thus can be executed sequentially on a device that is smaller than the one needed for full parallel implementation of the entire application.

More interesting, algorithms have been suggested for architectures where the reconfigurable HW is utilized to speed up SW tasks. Among others, Banerjee et al. [11] have formulated the solution as a 0-1 Integer Linear Programming (ILP) problem to obtain the minimum cost in terms of overall execution time. Noguera and Badia [12] proposed a HW/SW partitioning algorithm where tasks are moved between HW and SW until a minimum overall execution time is obtained. The method considers prefetching of configurations to reduce the reconfiguration overhead. The computational complexity of both works is high (non-polynomial for the first), leading to prohibitively long execution times of exploration algorithms. An approach with lower computational complexity has been proposed by Chatha and Vemuri [13]. The work consists of an algorithm with five steps: a) HW/SW partitioning, b) temporal partitioning of HW tasks, c) scheduling of HW and SW tasks, d) scheduling of HW reconfigurations, and e) scheduling of communications. A similar approach has been taken by Galanis et al. [14] who perform early HW/SW

partitioning based on execution time, followed by temporal partitioning of the HW tasks. However, [14] does not describe the combined scheduling of HW and SW tasks.

For heterogeneous, reconfigurable architectures, Dittmann et al. [15] have investigated a methodology for mapping and design of a multiprocessing system residing inside a partially reconfigurable FPGA. The binding and scheduling is performed by an evolutionary algorithm.

Algorithms for multiprocessor scheduling have been investigated, among others by Itradat et al. [16] who take into account inter-processor communication (IPC) overhead. Similarly, Sih and Lee [17] have proposed the Dynamic Level Scheduling (DLS) algorithm, with an extension to heterogeneous architectures.

The mentioned approaches do either utilize algorithms with high complexity (evolutionary algorithms) or do not cover heterogeneous, reconfigurable architectures consisting of several processing units. In this work we combine the known temporal partitioning algorithms with multiprocessor scheduling algorithms in an overall methodology for design space exploration of heterogeneous, reconfigurable architectures. The methodology is inspired by Chatha and Vemuri [13] and Galanis et al. [14], that both start with an initial HW/SW partitioning followed by assigning HW nodes to temporal partitions for HW nodes. Our contribution consists of the ability to handle several processing units by extensions to temporal partitioning algorithms. These are combined with heterogeneous multiprocessor scheduling in order to be able to schedule reconfiguration and HW tasks by a SW scheduler.

3 Modeling Heterogeneous Reconfigurable Architectures

Heterogeneous, reconfigurable architectures consist of both HW and SW processing elements (denoted Processing Units (PUs)). The characteristics of HW and SW executions are explained in the following two sections.

Software Execution

Software describes the program execution on an embedded microprocessor consisting of primarily a datapath (with an Arithmetic Logic Unit (ALU) and a multiplier) and a register file. The execution of a software instruction requires the use of the datapath and register plus memory depending on the instructions that are executed.

Digital Signal Processors may have multiple parallel datapaths like the Texas Instruments TMS320C6xxx series [18] or the Analog Devices Blackfin, Sharc, and TigerSharc series [19]. These processors are Single-Instruction-Multiple-Data (SIMD), such that the dispatch of instructions is performed from a single line of machine-code and thus the use of the parallel datapaths is dependent of the code. The utilization of the pipeline in such datapaths is dependent on the application and the pipeline structure. Thus we abstract ourselves from the pipeline utilization. Similarly, the details of register and memory utilization are not considered. Consequently, SW is considered purely based on the SW execution time without regard to pipeline, multiple datapaths, and register/memory usage.

Hardware Execution

We assume that hardware execution in reconfigurable architectures is performed in FPGAs. FPGAs consist of a certain (often large) number of computation resources in terms of logic units (denoted Configurable Logic Blocks (CLBs) in Xilinx FPGAs) plus possibly some specialized units for e.g. multiplication, or more coarse grained DSP operation blocks. The behavior of the computations is determined by memory cells that must be configured with a *configuration bitstream* at startup. Similarly, reconfiguration is performed by loading a new configuration bitstream. The configuration bitstream is loaded from external memory through a configuration interface (e.g. the Internal Configuration Access Port (ICAP) on Xilinx FPGAs) [23]. In the configuration interface, the bitstream is decoded and written to the corresponding memory cells. Although execution may be performed in parallel in two areas of the same FPGA, configuration has to be scheduled onto the same configuration interface - thus it can be considered sequential.

Execution in FPGAs can be considered as two phases, *reconfiguration* and *execution*. Both phases apply to the same area of CLBs. First they are (re)configured, followed by the actual execution.

Reconfiguration is performed under either internal or external control. By internal control, a state-machine (or embedded soft-core microprocessor) is handling the loading of external configuration data through an internally controlled configuration interface. By external configuration, the configuration interface control is handled externally from e.g. a microprocessor.

Architecture and Execution Model

Our modeling of processing units reflects the fact that both HW and SW execution consume a certain execution time. However, they differ in that SW operations occupy the full SW resources during operation, while HW operations consume a certain fraction (operation dependent) of the HW resources during the execution time. However, until that area has been reconfigured, it is still not available for other functionalities than the one for which the area is configured. Thus, the execution of SW is represented by time, whereas the HW execution is represented by resource usage as well as execution and reconfiguration time.

HW and SW elements are assumed to communicate via the communication infrastructure as illustrated in Fig. 1. For complex architectures, a single common bus can greatly reduce performance due to the bottleneck caused by large Inter-Processor Communication (IPC) overhead. Therefore, larger heterogeneous architectures include more buses - dedicated to various communication depending on the elements to which they are connected. This increases the complexity of the architecture and the architecture model, and thus increases the execution time of the scheduling algorithm. In addition to data communication, there is also program memory and configuration memory communication. However, in our model, this is assumed to be performed and communicated via dedicated memory and buses without conflict with communication data. Thus our communication model considers only the communication of data.

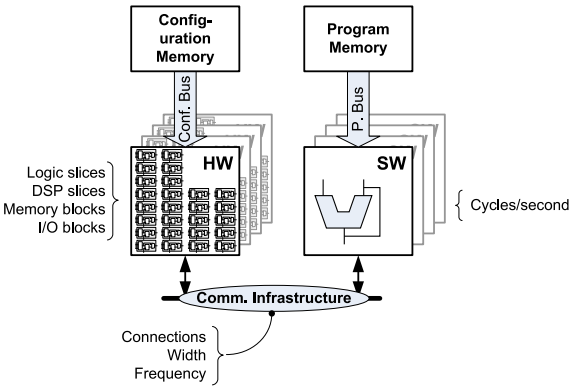


Figure 1: General Architecture Model. The attributes for each architecture element are found by studying its data sheet.

4 Mapping Framework

The proposed design flow builds upon a mapping methodology for heterogeneous, reconfigurable architectures. The mapping methodology, outlined in Fig. 2, consists of a framework that combines multiprocessor scheduling and temporal partitioning for reconfigurable architectures.

The initial step consists of specifying the application, the architecture, and the cost-library. The application is described as a Directed Acyclic Graph (DAG) of which the nodes represent the tasks and the directed edges the data-dependencies between the tasks. The representation of the algorithm by a DAG allows the specification of many types of sequential and parallel signal processing algorithms, both at a fine and coarse level of granularity. The architecture is specified in terms of the size, type, and capability of its processing elements as well as the communication topology of the architecture. The purpose of the cost-library is to provide implementation cost alternatives in terms of execution time and resource usage for each task of the application executing on the architecture.

In the second step, the application's tasks are partitioned between HW and SW units, and between several HW units. Tasks partitioned to HW units are fed to the recnofigurable HW flow to generate partitions of HW task. The partitions are fed to the multiprocessor binding and scheduling flow and are then handled as SW tasks tied to the HW unit.

The purpose of the hardware flow is to generate temporal partitions for the HW tasks that can be treated as super-nodes in the DAG. Thus, these super-nodes can be scheduled as SW nodes in the heterogeneous multi-processor mapping framework. Simulations performed and described in Sec. 5 suggest that the two simple approaches proposed by Purna and Bhatia [10] are suitable as temporal partitioning algorithms.

Input Specifications

The application input model of the mapping framework is a directed acyclic data flow graph, $G = (V, E)$, where the nodes, $\{v_0, \ldots, v_{n_v-1}\} \in V$, represent tasks. The edges,

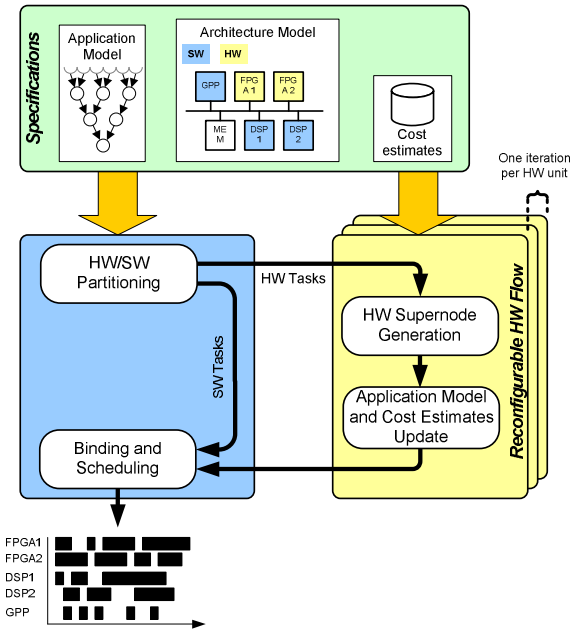


Figure 2: The proposed mapping methodology. The first step is the specification of the application, architecture, and their interrelation via a library of cost estimates. This is followed by a partitioning between HW and SW tasks. The HW tasks are sent to the reconfigurable HW flow, which is performed for each HW unit. In the reconfigurable HW flow, the tasks are partitioned into temporal partitions of HW tasks. The HW tasks and the corresponding reconfiguration of the HW area are considered as super-nodes of tasks. The application graph is updated, before the graph is fed to the multiprocessor binding and scheduling process.

$\{e_0, \ldots, e_{n_e-1}\} \in E$, represent data dependencies. To each edge is assigned a width, $w_0, \ldots, w_{n_e-1}$, describing the quantity of data transmitted between the nodes.

The architecture model is an abstract architecture model describing the types and numbers of available processing elements, as well as their interconnection. To the processing elements, $p_k \in P$, are assigned an index, $k = \{0, \ldots, n_{pu} - 1\}$, and a type, (sw $\in$ PE_{SW}, hw $\in$ PE_{HW}). PE_{SW} and PE_{HW} are the available set of SW and HW processing elements, such as the software processor type and the type of reconfigurable logic fabric. Interconnection of processing elements is described in terms of buses, $b_m \in B$, also assigned an index, $m = \{0, \ldots, n_b - 1\}$. To the buses are furthermore assigned a direction (unidirectional or bidirectional), width, and a speed (as time it takes to transfer one package of data with the size of the bus width). According to their type, the processing elements are assigned the available number of resources in terms of Look-Up-Tables (LUTs), Flip-Flops (FFs), Block RAMS (BRAMs), and DSP slices.

The cost estimates provide the interrelation between the application and the architecture model. Each task, $v_i \in V$, has at least one cost entry, $c_{v_i,p_k} \in C$, and there exist cost entries for each binding alternative of a processing unit, $p_k \in P$, and a task, v_i. The

```
for all v_i ∈ V do
    if t_HW + α t_HW,reconf < t_SW then
        Perform in HW
    else
        Perform in SW
    end if
end for
```

Figure 3: Algorithm for HW/SW partitioning. t_{HW} and $t_{HW,reconf}$ are the HW execution and reconfiguration times, respectively. t_{SW} is the SW execution time.

costs are execution times as well as resource utilization for HW units. Thus every cost entry contains at least a task index, a processing unit index, and an execution time. For SW, the costs are $c_{v_i,p_k,\text{t\_exec}}$, whereas for HW the costs are $c_{v_i,p_k,\text{t\_exec}}$, $c_{v_i,p_k,\text{t\_reconf}}$, $c_{v_i,p_k,\text{LUT}}$, and possibly $c_{v_i,p_k,\text{DSP}}$

HW/SW Partitioning

HW/SW partitioning is performed to split the tasks into HW and SW parts, as HW and SW have different characteristics, as described in Sec. 4. The split between HW and SW is also depicted in Fig. 2, where HW is treated in a special flow.

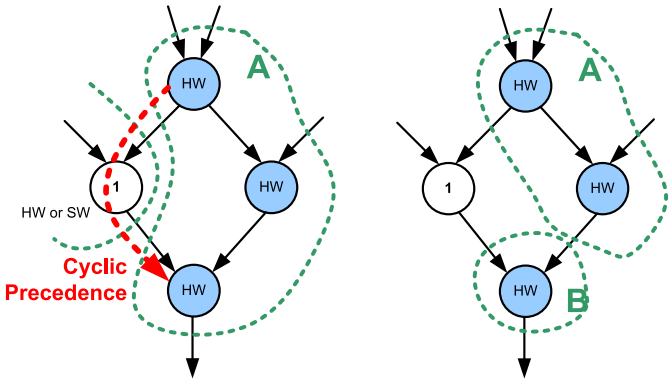
The algorithm for performing the partitioning is given in Fig. 3. The selection is based on which is faster: software execution time t_{SW} or the hardware time, composed of the hardware execution time, t_{HW} and the reconfiguration time, $t_{HW,reconf}$, given the weight α. The low-complexity time-dependent partitioning algorithm was selected due to its simplicity over more sophisticated algorithms taking e.g. also resource costs into account. Furthermore, partitioning based on execution time has shown good performance in heterogeneous, reconfigurable architectures [14].

The reconfiguration time is included to make sure that tasks are not uncritically bound to HW. The reconfiguration time is computed based on the number of LUTs that a task consumes.

The parameter, α, is introduced as a mechanism to regulate the weight of the reconfiguration in the HW/SW partitioning. Experiments with varying the α value are described in Sec. 5. The results clearly show (67.5% of the cases) that the value should be 1. However, the partitioning based on pure execution and reconfiguration times is the reason that for some cases, a value of $\alpha \neq 1$ gives better results than $\alpha = 1$. The reason is that it may cause improper balancing of resource usage.

Reconfigurable HW Flow

The reconfigurable HW flow is split into two sequential parts: generation of HW super-nodes, followed by an update of the application graph and cost table. HW super-nodes are generated based on the temporal partitioning work by Purna and Bhatia [10]. Partitions are the clusters of HW-tasks that are executed (and configured) to the HW at the same time. Purna and Bhatia have proposed a split into two approaches: a) level-based partitioning, and b) clustering-based partitioning - which have different approaches to the



(a) Precedence problem: Cluster A is dependent on input from node 1, which can be either a HW or a SW task. Task 1 is itself dependent on the output of A.

(b) No precedence problem: No inputs to B are dependent on B.

Figure 4: Illustration of the cyclic precedence relation problem in temporal partitioning.

cut between partitions. They both work from the principle that nodes are successively assigned to a partition. If there are enough resources for a node, then it is placed in this partition, otherwise a new partition is created. However, as an addition to the work of Purna and Bhatia, we propose an algorithm to ensure that HW partitions do not contain cyclic precedence relations giving unstable inputs. The problem is presented in [10] and is illustrated in Fig. 4. However, as their algorithm is made to handle all tasks of the application, we propose an extension which consists of a search for paths that indicate cyclic dependencies from an output of a partition to an input to the same partition. In case such a path exists, a flag is raised, and the current task is placed in a new partition.

The level-based algorithm originates from the proposition by Purna and Bhatia and uses a ready list, based on As-Soon-As-Possible (ASAP) levels. The nodes are assigned partitions as outlined in Fig. 5. The set of nodes, V, is the full application and $V_{\mathrm{HWunit}} \subseteq V$ are the nodes for the currently concerned HW unit.

In the level-based partitioning algorithm, p is the index of the current partition, Lev is the current level that is treated (starting from 1 to avoid assigning source nodes), whereas $MaxLevel$ is the maximum ASAP level of the node-set N. *AreaFilled* is the area utilization of the partition.

The functions "IdentifyTerminalEdges(v_i, p)" and "CalculateFSMCost(e)" are related to the cost of control structure for the HW partitions. IdentifyTerminalEdges(v_i,p) identifies all edges that go in/out of the partition when node v_i is added to partition p. CalculateFSMCost(e) computes the size-cost of the control-structure handling the terminal edges, and may become negative if edges are removed.

"TotalCost" is the full HW cost of adding node v_i to the current partition. It consists of the control logic (CalculateFSMCost(e)), the HW cost of node v_i, Size(v_i), and a constant "RCost" which ensures routability of the partition. RCost can be seen as the overhead due to composition of nodes into larger partitions. "SRPU" is the total number of resources in the HW unit.

The path search algorithm, PathSearch(v_i, p), which we propose, is performed as de-

```
    p ← 0, Lev ← 1, AreaFilled ← 0
    Assign ASAP Levels to V_HWunit
    while Lev ≤ MaxLevel do
      for all v_i ∈ V_HWunit ∀Level(v_i) = Lev do
 5:     e ← IdentifyTerminalEdges(v_i,p)
        PathOutsidePartition ← PathSearch(v_i,p)
        TotalCost ← CalculateFSMCost(e) + Size(v_i)
                                        + RCost
        if AreaFilled + TotalCost ≤ SRPU) AND (NOT PathOutsidePartition) then
10:        Partition(v_i) ← p
           AreaFilled ← AreaFilled + TotalCost
        else
           p ← p + 1
           Partition(v_i)← p
15:        e ← IdentifyTerminalEdges(v_i,p)
           AreaFilled ← CalculateFSMCost(e) + Size(v_i) + RCost
        end if
      end for
      Lev = Lev + 1
20: end while
```

Figure 5: Algorithm for level-based partitioning. The algorithm is an extension of the original algorithm [10], with the addition of the PathSearch procedure in line 6 and 9. The PathSearch(v_i,p) algorithm is described in Fig. 6.

scribed in Fig. 6, taking the outset in V. Any node $v_i \in V$ may have a set of *fan_in* and *fan_out* nodes, which are the immediate (direct connection via an edge) predecessors and successors, respectively.

Similarly to the level-based partitioning, a clustering-based partitioning is developed and evaluated. The algorithm is described in Fig. 7 and utilizes the PathSearch algorithm similar to level-based partitioning. Contrary to the level-based partitioning, the clustering-based approach seeks to minimize the number of terminal edges. The order of execution of the edges is controlled by a *ReadyList* that contains all nodes ready to be scheduled. A node is ready to be scheduled if its "in_degree" is equal to zero. The in_degree is the difference between the number of *fan_in*s, and the precedessors which have already been scheduled.

If the in_degree is equal to zero (given by ReadyList), and the node is assigned to the unit, it is partitioned. Otherwise, the ReadyList is updated. ReadyList then contains all ready nodes, not only those for the current unit.

The UpdateReadyList(v_i) algorithm is described in Fig. 8, where it is ensured that in_degree is updated.

After the partitioning of the HW-tasks into configurations, the application graph and cost-table update follow the scheme as described below and refer to the illustration of the application graph update in Fig. 9. Firstly, all nodes in the same temporal partition are replaced by a single super-node (#1 and #2 in figure 9). This is performed for all temporal partitions. All edges going to/from those nodes are being redirected to the corresponding

```
    fan_outs ← fan_outSet(v_i ∈ V) ∀ Partition(v_i)=p
    while fan_outs ≠ ∅ do
      w_k ← fan_outs_0
      fan_outs ← {fan_outs_1, fan_outs_2, ...}
 5:   if w_k ∉ V_HWunit OR Partition(w_k)= ∅ then
        Partition(w_k)← Identifier(p)
      end if
      if fan_outSet(w_k ∈ V) ≠ ∅ then
        fan_outs ← {fan_outs, fan_outSet(w_k)}
10:   end if
    end while
    for u_i ∈ fan_inSet(v_i ∈ V) do
      if Partition(u_i)= Identifier(p) then
        PathOutsidePartition = TRUE
15:   else
        PathOutsidePartition = FALSE
      end if
    end for
```

Figure 6: Proposed PathSearch(v_i, p) algorithm. Lines 1-11 concern the update of the task graph to ensure that all predecessors of node v_i are marked with a partition name, once a partition to v_i is assigned. The path-search operates on a copy of the set V, such that an externally assigned partition is not overwritten. $Identifier(p)$ is an identifier that identifies the partition, however, it cannot be the partition number. The condition in line 5 checks whether w_k is in the current HW unit or if no HW partition is assigned to w_k. Lines 12-17 check the input of the *node of concern*. If the predecessors of node v_i are marked with the partition $Identifier(p)$, there is a path outside the partition and PathOutsidePartition is set to TRUE.

super-nodes, preserving the direction of the edge. Secondly, reconfiguration nodes (**R**1 and **R**2) are added to all the new super-nodes. If the super-node has preceeding super-nodes, an edge is added between their reconfiguration nodes. At this stage, the only successor of the added reconfiguration node is the corresponding super-node. Thirdly, the cost-table is updated by firstly removing the entries for the nodes that are replaced by super-nodes. Secondly, entries are added for each super-node. The execution time is the maximum execution time of the tasks in the super-node, based on an as-soon-as-possible (ASAP) schedule. The resource cost is the sum of all tasks in the super-node. Finally, cost table entries are added for the reconfiguration nodes. The execution time is similar to the reconfiguration time of the unit, and the resource costs are similar to the size of the super-node that is reconfigured.

The choice between level-based and clustering-based temporal partitioning in framework was made based on experiments described in Sec. 5. These experiments surprisingly showed, that even though clustering-based partitioning was intended to reduce communication overhead, the level-based partitioning has in general the best performance in terms of makespan of the obtained schedules. The reason is that the level-based partitioning algorithm was less likely to create HW partitions with cyclic precedence relations, forcing

```
    ReadyList ← ∅
    for all v_i ∈ V with in_degree(v_i)= 0 do
      ReadyList ← {ReadyList, v_i}
    end for
 5: AreaFilled ← 0
    p ← 1
    while ReadyList ≠ ∅ do
      u_i ← ReadyList_0
      if u_i ∈ V_HWunit then
10:     e ← IdentifyTerminalEdges(u_i)
        PathOutsidePartition ← PathSearch(u_i,p)
        TotalCost ← CalculateFSMCost(e) + Size(u_i) + RCost
        if (AreaFilled + TotalCost ≤ SPRU) AND (NOT PathOutsidePartition) then
          Partition(u_i) ← p
15:       AreaFilled ← AreaFilled + TotalCost
        else
          p ← p + 1
          Partition(u_i) ← p
          e ← IdentifyTerminalEdges(v_i,p)
20:       AreaFilled ← CalculateFSMCost(e) + Size(v_i) + RCost
        end if
      end if
      ReadyList ← UpdateReadyList(u_i)
    end while
```

Figure 7: Algorithm for clustering-based temporal partitioning. The algorithm is a modification of the original algorithm [10]. Line 9, 11, and 13 are additions resulting from our work. The UpdateReadyList in line 23 is further described in Fig. 8.

extra HW partitions in order to avoid precedence problems.

Binding and Scheduling

For binding and scheduling, the Extended Dynamic Level Scheduling algorithm (EDLS) is used [17]. The extended DLS algorithm has been selected due to its ability to handle heterogeneous multiprocessing architectures consisting of several HW and SW units taking interprocessor communication costs into account. The algorithm consists of the steps shown in Fig. 10, where i is the task index, and j is the processor index.

The Data Available value, $DA(N_i, P_j)$, is defined as the earliest time all data required by node N_i are available at processor P_j, depending on the communication state Σ. The Dynamic Level, $DL(N_i, P_j, \Sigma)$, is a measure of the match between node, N_i, and processing element, P_j, and is calculated as (1)

$$\begin{aligned} DL(N_i, P_j, \Sigma) &= SL(N_i) + \Delta(N_i, P_j) \\ &\quad - \max\{TF(P_j, \Sigma), DA(N_i, P_j, \Sigma)\} \ , \end{aligned} \tag{1}$$

```
    fan_outs ← fan_outSet(v_i ∈ V)
    while fan_outs ≠ ∅ do
      w_k ← fan_outs_0
      fan_outs ← {fan_outs_1, fan_outs_2, ...}
5:    in_degree(w_k) ← in_degree(w_k)−1
      if in_degree(w_k)= 0 AND w_k ∉ ReadyList then
        ReadyList ← {ReadyList,w_k}
      end if
    end while
```

Figure 8: UpdateReadyList(v_i) algorithm for the clustering-based temporal partitioning algorithm in Fig 7.

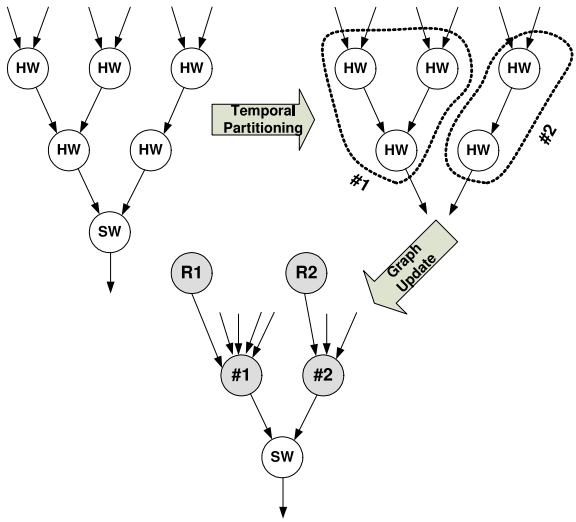


Figure 9: Illustration of the application graph update. Firstly, HW nodes are temporally partitioned. Secondly, nodes in temporal partitions are replaced by super-nodes, followed by insertion of reconfiguration nodes for each super-node.

where $TF(P_j, \Sigma)$ is the finish-time of processor P_j's currently executing task (if any). $\Delta(N_i, P_j)$ is an adjustment factor to handle heterogeneity and is defined as the deviation from a node's median execution time, $\Delta(N_i, P_j) = E^*(N_i) - E(N_i, P_j)$, where $E^*(N_i)$ is the median execution time for node N_i for valid node/processor combinations. $E(N_i, P_j)$ is the execution time of node N_i on processor P_j, and is set to infinity for invalid node/processor combinations, leading to a DL of $-\infty$.

The state of the communication resources, Σ, is modeled as occupied slots of the communication bus. The state includes the time-slots where the bus is occupied. A time-slot is one (or a set of consecutive) clock-cycles. When scheduling the communication, it is evaluated whether the bus is ready. In case a bus conflict occurs, the communication is moved to the next free time-slot as illustrated in Fig. 12. The state is included in two steps of the EDLS algorithm:

```
Compute Static Level (SL)
Find ready nodes and put in Ready List (RL)
while RL not empty do
  for all Combinations of Ready-Nodes (N_i), and processors, P_k do
    Find Ready Processor (RP) for task N_i
    Compute Data Available DA(N_i, P_j, Σ) for task N_i for each processor, P_j.
    Compute DL(N_i, P_j, Σ)
  end for
  Select and schedule the node/processor combination with max{DL(N_i, P_j, Σ)}
  Update communication state Σ
  Update Ready List
end while
```

Figure 10: Extended Dynamic Level Scheduling (EDLS) algorithm based on [17]. i is the task index and j is the processor index. The Data Available value, DA, is defined in calculated as described in Fig. 11. Dynamic Level, DL, is defined by (1).

- **D**A computation: If the communication resource is free to provide communication from the predecessor to the current node, the communication time is assumed to take place right after finishing the predecessor, otherwise the communication is moved to the next free time-slot. If more than one edge utilize the same communication slot, the communication is scheduled sequentially on the given communication bus, as illustrated in Fig. 12. The computation of DA is described in Fig. 11. The DA computation is dependent on the communication state, thus the DL computation is dependent on Σ as well.

- **S**cheduling of nodes: When the node with the highest DL is scheduled, it is performed based on the calculated start time in the previous step. This is followed by an update of the state of the communication resources, based on the task N_i and P_j selected for scheduling. The values of $\Sigma_{\mathrm{Temp}}[N_i, P_j]$ are appended to Σ.

Algorithm Complexity

The complexity of the algorithms depend on a few parameters: N and M are the number of tasks and edges in the application graph, respectively. P is the number of processing elements in the architecture model, no difference is made between HW and SW processing elements. The algorithm complexity is recapitulated in Table 1.

The reconfigurable HW flow is described in Sec. 4. The original level- and clustering-based partitioning algorithms have the complexity $O(N + M)$. However, the proposed path search algorithm traverses all edges, so the complexity is $O(N + (N \times M))$. The cost-table and application graph update have the complexity of $O(N + M)$, but the full reconfigurable HW flow may be performed P times leading to a complexity of $O(P(N + M + N \times M))$.

The multiprocessor scheduling algorithm, described in Sec. 4, is initialized by the computation of SL ($O(N)$). The other operations are performed up to $N \times P$ times for N tasks, thus $O(P \times N^2)$. However, the DA values cause the traversing of up to $N \times M$

```
    Σ_Temp[N_i, P_j] ← Empty
    if fan_inSet(N_i)= ∅ then
       DataAvailable(N_i,P_j,Σ) ← 0;
    else
 5:    for all k ∈fan_inSet(N_i) do
          k_edge ← FindEdge(k, N_i);
          dataAvailable[k]←finishTime(N_i)
          BusCandidates(N_i,P_j)← { }
          if ProcessingUnit(k)≠ P_j then
10:          CBus ← FindBus(ProcessingUnit(k),P_j)
             CommDelay ← ComputeDelay(CBus,k_edge)
             (FreeSlot,Σ_Temp[N_i, P_j])←
             FindFreeSlotAssign(CBus,Σ,Σ_Temp[N_i, P_j])
             if FreeSlot.Finish ≥ dataAvailable[k] then
15:             dataAvailable[k] ← FreeSlot.Finish
             end if
          end if
       end for
    end if
```

Figure 11: Computation of DataAvailable(N_i,P_j,Σ). $\Sigma_{\text{Temp}}[N_i, P_j]$ contains correlated values in arrays: BusIndex, Start, Finish, Edge for each task and processor. The functionality of FindFreeSlotAssign(CBus,Σ,$\Sigma_{\text{Temp}}[N_i, P_j]$) in line 11 is illustrated in Fig. 12.

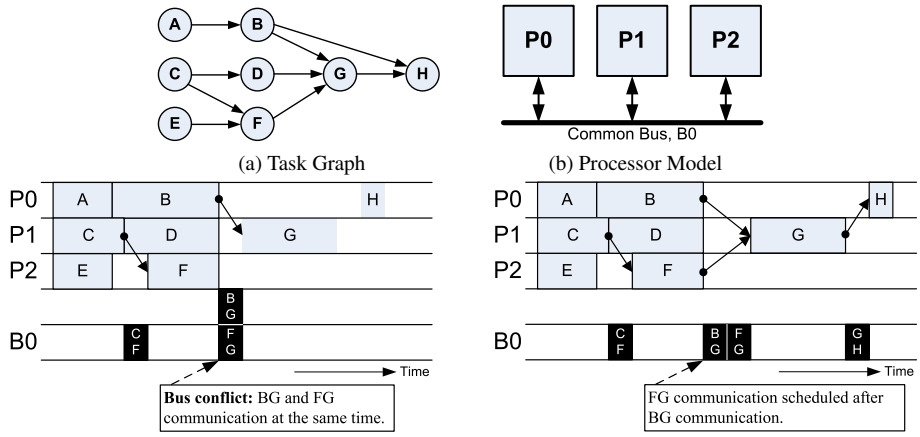
Table 1: Complexity of the algorithms used in the mapping framework. N and M are the number of tasks and edges, respectively, in the application graph. P is the number of processing elements in the architecture model.

Algorithm	Complexity (O)
HW/SW Partitioning	$O(N \times P)$
Temporal Partitioning	$O(N + (N \times M))$
Cost Table and Graph Update	$O(N + M)$
Full Reconfigurable HW Flow	$O(P(N + M + N \times M))$
Extended DLS	$O(N + P \times M \times N^3)$
Total	$\mathbf{O(N \times P + P(N + M + N \times M) + N + P \times M \times N^3) \rightarrow O(P \times M \times N^3)}$

edges, so the full complexity of extended DLS is $O(N + P \times M \times N^3)$. Finally, the overall complexity approaches $O(P \times M \times N^3)$ for high N and M.

5 Simulations

The behavior of the mapping framework has been evaluated by simulations on abstract examples, e.g. [20]. These simulations are outlined in this section. The simulations were performed as a series of mapping experiments for various task-graphs.



(a) Task Graph

(b) Processor Model

(c) Bus conflict (**G** and **H** not scheduled): **B** to **G** communication is supposed to be performed at the same time as the **F** to **G** communication.

(d) Resolved Bus conflict: **B** to **G** communication is scheduled before **F** to **G** communication, thus the conflict is resolved.

Figure 12: Illustration of the communication modeling. The state Σ of the communication resources are saved as the used slots.

Table 2: Description of application graphs for the experiments. CP denotes the length of the critical path in terms on number of nodes.

Case	Nodes	Edges/Node	CP [nodes]
1	5	1.2	3
2	5	1	4
3	10	1.6	3
4	10	0.8	4
5	10	1.2	5
6	10	1.8	5
7	15	0.8	5
8	15	1	8
9	15	1.2	6
10	15	1.53	6
11	20	1.15	8
12	16	1.06	4

Reconfiguration Weight, α

In order to evaluate the influence of the weight, α, of the reconfiguration to execution time for HW/SW partitioning, some simulations were carried out. A number of task-graphs were manually generated with the node, edges/node and critical path as described in Table 2.

All simulations were based on the same architecture model: a HW/SW architecture consisting of one SW processor and one HW unit containing 8160 logic slices and 288 DSP slices. The full reconfiguration time of the HW unit was set to 625604 cycles. The costs were based on the Xilinx Virtex-5 SX50T FPGA [23].

The cost-tables were generated from uniform distributions in the intervals:

- HW Cost: $[10; 8000]$ logic slices, $[0; 250]$ dsp slices
- HW Execution time: $[20; 50000]$ cycles

- SW Execution time: $[20\,R_{\mathrm{HWSW}}; 50000\,R_{\mathrm{HWSW}}]$ cycles

All simulations (for $\alpha = [0, 5]$) were performed based on five different HW to SW ratios on execution time: $R_{\mathrm{HWSW}} \in \{1, 5, 10, 100, 1000\}$. The HW reconfiguration time was based on the ratio between HW Cost and HW Resources, multiplied by the full device reconfiguration time of 625604 cycles. These cost-tables entail 60 experiments using level-based partitioning, and 60 experiments using clustering-based partitioning. The results generated by the mapping framework (in terms of makespan) are shown in Fig. 13 and Fig. 14.

The results show that in general, the best mapping (shown by the shortest makespan) is obtained by a value of $\alpha = 1$ in 67.5% of the cases with these experiments. The results have been examined in order to determine a pattern as well as a reason for not reaching 100% for $\alpha = 1$, e.g. $\alpha = 0.5$ in Fig 13 (graph 2) and $\alpha = 3$ in Fig 13 (graph 3 and 5). The reason is that for heterogeneous multiprocessor systems, partitioning cannot be performed with the sole comparison of HW and SW execution times as there is a risk that the load becomes unbalanced and the execution of tasks has to be postponed until the finish of an overloaded processing unit. Thus the total execution time is increased even if all processing is performed in the fastest unit.

Temporal Partitioning Algorithm Selection

Further simulations were performed in order to evaluate which temporal partioning algorithm provided the best result. The simulation results are also presented in [20]. Application cases 1-10 in Table 2 were considered for an abstract architecture model and cost library: a HW/SW architecture consisting of one SW processor and one HW unit. The HW unit had 15 logic slices, and the reconfiguration time was 10 cycles. We assumed a constant transfer time of two cycles between the SW and HW units. It was assumed that data transfer did not interrupt HW nor SW execution. The SW and HW execution times as well as the HW-cost were randomly generated to each task, based on random distributions in the given intervals:

- SW execution time: $[1; 20]$ cycles
- HW execution time: $[1; 10]$ cycles
- HW Cost: $[1; 15]$ logic slices

The results of the simulations are shown in Fig. 15. Based on the results, in 90% of the cases, the level-based partitioning performed equal to or better than the clustering based approach.

6 Case Study: Iterative Receiver for MIMO Systems

A case study was performed in order to evaluate the quality of the schedules generated by the mapping framework. The considered case is a Minimum Mean Square Error (MMSE) equalizer for a Multiple-Input Multiple-Output (MIMO) receiver implemented on a Xilinx Virtex-5 SX50T FPGA.

The system under consideration is a linearly precoded MIMO system with n_t antennas at the transmitter and n_r antennas at the receiver. The transmitter scheme is a serial

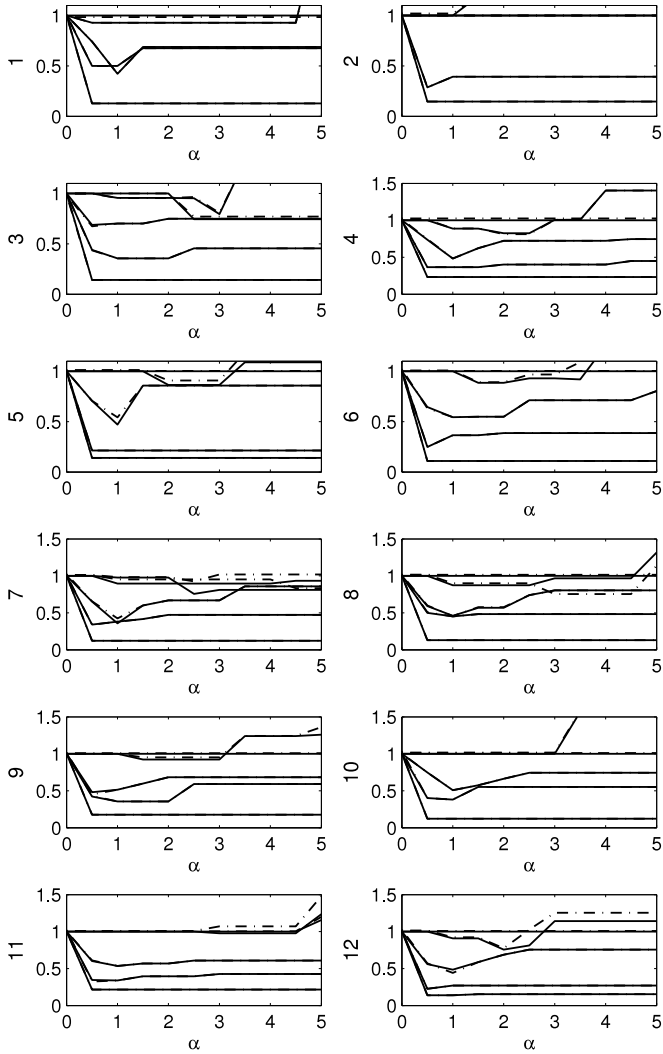


Figure 13: Simulations with varying α parameters. The results are sorted by the experiments index (shown to the left of the Y-axis of each graph) related to Table 2. The x-axis shows α and the y-axis shows the makespan, normalized to the value for $\alpha = 0$ for each experiment.

concatenation of a binary convolutional code, a bit interleaver, a bit-to-symbol mapper followed by a linear precoder, a space interleaver and the n_t transmit antennas. We have chosen a MIMO system that employs $n_t = 2$ transmit antennas, $n_r = 2$ receive antennas and a QPSK modulation. The linear precoding is designed to exploit the transmit spatial diversity. The precoding matrix is of Hadamard type and its size is equal to $n_t = 2$. We consider the convolutional code of rate 1/2 and 64 states $(133, 171)_8$ used in the IEEE802.11 a/g WLAN standard. As IEEE802.11 supports variable length frames, a

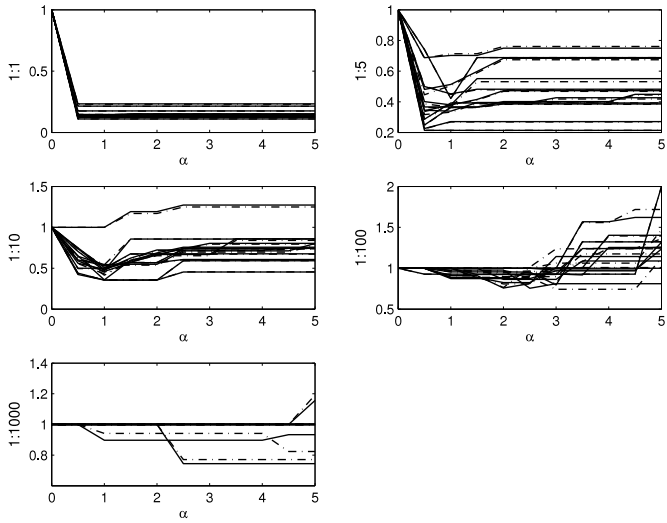


Figure 14: Simulations with varying α parameters. The results are sorted by the HW to SW execution time ratios (shown to the left of the Y-axis of each graph) in the interval $1, 5, 10, 100, 1000$. The x-axis shows α and the y-axis shows the makespan, normalized to the value for $\alpha = 0$ for each experiment. The solid lines indicate Level-based TP, whereas the dash-dotted lines indicate Clustering-based TP.

frame size of $N_{frame} = 2048$ information bits has been chosen for a first prototype. We also consider a Rayleigh flat fading MIMO channel that is perfectly known by the receiver. The MIMO channel is quasi-static in that there is a single realization of the channel per frame. Detailed information can be found in [21].

The iterative receiver, illustrated in Fig. 16, is divided into two main elements: a MIMO equalizer, which processes jointly linear precoding and MIMO signal demapping, and a channel decoder, improving the information on the coded bits and estimating the information bit sequence. The iterative process is based on the exchange of soft values between these two elements. This exchange is maintained thanks to soft mapper/demapper converting Log Likelihood Ratios (LLRs) required by the channel decoder to and from complex symbols. The MIMO equalization is based on the MMSE criterion with interference cancellation, and the channel decoder is based on the SUBMAP forward-backward algorithm. At the first iteration, no a priori information is available at the equalizer input. Through the iterations, the a priori probability on the constellation points becomes more and more accurate. The soft interference cancellation minimum mean square error equalizer treats each of the symbols in the transmitted symbol vector, $\mathbf{x}$, as being distorted by the other symbols in $\mathbf{x}$ and the noise. The interference caused by the symbols is physically due to multiple antenna interference and linear precoding interference. Taking advantage of a priori information available at its input in the form of soft estimated complex symbols, the MMSE equalizer estimates and cancels the interference due to the other symbols.

A new formulation of the MMSE algorithm has been considered as suggested in [22].

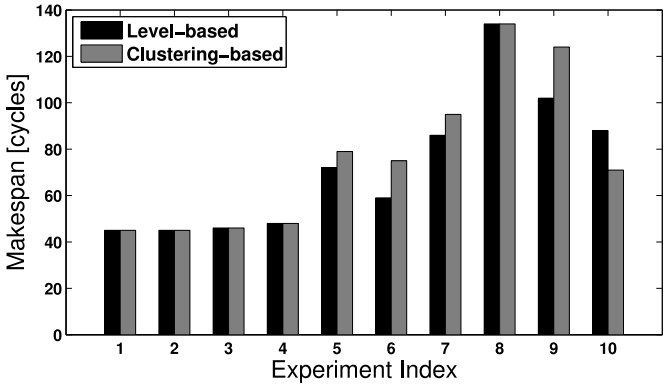


Figure 15: Simulation result for experiments related to the task graphs of Table 2. Results are given for both the level-based temporal partitioning (black) and the clustering-based temporal partitioning (gray) results.

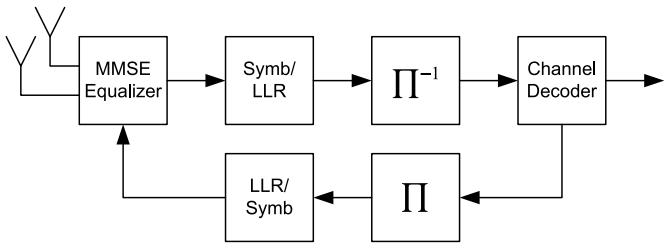


Figure 16: Iterative Receiver Scheme. The received signal is processed by the iterative MMSE Equalizer and the complex symbols are converted to Log Likelihood Ratios (LRR) and sent to the Channel Decoder through a deinterleaver, Π^{-1}. The decoded symbols are fed back to the MMSE Equalizer.

According to this, the estimated symbols at the MMSE-IC output are given by equation (2):

$$\tilde{\mathbf{x}} = diag(\lambda)(\mathbf{S}^H.\mathcal{H}^H.\mathcal{F}^H(\mathbf{y} - \mathcal{H}.\mathbf{S}.\hat{\mathbf{x}}) + diag(\beta).\hat{\mathbf{x}}) \tag{2}$$

where,

$\tilde{\mathbf{x}}$: vector of estimated symbols

$\mathcal{H}$: channel matrix, $\mathbf{S}$: space-time matrix,

$\mathbf{y}$: received signal, $\hat{\mathbf{x}}$: vector of decoded symbols,

σ_x^2 : constellation variance,

$\sigma_{\hat{x}}^2$: variance of estimated symbols,

$\mathcal{F} = ((\sigma_x^2 - \sigma_{\hat{x}}^2)\mathcal{H}.\mathcal{H}^H + \sigma_w^2.\mathbf{I})^{-1}$: equalization matrix

$(\beta, \lambda)s$: equalization coefficients,

The Hadamard precoding matrix leads to a space-time matrix that involves (± 1)s. The proposed formulation takes advantage of this property. Thus, this approach simplifies the space-time demapping by replacing multipliers by adders/subtractors and consequently reduces the total number of operations executed by the detector blocks. Moreover, the coefficients of the channel matrix $\mathcal{H}$ are memorized once per frame for each channel realization. Thus, $\mathcal{F}$ is computed at each iteration, but only once per channel realization.

The modeling of the equalizer is described in the following section.

MMSE-IC Equalizer Modeling

The MMSE-IC equalizer can be decomposed into two main parts (**B1** and **B2**). A first part, **B1**, generates the complex matrix $\mathcal{A} = \mathcal{H}^H.\mathcal{F}^H$ and the equalization coefficients: β, λ and $\mathbf{g} = \beta \times \lambda$. A second part enables the subtraction of the interference term from the received signal $\mathbf{y}$, which is denoted by **B2**. Moreover, this part generates the estimated symbols from the data obtained during the previous computations. The first part, **B1**, has to be performed once per frame because the equalization coefficients have to be computed only once per frame since the MIMO channel is quasi-static. In contrast, part **B2** is executed all along the received frame. The resulting application model is shown in Fig. 17 and includes task nodes as well as source and sink nodes to represent data input to the tasks.

The cost estimates for the tasks identified in the task graph are obtained as described in the following section.

Cost Estimates

The mapping experiments started by obtaining cost estimates. The FPGA under consideration is the Xilinx Virtex5-SX50T FPGA [23] on the Xilinx ML506 evaluation board.

Each data value is represented by 16 bits. Complex numbers are represented as an array of two numbers, one for the real part, another for the imaginary part - thus 32 bits. Vectors and matrices are represented by arrays of complex numbers.

To obtain the SW costs, a softcore processor project is created in Xilinx Platform Studio (XPS) 9.2i. The processor contains external memory, a UART, a timer, and Fast Simplex Link (FSL) ports to communicate with surrounding logic via a FIFO buffer. The process has several steps. Firstly, the HW platform is synthesized and the bitstream is loaded onto the FPGA. Secondly, the SW project (with the code to profile) is compiled and loaded onto the HW target via the Xilinx Microprocessor Debugger (XMD) tool. Finally, the code is executed, and outputs the number of cycles to a terminal via the UART interface.

The HW costs are obtained by sample projects for each task type coded in VHDL. We use the Xilinx ISE 9.2i synthesis tools to obtain cost estimates. Execution time is estimated based on the Xilinx Post Place & Route Timing Simulator.

The cost of the SW project is given in Table 3, based on the synthesis output.

For HW/SW communication the FSL-link is used to transfer data between HW and SW. Communication time is estimated to be four cycles per 32 bit word, based on profiling the transmission of sending data over the FSL-link.

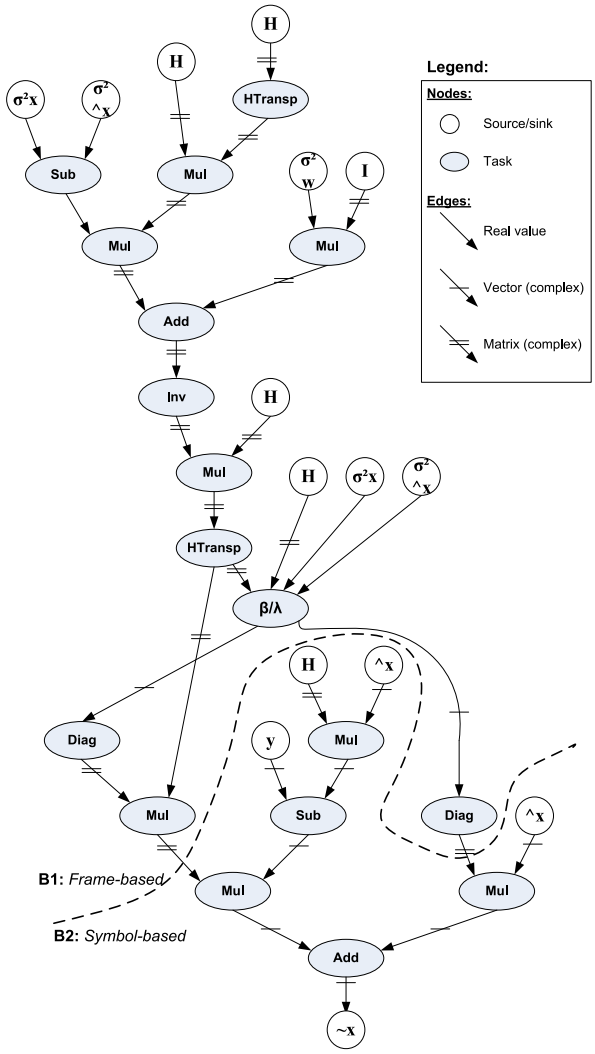


Figure 17: Application graph for MMSE-IC Equalizer. The tasks are divided into a frame-based (**B1**) and symbol-based (**B2**) part. The grey-filled nodes are tasks, whereas the white circles are source or sink-nodes. Dependencies are represented by edges that are either single scalar values or vectors or matrices of complex data.

Table 3: Cost of SW project: 32 bit HW multiplier is used, optimization setting is based on performance.

Resource	SW processor usage	Total available in FPGA [23]
LUT-FF pair	7478	32640
DSP48E slice	3	288
Block RAM	16	132

Table 4: Cost Estimation Results for the MMSE-IC Equalizer. Times are given in 100 MHz cycles, Area in terms of LUT-FF pairs or DSP slices. β/λ Generation is performed in SW only.

Task	t_{SW} [cycles]	t_{HW} [cycles]	$A_{\mathrm{HW,LUT-FF}}$ [LUT-FF pairs]	$A_{\mathrm{HW,DSP}}$ [slices]
Scalar Sub	4	1	16	0
Vector Add/Sub	76	18	486	0
Matrix Add/Sub	97	31	792	0
Matrix Mul	639	62	755	8
Matrix/Vector Mul	321	34	688	8
Matrix Scaling	89	34	472	4
Diagonalization	54	14	361	0
Matrix Inversion	823	103	1426	14
Hermitian Transpose	59	33	350	0
β/λ Generation	1520	-	-	-

Two possible HW supernodes were synthesized in order to quantify the routing cost, RCost, and input/output control cost, FSMCost, for supernodes. Experiments lead to values of RCost=100 LUT-FF pairs, FSMCost=30 LUT-FF pairs.

The obtained costs are outlined in Table 4. The estimated costs show SW to HW execution time ratios ranging between 1.79 (hermitian transpose) and 10.31 (matrix multiplication). HW costs vary between $\{16,0\}$ (scalar subtraction) and $\{1426,14\}$ (matrix inversion) with units $\{$LUT-FF pairs, DSP48E slices$\}$.

The estimated costs are paired with the tasks of the application and are provided to the mapping framework. The mapping experiments are described in the following section.

Mapping Experiments

The mapping experiments are conducted with the mapping framework. The MMSE-IC case study is composed of a frame-based and a symbol-based part. The symbol-based part has to be performed for each received symbol [21] and is implemented in dedicated HW, thus we only consider the frame-based part (**B**1) for the case study. Therefore, the application model is the frame-based part of the application graph in Fig. 17.

The architecture model consists of the SW processor, and a reconfigurable hardware area part. The SW processor system consists of the Xilinx μBlaze processor with an embedded multiplier plus a timer to measure execution times. Configuration bitstreams are loaded from external on-board SRAM memory to the ICAP interface via the SW processor. The reconfigurable hardware is characterized by A_{reconf} in terms of LUT-FF pairs and DSP48E slices and the total HW cost in Table 5. The lowest total HW cost (7478 LUT-FF pairs) is obtained by using a single-processor pure SW solution, whereas the highest costs are obtained by utilizing 100 % of the remaining resources as reconfigurable HW. The reconfiguration time, t_{reconf} was estimated based on the expression in (3). The

Table 5: Architecture Models. The models have varying characteristics. t_{reconf} is the reconfiguration time for the reconfigurable area. Total HW Cost include the symbol-based part of the application.

Arch. Model	# SW proc.	A_{reconf} {[LUT-FF],[slices]}	t_{reconf} [10^3 cycles]	Total HW Cost {[LUT-FF],[slices]}
A	1	{25000,285}	479.2	{32478,288}
B	1	{5760,128}	110.6	{13238,131}
C	1	{2880,64}	55.8	{10358,67}
D	1	{1440,32}	27.9	{8918,35}
E	1	{0,0}	-	{7478,3}
F	2	{5760,128}	110.6	{20716,134}
G	2	{2880,64}	55.5	{17836,70}
H	2	{1440,32}	27.9	{16396,38}
I	2	{0,0}	-	{14956,6}
J	3	{0,0}	-	{22434,9}

numerator is the estimated reconfiguration bitstream file size [24].

$$t_{\text{reconf}} = \frac{\left(\frac{c_{v_i,p_k,\text{LUT}}}{A_{Total,LUT}} F_{\text{config}}\right) * 41 + W_{\text{overhead}}}{f_{\text{ICAP}}} \quad [\text{s}] \tag{3}$$

where the parameters are given as:

$c_{v_i,p_k,\text{LUT}}$	LUTs consumed by module i.
$A_{Total,LUT}$	Total LUTs in the FPGA.
F_{config}	Total number of configuration frames.
W_{overhead}	Bitstream overhead in [32 bit words].
f_{ICAP}	ICAP configuration speed [Hz].

The parameters F_{config} and W_{overhead} are found in [24] to be 15252 frames and 272 words respectively. f_{ICAP} is set to 100 MHz. The multiplication by 41 is performed as each configuration frame is configured by 41 words [32 bit/word]. The values for the parameters are specific for each FPGA.

The application model, architecture model, and cost estimate library are inputs to the mapping framework. The mapping framework is then invoked for all application models, resulting in a binding and a schedule. The schedules do concern the frame-based part and are characterized by their makespan, i.e. the time from the start of the first task, to the finish of the last task.

The relation between the total HW cost and makespan is shown in figure 18. The solutions with 2 and 3 processors (F-J) have the lowest makespan of 4713 cycles. There is no difference between the two, as only two of the operations can be performed in parallel at one time. The architecture with 1 processor (A-E) has only slightly higher makespan (4767 cycles). For all the generated schedules, all operations were performed in SW as the mapping framework took into account that reconfiguration time is long compared to the SW schedule.

In order to evaluate a schedule using reconfiguration, we modified the α value to 0.03, such that reconfiguration time was partially neglected in the HW/SW partitioning. This resulted in the Pareto-chart in Fig. 19. The pure SW solutions were disregarded, and in such cases the architecture model D gave the lowest makespan, with a value of 83833. The corresponding schedule contained three supernodes with the operations of

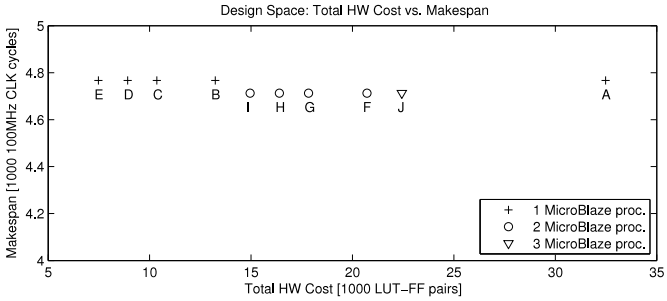


Figure 18: Pareto-chart describing the relation between total HW cost and the makespan of the resulting schedule for the frame-based part of the MMSE-IC equalizer. ($\alpha = 1$, RCost=100, FSMCost=30).

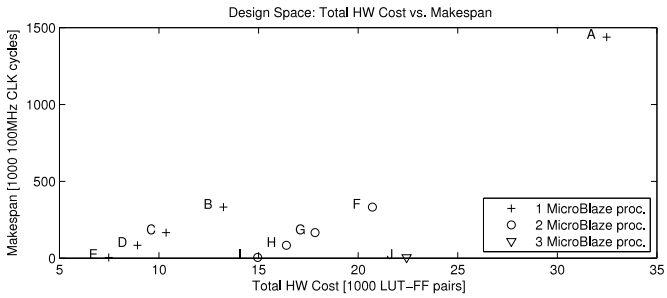


Figure 19: Pareto-chart describing the relation between total HW cost and the makespan of the resulting schedule for the frame-based part of the MMSE-IC equalizer. ($\alpha = 0.03$, RCost=100, FSMCost=30).

matrix multiplications task 2, 7, and 10, respectively. The schedule was selected for trial implementation on the Virtex-5 SX50T FPGA, and is shown in Fig. 20.

Furthermore, we also implemented architecture model E to evaluate the full SW solution. The results were based on a reconfiguration time of 27861 cycles as seen in Table 5. However, the implementation showed that using the HWICAP to load configuration data from SRAM resulted in a reconfiguration time of 1196900 cycles as opposed to the estimated 27861 cycles (approx. 43 times higher than estimated). This result was included in the schedule, in order to correct for this deviation. An estimated makespan of $3.6E6$ is thus obtained.

The results clearly indicate that reconfiguration time is a limiting factor for such applications. The reconfiguration controller should be optimized in order for the reconfiguration time to be reduced to the estimated reconfiguration time. The reconfiguration interface frequency of 100 MHz is shown possible by Manet et al. [4], however, we obtained only a frequency of approx. 2 MHz. This reduction in speed was identified to originate from the SW driver. The results from the framework clearly showed that reconfiguration time is dominating the overall execution time for this case study. However, when the estimated reconfiguration times were adjusted to the measurements, the output

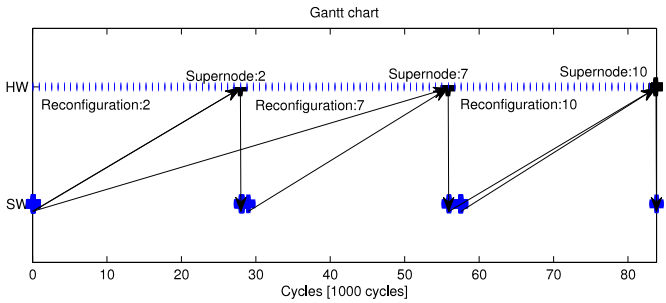


Figure 20: Gantt-chart for the obtained schedule for architecture D, using the mapping framework. ($\alpha = 0.03$, RCost=100, FSMCost=30).

Table 6: Implementation Result in 100 MHz clock cycles: Difference between framework estimated makespan and implementation result on Xilinx Virtex-5SX50T FPGA. Deviation is the deviation of the estimate related to the measured makespan.

Implementation	Estimated makespan	Measured makespan	Deviation
Reconfiguration only	27861	1195031	98 %
Arch. model E - Full SW	4767	4755	0.2 %
Arch. model D	83833	3594008	98 %
D, upscaled t_{reconf}	3595498	3594008	< 0.1 %

schedule of the mapping framework were within less than 0.1% from the measurements.

7 Discussion

The framework has been developed to provide designers with a schedule to be implemented on heterogeneous, reconfigurable architectures consisting of SW-processors and FPGAs. The simulations based on abstract applications and architectures showed that the parameter α should be set to 1, so that reconfiguration time was fully taken into account by the scheduling algorithms. This gave the lowest makespan.

Simulations also showed that level-based temporal partitioning proved better results than clustering-based. This was unexpected due to the fact that clustering-based partitioning has been developed to reduce communication between temporal partitions. However, this is explained by the fact that the level-based approach is less likely to create paths outside a single temporal partition; thus the number of partitions is potentially lower and leads to fewer time-consuming reconfigurations.

The reconfiguration time of today's reconfigurable devices, mainly FPGAs which are considered as fine-grain reconfigurable devices, is relatively high. This was also seen for both estimation and measurement. However, due to the SW driver, reconfiguration speed was observed to be 2 MHz instead of 100 MHz.

The mapping framework takes as input an architecture model where the size of the reconfigurable HW is given. This actually limits the use of dynamic partially reconfig-

urable HW. However, these limitations do also exist in the design flow of Xilinx FPGAs, where the size and shape of the reconfigurable regions must be set at compile-time [25]. The cost estimates are required for the framework, and it may require a significant effort to obtain these estimates. However, according to the Embedded Market Study [7], 57% of the embedded design projects are upgrades of functionality or HW. Furthermore, 72% of the designs reuse hardware IP, so we believe that for many cases, it will be an acceptable workload for obtaining these estimates.

The MIMO equalizer, that has been considered as a case study did not obtain the lowest makespan by using reconfigurable HW. The lowest cost was obtained by pure SW execution in one processor, due to the low level of parallelism in the application. Furthermore, the speedup obtained by utilizing reconfigurable HW did not justify the high reconfiguration time. However, the case study was included to verify the generated schedule, which performed as expected.

8 Conclusion and Outlook

The work presented in this paper was centered around a mapping framework for heterogeneous, reconfigurable architectures. The algorithms used to schedule execution and reconfiguration have been described, and they have been evaluated, both via simulations and via a case-study.

As input model the framework uses a general task graph that can cover both fine-grain and coarse-grain applications. The obtained schedules were very close (0.2%) to the implementation result for the investigated case-study. The lowest makespan was obtained by utilizing the mapping framework with level-based partitioning and an α value of 1. In order to improve the work, we propose the consideration of other HW/SW partitioning schemes, possibly with feedback from binding and scheduling. These considerations and investigations should be followed by experiments with architectures with multiple HW units.

The observed case-study suffered from a large reconfiguration overhead, which did not make the use of reconfiguration feasible for this particular application and architecture combination. It is clear that reconfiguration overhead has a large impact on the feasibility of reconfigurable architectures. However, our studies indicate that for applications, e.g. multimedia processing, with a high degree of inherent parallelism, there will be a larger speedup gain by HW execution as compared to SW execution. Another useful property of such applications is that it may be possible to alleviate the reconfiguration overhead when there is a lower reconfiguration to execution time ratio. Furthermore, reconfiguration time is technology dependent, so we find it reasonable to assume that reconfiguration overhead will be lower in the future.

We propose the investigation of other case-studies where the reconfiguration to execution time ratio is lower, possibly within multimedia processing with massively parallel algorithms.

We see the framework as a tool for designers to make decisions of the design of applications for heterogeneous, reconfigurable multiprocessing architectures.

References

[1] R. Hartenstein, "Trends in reconfigurable logic and reconfigurable computing," in *9th International Conference on Electronics, Circuits and Systems*, vol. 2, September 2002, pp. 801–808.

[2] J. P. Delahaye, G. Gogniat, C. Roland, and P. Bomel, "Software radio and dynamic reconfiguration on a dsp/fpga platform," in *3rd Karlsruhe Workshop on Software Radios*, 2004.

[3] M. Ihmig, N. Alt, C. Claus, and A. Herkersdorf, "Resource-efficient sequential architecture for fpga-based dab receiver," in *5th Karlsruhe Workshop on Software Radios*, 2008.

[4] P. Manet, D. Maufroid, L. Tosi, G. Gailliard, O. Mulertt, M. D. Ciano, J.-D. Legat, D. Aulagnier, C. Gamrat, R. Liberati, V. L. Barba, P. Cuvelier, B. Rousseau, , and P. Gelineau, "An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications," *EURASIP Journal on Embedded Systems*, vol. 2008, pp. 1–11, 2008.

[5] T. Becker, W. Luk, and P. Y. K. Cheung, "Energy-aware optimisation for run-time reconfiguration," in *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 55–62.

[6] T. Komuro, T. Tabata, and M. Ishikawa, "A reconfigurable embedded system for 1000 f/s real-time vision," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 20, no. 4, pp. 496–504, April 2010.

[7] E. T. Group, "Embedded market study 2010," May 2010, `www.techonline.com`.

[8] S. Borkar, "Thousand core chips - a technology perspective," in *44th ACM/IEEE Design Automation Conference*, June 2007, pp. 746–749.

[9] M. Kaul and R. Vemuri, "Optimal temporal partitioning and synthesis for reconfigurable architectures," in *Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 389–397.

[10] K. M. G. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *Transactions on Computers*, vol. 48, no. 6, pp. 579–590, June 1999.

[11] S. Banerjee, E. Bozorgzadeh, and N. D. Dutt, "Integrating physical constraints in hw-sw partitioning for architectures with partial dynamic reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 11, pp. 1189–1202, November 2006.

[12] J. Noguera and R. M. Badia, "A hw/sw partitioning algorithm for dynamically reconfigurable architectures," in *Proceeding of Design, Automation and Test in Europe*, March 2001, pp. 729–734.

[13] K. S. Chatha and R. Vemuri, "An iterative algorithm for hardware-software partitioning, hardware design space exploration and scheduling," *Design Automation for Embedded Systems*, vol. 5, pp. 281–293, 2000.

[14] M. D. Galanis, G. Dimitroulakos, and C. E. Goutis, "Partitioning Methodology for Heterogeneous Reconfigurable Functional Units," *The Journal of Supercomputing*, vol. 38, pp. 17–34, 2006.

[15] F. Dittmann, M. Götz, and A. Rettberg, "Model and Methodology for the Synthesis of Heterogeneous and Partially Reconfigurable Systems," *IEEE Parallel and Distributed Processing Symposium*, pp. 1–8, 2007.

[16] A. Itradat, M. Ahmad, and A. Shatnawi, "Scheduling of dsp algorithms onto heterogeneous multiprocessors with inter-processor communication," in *IEEE Northeast Workshop on Circuits and Systems*, June 2005, pp. 95–98.

[17] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175–187, February 1993.

[18] *TMS320C6000 Programmer's Guide*, Texas Instrument Inc., rev. J, April 2010.

[19] *Blackfin Processor Core Basics*, Analog Devices Inc., `http://www.analog.com/en/embedded-processing-dsp/Blackfin/processors/Blackfin_core_basics/fca.html`, June 2010.

[20] A. Popp, Y. Le Moullec, and P. Koch, "Scheduling temporal partitions in a multiprocessing paradigm for reconfigurable architectures," in *NASA/ESA Conference on Adaptive Hardware and Systems*, 2009, pp. 230–235.

[21] D. Karakolah, C. Jégo, C. Langlais, and M. Jezequel, "Design of an iterative receiver for linearly precoded mimo systems," in *IEEE International Symposium on Circuits and Systems*, May 2009, pp. 597–600.

[22] D. Karakolah, C. Jégo, C. Langlais, and M. Jezequel, "Architecture dedicated to the mmse equalizer of iterative receiver for linearly precoded mimo systems," in *3rd International Conference on Information and Communication Technologies: From Theory to Applications*, April 2008, pp. 1–6.

[23] *Virtex-5 FPGA User Guide (UG190)*, Xilinx Inc., May 2008.

[24] *Virtex-5 FPGA Configuration User Guide (UG191)*, Xilinx Inc., June 2009.

[25] *Partial Reconfiguration User Guide (UG702)*, Xilinx Inc., May 2010.